

Lean on Vampire *Models*

Kernel-Checked Finite Counter-Models for Lean 4: a Verified Reflective Checker, a Measured Scaling Recipe, and a Million-Fact Certificate

Michael Jagdeo, Exponent Labs LLC

Jeff Dumps

Whitepaper, October 2026. Code and data: <https://github.com/exponentlabshq/lean-countermodel>

Contents

Abstract	1
List of tables and figures	2
1. Introduction	3
2. Background and Problem	4
3. Method: Find, Rewrite, Compile, Certify	5
4. The Verified Checker: Data Structures and Trust	14
5. Scaling: Design Patterns, Data Structures and Measured Effects	21
6. Evaluation and Case Studies	27
7. Related Work	36
8. Limitations and Future Work	40
9. Conclusion	42
Appendix A: Reproducing Every Number	43
References	44

Abstract

To show that axioms do **not** imply a claim, one exhibits a structure in which the axioms hold and the claim fails. Isabelle has had a finder for such counter-models for over fifteen years. Lean 4 can import *proofs* from automated provers and search for concrete counterexamples in fixed types, but it has no general, automated way to obtain counter-models of abstract first-order theories; projects build them by hand.

We present `lean-countermodel`, which turns finite models found by the Vampire prover into certificates checked by Lean’s kernel. Certificates take two forms:

- **Shallow:** the model as a Lean inductive type, with one theorem per formula.
- **Reflected:** a Lean library defines first-order formulas as data, a Boolean evaluator, and ordinary first-order structures. It proves, once, that for a model whose domain is non-empty and whose functions are closed, and for closed, well-formed formulas, the evaluator agrees with **ordinary first-order satisfaction** in a structure on `Fin n`. A certificate checks validity, well-formedness and truth in one kernel evaluation, and concludes that this structure satisfies every formula. It uses only Lean’s standard axioms (`propext`, `Quot.sound`).

Vampire is untrusted: a model that is ill-formed or that falsifies the encoded problem yields no certificate. The translation from TPTP into Lean remains trusted code, tested against an independent evaluator of the *original* formulas.

Most of the work is in making the check finish. One benchmark clause has about 10^{11} cases, and a 10,000-formula file overflowed Lean 4.13’s kernel. We give a recipe of classic patterns: backtracking via miniscoping, bitsets, balanced trees, hierarchical aggregation, content-addressed caching, blocking and reflection. Each is kept or rejected on measurement, and we report the negative results. For our large ground-fact certificates, the cost was dominated by elaborating statements rather than evaluating them. Quantifier-heavy checking has a separate evaluation bottleneck.

Evaluation:

- **Three samples of 100 TPTP problems** with status Satisfiable or CounterSatisfiable [34] (fof/cnf), inputs rebuilt byte for byte from the official TPTP release. After the round-4 repair (below), all three were rerun at one commit. On the **third sample, selected before its first execution**, Vampire found 67 finite models within 30 s, and **59 were certified under our strict kernel-only policy (88.1% of models, 59% of all problems)**; 3 more with native evaluation, which enlarges the trusted base. The development set gives 71 of 78 (91.0%) and the second sample 55 of 62 (88.7%), with 2 native-assisted each. Conditional certification rates are similar across the three; full-pipeline success varies more (71%, 55%, 59%), mainly with how many models the finder returns. Uncertified candidates either hit Lean’s time or instance-search limits or, in one case (SY0595–1), crashed in certificate generation; one further input crashed the translator. All are reported as failures. All three samples have now been exposed to development, so these are descriptive results, not an untouched test.
- **A source-fidelity correction.** Peer review found that the record reader could skip a legal TPTP record without error. **Eight earlier certificates omitted source premises and were withdrawn. The repaired pipeline subsequently certified their complete source problems. Unchanged strict totals do not validate the superseded certificates.** One previously native-assisted result timed out in the rerun and is reported as a failure. Records are now read totally and fail-closed, the test oracle reads them independently, and every certified result is checked against its original source file (§6.2.2).
- **A million-fact theory.** NLP263+1 (1,026,857 facts) is certified in a median 5.2 minutes reflected, against 14.1 minutes shallow, in a controlled, repeated comparison on the same Lean version, worker budget and laptop: about 2.7× in wall time and 2.6× in CPU time. These timings were measured at an earlier commit (62d43d6). The final code generates a byte-identical reflected certificate; its shallow certificate differs only by renamed helper declarations and one conjunct (`Nonempty T`), and was not re-timed (§6.3.1). Its model has one element, so this measures certificate throughput, not combinatorial checking.
- **A published Lean formalization.** The tool reproduces, from axioms alone, both hand-built counter-models of Nakamura’s Lean reconstruction of Spinoza’s *Ethics*, with smaller models. It also produces a certified negative-dependency map of that reconstruction’s register.

Keywords: Lean 4 · finite model finding · counter-models · Vampire · TPTP · proof by reflection · certificates · interactive theorem proving

List of tables and figures

- Figure: 3.1 Pipeline
- Table: 3.3 Model parse: `model(path)`
- Table: 3.4 Rewrite: `problem, literals, clause, miniscope`
- Figure: 3.4 Rewrite: `problem, literals, clause, miniscope`
- Table: 3.5 Compile (shallow): `emit`
- Table: 3.8 Trust boundary
- Figure: 3.9 Worked example: `transitive_cnf`
- Table: 3.9 Worked example: `transitive_cnf`
- Table: 3.10 Independent checks (`test_cm.py`)
- Table: 4.1 Data structures
- Table: 4.2 Models in generated certificates
- Table: 4.3 Problem encodings
- Figure: 4.3 Problem encodings
- Table: 4.4 Soundness
- Table: 4.5 Shallow vs reflected certificates
- Table: 4.6 Independent checks of the reflected path (`test_cm.py`)
- Table: 5. Scaling: Design Patterns, Data Structures and Measured Effects
- Table: 5.1 Master table
- Table: 5.3 NLP263+1: shallow vs reflected
- Figure: 5.4 Illustrations
- Table: 6.1 Case study A: a published Lean formalization (Spinoza’s *Ethics*)
- Table: 6.1.1 The assumption-dependency map
- Table: 6.2 Case study B: three samples of TPTP
- Table: 6.2.1 The second and third samples

- Table: 6.2.2 Round-4 repair: record omission, and what it changed
- Table: 6.3.1 Controlled comparison
- Table: 6.4 Summary
- Table: 7.6 Comparison
- Table: 8.1 Supported input
- Table: 8.2 Limitations
- Table: Appendix A: Reproducing Every Number

1. Introduction

Formal developments are full of *negative* claims:

- this axiom is independent of those;
- this hypothesis is needed;
- this conjecture is false;
- these assumptions are consistent.

Each is shown the same way. One exhibits a **counter-model**: a structure that satisfies the premises and falsifies the conclusion. Automated model finders such as Mace4 and Vampire’s finite-model builder find small counter-models in milliseconds. Isabelle users have had Nitpick for this since 2010 [3].

Lean 4 users have no equivalent for abstract theories. Lean can replay proofs from automated provers, and it can search for concrete counterexamples over types it already knows, such as `Nat` and `Fin n`. Lean can *represent* any finite structure, but it has no general automated front end that *finds* one, a universe together with interpretations of uninterpreted symbols, and certifies it. So Lean projects build their counter-models by hand. For example, the two non-derivability results in a recent Lean formalization of Spinoza’s *Ethics* rest on hand-built models of 4 and 3 elements [22, §4.4]. The Equational Theories Project [1] built project-specific machinery to check about 587,000 magma counterexamples.

1.1 Motivation: philosophy written in geometric order

Spinoza’s *Ethics* was published in 1677 as *Ethica ordine geometrico demonstrata*, “demonstrated in geometrical order”. Like Euclid, it gives definitions, axioms and propositions, each with a demonstration citing earlier ones. A philosophy written this way invites machine checking. Its central arguments have been disputed for three and a half centuries; whether Proposition 5 (“no two substances share an attribute”) follows from what Spinoza states is one example.

Formal philosophy has long used automated provers and model finders on such questions [28]. Examples include computational metaphysics with Prover9 and Mace [29], a computationally discovered simplification of the ontological argument [30], a computer-assisted reconstruction of an ontological argument in the *Ethics* itself [31], and recent Lean 4 work on Gödel’s and Scott’s ontological arguments [21]. A research agenda for this field describes a seven-stage workflow, with the companion paper `paper/formal-philosophy.md` as the starting point: reconstruct the text, axiomatize it, explore with an automated prover, check candidate models independently, verify in a proof assistant, *ablate* each assumption, and review the interpretation. It frames the field’s most important output as **a map of the assumptions that hold a theory together**, and we know of no tool that produces such a map with certified counter-models.

`lean-countermodel` implements stages 3–6 for first-order reconstructions. Applied to a published Lean formalization of the *Ethics* [22], it produces that map (§6.1.1), with every “this assumption is needed” backed by a counter-model that Lean’s kernel checks.

1.2 Contributions

This paper. We connect Vampire’s finite-model builder to Lean’s kernel, for untyped first-order problems in TPTP (`fof` and `cnf`; §8 lists the unsupported features), and we make the result scale. The contributions are:

1. **A tool, `lean-countermodel`**, which compiles a Vampire model into a Lean certificate. Vampire is untrusted: a model that is ill-formed or falsifies the encoded problem yields no certificate (§3).
2. **A verified reflective checker** (`Checker.lean`). It provides first-order formulas as data, a Boolean evaluator, and ordinary first-order structures, and proves once that, for a valid model (non-empty domain, closed functions) and closed, well-formed formulas, the evaluator agrees with ordinary satisfaction on `Fin n (sem_iff_sat)`. Certificates conclude that this structure satisfies every formula, with Lean’s standard axioms only (§4).

3. **A measured scaling recipe.** Classic algorithmic patterns, each kept or rejected on measurement, take certificates from about 10^{11} cases, and from files that crash Lean’s kernel, to a million-fact problem certified in a median 5.2 minutes (range 5.2–5.8, three runs). Three findings, each measured on our generated workloads and stated with that scope in §5 and §8:
 - for our large ground-fact certificates, the cost was dominated by **elaborating statements, not evaluating them**;
 - on Lean 4.13, with our generated pattern and default options, **files of more than about 20,600 declarations failed** with kernel “deep recursion”;
 - in our decoder, **lazy kernel evaluation punished nested data** (§5).
4. **An evaluation.** It covers three 100-problem samples of TPTP problems with status *Satisfiable* or *CounterSatisfiable*: a development set (not uniformly drawn), a second uniform sample that was later used to find and fix defects, and a third uniform sample run once with frozen code; a 1,026,857-fact problem; and a published Lean formalization (§6).
5. **An assumption-dependency map** of Nakamura’s register for *Ethics* I (§6.1.1), produced by a reusable ablation tool (`ablate.py`).
6. **A trust case study.** Four real bugs in the trusted rewrite and naming layers, one of which made Lean certify a statement other than the intended one, were each caught by an independent check. Peer review then led us to two more: silent misreading of unsupported TPTP syntax, now closed by fail-closed parsing, and a TPTP variable named τ capturing the domain type (Lean rejected it; variables are now primed), and to a kernel-checked validity theorem for the checker’s semantics. A third-round audit of the benchmark inputs against the official TPTP release found that include expansion also expanded includes inside comments, which changed two inputs; neither had been certified (§4, §5, §6.2, §8).

What is not new. Model finding, kernel-checked finite certificates and proof by reflection are each well established (§7). The closest predecessor is Goubault-Larrecq’s extraction of finite models of first-order clause sets with Coq-checkable model-checking proofs [32]; finite model theory itself has been mechanized in Coq [33]. Our contribution is a specific, implemented combination: general untyped first-order models from Vampire, certified in Lean 4 by a verified reflective evaluator with a validity theorem, at the scale measured here. A focused search found no implementation with this combination; that is not a claim of first-ever status.

2. Background and Problem

2.1 Counter-models

A first-order *problem* is a set of axioms Γ and, optionally, a conjecture φ . A **counter-model** is a structure $\mathcal{M}$ with $\mathcal{M} \models \Gamma$ and $\mathcal{M} \not\models \varphi$. By soundness of first-order logic, a counter-model proves $\Gamma \not\models \varphi$. If there is no conjecture, a model of Γ shows that Γ is consistent. A **finite** counter-model has a finite universe $\{0, \dots, n - 1\}$ and finite tables for every constant, function and predicate. Checking it is decidable by enumeration, but naive enumeration costs n^v cases for a formula with v variables.

2.2 TPTP and Vampire

TPTP [23] is the standard problem library for automated theorem proving. Its 26,990 problems each carry a status. **2,561 of them are untyped first-order (fof or cnf) with status *Satisfiable* or *CounterSatisfiable*.** These statuses mean a model exists, not that a *finite* one does: the SZS ontology has separate *FinitelySatisfiable* and *FinitelyCounterSatisfiable* statuses [34]. A theory may have only infinite models, so a finite-model finder can fail on these problems for reasons other than time. Vampire [24] includes a finite-model builder (`-sa fmb`), which prints models as TPTP `tff` tables. We use it as an untrusted oracle.

2.3 Lean 4’s kernel and decide

Lean 4 [25] is an interactive theorem prover whose small kernel checks proof terms. For a decidable proposition P :

- **decide** evaluates the `Decidable P` instance in the elaborator, then has the kernel re-check it;
- **decide +kernel** (Lean ≥ 4.15) asks the kernel alone to evaluate;
- **native_decide** compiles the check to machine code. It is fast, but it adds the compiler to what must be trusted. Since Lean 4.29, each such computation is recorded as its own axiom, with `_native` in its name [35].

Our strict policy accepts only kernel checks: `#print axioms` must list at most the standard axioms `propext` and `Quot.sound.cm.py` never emits `native_decide`. The benchmark harness tries it only after a kernel timeout, and reports

such certificates in a separate, weaker category.

2.4 The problem

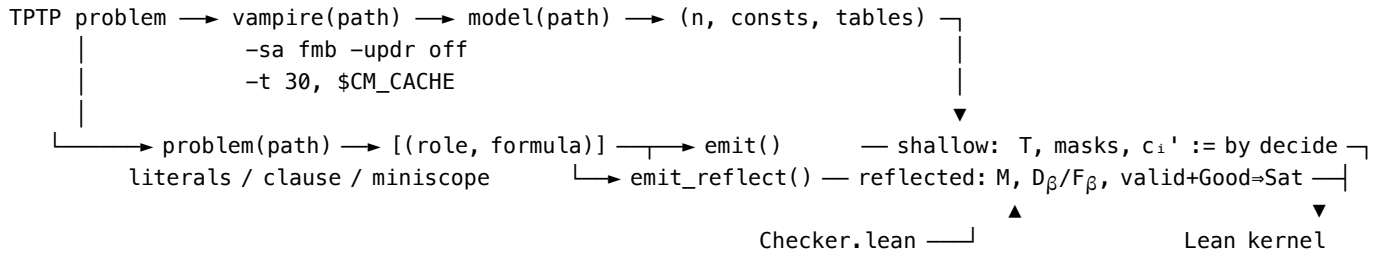
Given a TPTP problem and Vampire’s finite model, produce a Lean file whose main theorem states, faithfully, that the model satisfies every axiom and falsifies the conjecture, and which **Lean’s kernel accepts in reasonable time and memory**. Three difficulties make this hard:

1. **Search size.** Brute-force checking is n^v per formula: about 10^{11} for one clause in our benchmark.
2. **Elaboration size.** A problem can have over a million formulas. Lean’s per-declaration costs (parsing, elaboration, typeclass search, kernel type checking) dominate, and Lean 4.13 hits hard limits: kernel recursion depth, and declarations per file.
3. **Fidelity.** Every rewrite that makes checking fast is trusted code. A wrong rewrite can make Lean certify a statement *other than* the intended one, and we observed exactly that (§5).

3. Method: Find, Rewrite, Compile, Certify

cm.py turns a TPTP problem (fof or cnf) into a Lean 4 proof that a finite structure satisfies axioms $\wedge$ $\neg$ conjecture. An untrusted search finds the structure; a Python translation states the claim; Lean’s kernel checks it.

3.1 Pipeline



3.2 Find: vampire(path)

vampire runs \$VAMPIRE (default vampire) with `-sa fmb -updr off -t 30`: the finite-model-building saturation algorithm, a 30 s limit, and unused-predicate-definition removal disabled. Without `-updr off`, Vampire may settle a problem by deleting unused predicates and print no model.

If `$CM_CACHE` is set, the output is stored under `sha256(problem bytes || flags || vampire --version)`. A cache hit skips Vampire entirely, so a Lean upgrade reruns only Lean. Vampire’s answer is classified by an explicit table (cm.classify). **Definite answers** are cached: a model (Satisfiable or CounterSatisfiable, with termination reason Satisfiable); a refutation (Theorem, Unsatisfiable or ContradictoryAxioms, with Refutation); or a give-up (GaveUp, with Refutation not found, incomplete strategy). **Resource limits** are recorded as *no model* and never cached: termination reason Time limit or Memory limit, with no status or Timeout, MemoryOut or ResourceOut. Vampire 5.1 reports a timeout this way, with exit status 1 and no SZS status. **Everything else** is a *finder error*: an SZS Error, InputError or OSError, any status or termination reason outside the table, no status at all, or an executable that cannot run. vampire raises with the exit code and diagnostics, nothing is cached, and the benchmark records *finder error*, never *no model*. All 401 cached answers, and every uncached outcome in the final ledgers, fall in the first two classes (test_recorded_vampire_answers_classify_as_before). A cache entry is used only if it classifies as a definite answer itself: any other entry, such as an error response left by an earlier version, is ignored and Vampire runs again (test_a_cached_error_is_not_a_search_outcome; found in the closing review).

3.3 Model parse: model(path)

model returns (n, consts, tables):

Field	Source in Vampire output	Python type
n	count of declare_\$i<k> lines	int

Field	Source in Vampire output	Python type
consts	tff(<c>_definition, axiom, c = 'fmb_\$i_k') → element k; tff(<p>_definition, axiom, (~)p) → bool	{name: int \ bool}
tables	each (~)f('fmb_\$i_...', ...) (= 'fmb_\$i_k');= k makes a function entry, a bare (negated) atom a predicate entry	{symbol: {args tuple: int \ bool}}

Elements are numbered 1 ... n as Vampire prints them. If the output has no `FiniteModel`, model exits with no `finite counter-model` found and the SZS status; for a theorem this is the correct outcome(`test_a_theorem_has_no_counter_model`).

Reading records: statements(path). Before any formula is translated, `statements` reads the whole file and must account for every character: each is whitespace, part of a % comment, or part of a complete `fof` or `cnf` record `kind(name, role, formula[, annotations])`. It walks the text once, respecting quoted tokens (whose %, commas and parentheses are not syntax), comments between any two tokens, and nesting depth. It splits each record at its top-level commas, so annotations are never read as part of the formula. A name must be a lower-case word, a single-quoted word (which may contain punctuation, as in `'thm.bool.TRUTH'`) or an integer. The role must be one that asserts a formula: `axiom`, `hypothesis`, `definition`, `assumption`, `lemma`, `theorem`, `corollary`, `plain`, `negated_conjecture` or `conjecture`. Anything else is rejected with its line number and no certificate is produced: another statement kind (`tff`, `include`, ...), a malformed or unterminated record, a bracket closed by the wrong kind (`(p)`), an empty field (`fof(a, axiom, p,)`), an unknown role, or a `/* */` comment. Within a formula, `lean()` and `tokens()` also check structure as they read: a parenthesis or argument list must close with `)`, a quantifier needs `[, variables, ]` and `:`, and a formula that ends early is an error. `problem` and `ablate.py` both read records only through `statements`.

This replaced a regular expression that *searched* for records rather than *reading* the file. A record it could not match was skipped without error: a quoted name containing punctuation, or a comment or whitespace between `fof` and `(`. The round-4 review found this with two contradictory two-record files, each of which lost its first axiom. In the benchmark it dropped records from 9 inputs (§6.2.2).

Lexing within formulas is shared by every stage: `TOKENS` treats a quoted name `'...'` (including escaped quotes and %) as one token, and `uncomment` strips % comments only outside quoted names.

3.4 Rewrite: problem, literals, clause, miniscope

`decide` and the reflected evaluator both evaluate bounded $\forall$ and $\exists$ lazily. A clause with v variables over n elements has n^v cases unless a false literal prunes its subtree. The rewrite moves each quantifier as close as possible to the first literal that uses its variable.

`problem(path)` returns `[(role, formula)]`, one entry per record that `statements` read. For each it calls `literals(f, cnf)`, which calls `clause(tokens, scope)` to flatten the formula into `(scope, disjuncts)`. The result is rewritten by `miniscope(scope, lits)` when the formula is `cnf`, or when it is `fof`, has a non-empty scope, and flattens into more than one literal. Every other formula is kept verbatim.

clause rules (applied recursively, on tokens; `unwrap` drops enclosing parentheses, `split` splits at a top-level operator):

Input shape	Output	Condition	Equivalence used
$! [\bar{X}] : B$	<code>clause(B, scope $\cup \bar{X}$)</code>	B contains no top-level binary operator, and $\bar{X} \cap \text{scope} = \emptyset$	$\forall \bar{X} B$ is $\forall \bar{X} B$; the variables become part of the clause's universal prefix
$A_1 \ \& \ \dots \ \& \ A_k \Rightarrow B$	<code>[$\neg A_1, \dots, \neg A_k$] ++ <code>clause(B)</code></code>	top-level $\Rightarrow$	$A \rightarrow (B \rightarrow C) \Leftrightarrow (A \wedge B) \rightarrow C \Leftrightarrow \neg A \vee \neg B \vee C$
$D_1 \ \backslash \ \dots \ \backslash \ D_k$	<code>clause(D₁) ++ ... ++ <code>clause(D_k)</code></code>	top-level $\backslash $	associativity of $\vee$
anything else	one literal	—	—

Two side conditions carry the soundness of the flattening.

1. **TPTP binding.** In TPTP, $! [X] : A \Rightarrow B$ parses as $(\forall X A) \rightarrow B$: the quantifier binds only the unit that follows it. `clause` pulls $\forall$ out only when its body B is a single unit (whole: no top-level $\&$, $|$, $\Rightarrow$, $\Leftarrow$). An earlier version read $! [X] : A \Rightarrow B$ as $\forall X (A \rightarrow B)$; on three TPTP problems the altered formula still held, and Lean certified a different statement (README §5). The regression is covered by `examples/quantifier_scope.p`.
2. **Fresh variables only.** A quantifier is pulled into the clause's prefix only if none of its variables is already in scope. Pulling a re-bound X out would capture occurrences of an outer X . When the condition fails, the subformula stays one literal with its own quantifier.

Pulling $\forall \bar{X}$ out of a disjunct is sound because, under the freshness condition, $\bar{X}$ does not occur in the other disjuncts: $A \vee \forall X B \leftrightarrow \forall X (A \vee B)$ for $X \notin \text{FV}(A)$.

For `cnf`, `literals` starts with `scope = VARS(f)`: the clause's free variables are implicitly universal. `VARs` collects $[A-Z]\backslash w^*$ identifiers after deleting quoted names, so `'X'` is never a variable.

miniscope(scope, lits, bound) rebuilds the clause as nested quantified disjunctions:

```
miniscope(scope, lits, bound):
  lit := argmin over lits of |(VARS(l) n scope) \ bound|      -- greedy: fewest new variables first
  body := lit
    (lit | miniscope(scope, lits \ lit, bound u new(lit)))    otherwise
  return ![new(lit)]: body if new(lit) ≠ ∅ else body
```

Each variable is bound immediately before the first literal that uses it, via $\forall X (A \vee B) \leftrightarrow A \vee \forall X B$ when $X \notin \text{FV}(A)$. The greedy order chooses the literal that introduces the fewest new variables. That can favour a useful guard, so lazy evaluation prunes early, but it does not estimate selectivity, which depends on the model's data (§5.2). Each literal is parenthesized by `literals`, because $! [Z] : A \Leftarrow B$ would otherwise bind only A .

3.5 Compile (shallow): emit

`emit(path, n, consts, tables)` produces a self-contained Lean file:

Model part	Lean form
Universe	inductive T \ e1 \ ... \ en deriving DecidableEq, T.all, T.mem, T.idx, and Decidable (∀ x, p x) / Decidable (∃ x, p x) instances via <code>decidable_of_iff</code> over T.all
Constant c	abbrev «c» : T := T.ek
Propositional symbol p	abbrev «p» : Prop := True \ False
Predicate p of arity k	abbrev «p» (x0 ... x(k-1) : T) : Prop := (MASK >>> (1 * x0.idx + n * x1.idx + ...)) % 2 = 1
Function f	def «f» : T → ... → T by a pattern-matching table
Formula i (from <code>lean(f)</code>)	theorem c<i>' : <φ _i > := by decide; a conjecture is emitted as $\neg \phi$
Whole claim	theorem countermodel : Nonempty T ∧ <tree of ∧> := (<T.e1>, <tree of (<_, _>)>)

Nonemptiness. A Lean inductive type with no constructors is empty, and $\forall x : T, \text{False}$ holds over it, although no first-order structure satisfies it. So the domain's nonemptiness is part of the certified statement: `Nonempty T`, proved by `<T.e1>` and checked by the kernel like every other conjunct. `emit` also refuses a model of size 0 before writing anything (`test_shallow_certificate_proves_a_nonempty_domain`).

`tree(xs, pair)` builds balanced conjunctions and anonymous-constructor proofs: depth $\log_2 m$ instead of m , since a chain of 10,000 $\wedge$ overflows the kernel. One theorem per formula gives each `decide` fresh caches.

Every generated file starts with `set_option autoImplicit false`: an unbound name is an elaboration error, never a silently added implicit argument. `sym(k)` writes a TPTP symbol as `<k>` if it matches $[a-z]\backslash w^*$, and as `<'k'>` otherwise. Quoted names keep their quotes, so a constant `'X'` becomes `<'X'>` and cannot be captured by a bound variable X , and `'T'`

cannot collide with the type T (examples/quoted_capture.p, examples/quoted_names.p). The emitter’s own helper declarations carry a prime ($c0'$, $P0'$, $C0'$, $chunk0'$), which no TPTP lower-case word contains, and quoted names are emitted inside their quotes, so no source symbol can collide with them; a source symbol named `countermodel`, the one unprimed public name, is emitted quoted ('countermodel' is the same TPTP symbol). Peer review found the collision: a source constant $c0$ clashed with the helper theorem $c0$ (test_source_symbols_cannot_collide_with_generated_names).

Modular output. If the problem has more than `CM_CHUNK` formulas (default 2000), `emit` returns a dict of modules instead of one string:

Module	Contents
Base	options, T , instances, predicate and function tables
Names< k >	constant abbrevs, $CM_CHUNK \times 5$ (10,000 by default) per module
Chunk< j >	def $P< i>'$: Prop := ϕ_i , theorem $c< i>'$: $P< i>'$:= by unfold $P< i>'$; decide, def $C< j>'$ (balanced $\wedge$ of its $P< i>'$), theorem $chunk< j>'$: $C< j>'$
Main	theorem <code>countermodel</code> : $\text{Nonempty } T \wedge \text{<tree of } C< j>'>$

Constants live in separate `Names` modules because Lean 4.13 fails with “deep recursion” on files with more than about 20,000 declarations (comment in `emit`). No module restates the whole problem, and `lake` checks chunks in parallel.

3.6 Compile (reflected): `emit_reflect`

`emit_reflect(path, n, consts, tables)` does not state formulas as Lean propositions. It writes the model as a `CM.Model` with packed tables, writes the problem as packed natural numbers, and proves each block with one `decide +kernel` call through the verified evaluator in `Checker.lean` (§4).

- `tokens(f, ids)` translates a formula into the prefix token stream read by `CM.decode`. Variables become de Bruijn indices (`env.index(k)`, last-bound variable at index 0). Symbols become numbers via `ids = {"pred", "fn", "const"}`, assigned in sorted name order; propositional symbols are numbered after the table predicates, as 0-ary predicates. A conjecture is prefixed with opcode 4 ($\neg$).
- `as_fact(toks)` recognizes ground literals $(\neg)p(c_1, \dots, c_k)$ over constants and returns (polarity, predicate, [constant ids]). Facts go into fixed-width fact tables ($F< j>$, 128 per block, proved by `holds_facts`); all other formulas go into token blocks ($D< b>$, about 4096 tokens per block, proved by `holds_of_check`).
- Lookups in the generated `M`: predicates are one packed mask per symbol, read by `get (get MASKS p mw) (index n args) 1 == 1`; constants are packed 4096 per block, the block chosen by `bsearch`, a balanced tree of if $c / 4096 < k$ comparisons; functions are selected by `bsearch` over function ids, then read with `get ... (index n args) w`.
- The certificate proves `valid : M.valid fns = true` and `good : Good M fns problem` (well-formedness and truth, a balanced tree of `good_append` over the blocks), and states theorem `countermodel :  $\forall \phi \in \text{problem}, \phi.\text{Sat } (M.\text{toStructure } \dots) (\emptyset, \dots) []$` : ordinary first-order satisfaction (§4.4). Above 32 blocks (`per_part`) the output is split into `Base`, `Part< k>` and `Main` modules.

`bench.py`’s `run_hybrid` chooses reflection when the share of ground-literal formulas (`fact_share`) is at least 0.9, and the shallow path otherwise.

3.7 Certify

The kernel checks every `decide` (shallow) or `decide +kernel` (reflected) proof. `native_decide` is never emitted by `cm.py`; `bench.py` tries it only as a fallback after a shallow `decide` timeout and reports such problems separately as certified (native).

3.8 Trust boundary

Stage	Code	Trusted?	Why / how it is checked
Input preprocessing (benchmark only)	<code>bench.expand,</code> <code>bench/inputs.py</code>	Yes	Inlines plain <code>include('Axioms/...')</code> directives so <code>cm.py</code> reads one self-contained file. Includes inside comments or quoted names are left alone; selective, nested, unresolved and non-Axioms/ includes are refused (<code>test_include_expansion</code>). Every benchmark input is rebuilt from the official TPTP v9.3.1 release, and its original and expanded SHA-256s are checked against <code>bench/inputs.json</code> (<code>test_benchmark_inputs_match_their_s</code>
Find	Vampire	No	A wrong model makes decide fail. Mutation tests confirm that Lean rejects broken models.
Cache	<code>vampire()</code> , <code>\$CM_CACHE</code>	No	It returns Vampire's output for an identical problem, flag set and version; the output is untrusted anyway.
Model parse	<code>model()</code>	No , for reflected certificates	A misparse that leaves the domain is caught: <code>Model.valid</code> (non-empty domain, functions closed) and <code>Fm.wf</code> (constants in range, bound variables, checked arities) are kernel-checked parts of every reflected certificate. A misparse that stays in the domain yields a different structure. Lean then certifies that structure if, and only if, it satisfies the formulas. For shallow certificates, constants and function values are constructors of T , so they lie in the domain by construction, and nonemptiness is a kernel-checked conjunct of the statement ($\text{Nonempty } T$, §3.5).

Stage	Code	Trusted?	Why / how it is checked
Record reading	statements	Yes	Which formulas exist. It must account for every character of the file or reject it, so a record cannot be dropped silently (§3.3). Tested against the oracle's separately written record reader and against hand-counted fixtures (test_every_record_is_accounted_for, test_unreadable_records_are_rejected). The round-4 regressions run a model of part of a problem through both certificate paths, which Lean must reject (test_a_model_of_part_of_the_problem).
Rewrite	problem, clause, literals, miniscope	Yes	A non-equivalent rewrite changes what is certified. Tested by test_rewriting_preserves_meaning and regression examples.
Translation (shallow)	lean(), sym(), emit()	Yes	The Lean statement must mean the TPTP formula. Tested by the independent evaluator and by mutant agreement; the statement is readable in the .lean file.
Translation (reflected)	tokens(), as_fact(), symbol numbering	Yes	The Fm data must mean the TPTP formula. Tested by test_reflected_mutants_agree_with_p against an evaluator of the <i>original</i> formulas; #eval problem is a diagnostic that prints the decoded data (with numeric symbol ids), not a fidelity proof.
Evaluator and semantics	Checker.lean (Fm.check, decode, fact, Model.valid, Fm.wf)	No (proved)	check_iff (evaluator = bounded semantics) and sem_iff_sat (bounded semantics = ordinary satisfaction, for valid models and well-formed formulas) are kernel-checked theorems; countermodel_of_good combines them.
Certify	Lean kernel	Yes	Axioms: propext (shallow); propext, Quot.sound (reflected, per bench/nlp263-reflected-lean4.35.json).

3.9 Worked example: transitive_cnf

Input(examples/transitive_cnf.p):

```
cnf(transitive, axiom, ~ r(X,Y) | ~ r(Y,Z) | r(X,Z)).
cnf(a_before_b, axiom, r(a,b)).
cnf(not_symmetric, negated_conjecture, ~ r(b,a)).
```

problem rewrites the first clause. scope = {X, Y, Z}; all three literals introduce two new variables, so min takes the first, ~r(X,Y), binding X, Y; then ~r(Y,Z) introduces only Z:

```
! [X,Y]: ((~ r ( X , Y )) | ! [Z]: ((~ r ( Y , Z )) | ( r ( X , Z ))))
```

Vampire returns a 2-element model: a = 1, b = 2, and r true only at (1, 2). emit encodes r as mask $4 = 2^{(0 + 2 \cdot 1)}$, bit index $x0.idx + 2 \cdot x1.idx$. Generated Cm/transitive_cnf.lean (excerpt):

```
inductive T | e1 | e2
  deriving DecidableEq
...
abbrev «a» : T := T.e1
abbrev «b» : T := T.e2
abbrev «r» (x0 x1 : T) : Prop := (4 >>> (1 * x0.idx + 2 * x1.idx)) % 2 = 1

theorem c0' : (∀ X' Y' : T, (¬(«r» X' Y') ∨ (∀ Z' : T, (¬(«r» Y' Z') ∨ («r» X' Z'))))) := by decide
theorem c1' : («r» «a» «b»)) := by decide
theorem c2' : ¬(«r» «b» «a»)) := by decide

theorem countermodel : Nonempty T ∧ (((∀ X' Y' : T, (¬(«r» X' Y') ∨ (∀ Z' : T, (¬(«r» Y' Z') ∨ («r»
  X' Z')))))) ∧
  (((«r» «a» «b»)) ∧ (¬(«r» «b» «a»)))) :=
  (⟨T.e1⟩, ⟨c0', ⟨c1', c2'⟩⟩)
```

TPTP variables are emitted primed (X becomes X'). A TPTP variable cannot contain ', so no variable can shadow the domain type T or a Lean constant such as True.

In the reflected path, with ids = {pred: {r: 0}, const: {a: 0, b: 1}}, tokens produces:

Formula	Tokens	Route
transitive	9 9 6 4 0 0 2 0 1 0 0 9 6 4 0 0 2 0 1 0 0 0 0 2 0 2 0 0	token block D0
r(a,b)	0 0 2 1 0 1 1 → as_fact = (1, 0, [0, 1])	fact table F0
~r(b,a)	4 0 0 2 1 1 1 0 → as_fact = (0, 0, [1, 0])	fact table F0

Reading the first stream: 9 9 is ∀∀, 6 is ∨, 4 0 0 2 (0 1) (0 0) is ¬r(var 1, var 0), i.e. ¬r(X, Y), because Y, bound last, is index 0.

group_comm.p illustrates the other branch of problem: each fof formula there is a single equation under one !, so clause yields one literal, the ok test fails, and the formula is emitted unchanged, e.g. theorem c3 : ¬(∀ X' Y' : T, ((«mul» X' Y') = («mul» Y' X')))) := by decide over the 6-element model S₃.

3.10 Independent checks (test_cm.py)

The trusted Python stages are tested against an evaluator with its **own record reader**. It shares two small helpers with cm.py: the lexer that its formula evaluator uses inside a formula (TOKENS) and the variable collector (VARS). It shares no record reader, parser, rewriter or translator. Until round 4 it also shared, by copying, the regular expression that decided which records exist, so both silently skipped the same records; peer review found this (§6.2.2). Each test declares the tools it needs: without Vampire, the 82 tests that need no Vampire model still run (record reading, rejection, preprocessing,

nonemptiness, input integrity, name collisions, finder classification, ablation bookkeeping, rewrite equivalence on small structures and the kernel boundary cases); with Vampire 5.1 and Lean 4.35, all 170 run.

Test	What it checks
<code>holds(s, n, consts, tables, env)</code>	Not a test: a brute-force evaluator of the original TPTP text. It parses with its own recursive descent and evaluates <code>!/?</code> with <code>itertools.product</code> over <code>1 ... n</code> . It never sees <code>lean()</code> , <code>tokens()</code> or <code>Checker.lean</code> .
<code>records(text), raw(path)</code>	The oracle's reader of source records: a separate lexer (<code>ORACLE_LEX</code>) and a token-level matcher for <code>kind(name, role, formula[, annotations])</code> , which must consume every token of the file. <code>raw</code> returns each record's role and formula, with each <code>cnf</code> clause closed by <code>![VARS]</code> . It never calls <code>statements()</code> or <code>problem()</code> , so none of <code>cm.py</code> 's record reading or rewriting reaches the oracle.
<code>countermodel(path, ...)</code>	Python's verdict on <code>raw(path)</code> : every original formula holds under <code>holds</code> , except the conjecture, which fails. Both mutant tests compare Lean against this verdict.
<code>test_compiles_to_committed_file,</code> <code>test_reflects_to_committed_file</code>	<code>cm.py</code> output equals the committed <code>Cm/<name>.lean</code> (shallow) and <code>Refl/<name>.lean</code> (reflected), for each of the 15 examples/*.p.
<code>test_kernel_accepts_with_propext_only</code>	<code>lake build</code> succeeds and every line of <code>#print axioms</code> (via <code>Axioms.lean</code>) ends with <code>[propext]</code> .
<code>test_python_agrees_it_is_a_countermodel</code> <code>test_circle_of_fifths_is_the_known_answer</code>	<code>holds</code> confirms Vampire's model for every example. The model has exactly 12 elements, and <code>f</code> and <code>g</code> are $x \mapsto x+7$ and $x \mapsto x+5 \pmod{12}$.
<code>test_mutants_lean_and_python_agree</code>	Two random one-entry corruptions per example (seed 0): Lean (shallow) accepts exactly the mutants that <code>holds</code> accepts, and at least one mutant is rejected.
<code>test_reflected_mutants_agree_with_python</code>	Same, seed 1, through <code>emit_reflect</code> and <code>Checker.lean</code> . This tests the reflected translation, the trusted part of that path.
<code>test_reflected_certificate</code> <code>test_rewriting_preserves_meaning</code>	Each example's model certifies through <code>emit_reflect</code> . Each rewritten formula agrees with its raw original (a <code>cnf</code> closed with <code>![VARS]</code>) under <code>holds</code> , on the model and on every one-entry mutation of every table.
<code>test_a_theorem_has_no_counter_model</code> <code>test_large_problems_split_into_modules,</code> <code>test_constants_split_into_name_modules</code> <code>test_vampire_cache_is_transparent</code>	<code>cm.py</code> exits non-zero on $\forall X \ p(X) \vdash p(k)$. With <code>CM_CHUNK=2</code> or 1, modular output builds with <code>lake</code> ; <code>#print axioms</code> on the result contains no <code>sorryAx</code> . With <code>CM_CACHE</code> , the second run hits the cache (one entry) and emits identical Lean.
<code>test_boundary_cases_are_rejected_by_the_kernel</code>	Builds <code>tests/Boundary.lean</code> : the reviewer's out-of-range constant passes check but fails <code>wf</code> ; an empty domain and a leaking function fail <code>valid</code> ; an unbound variable and a wrong arity fail <code>wf</code> ; too little decoder fuel yields <code>⊥</code> (documented behaviour, not general truncation detection).
<code>test_quoted_and_plain_names_are_the_same_symbol</code>	<code>cat</code> and <code>'cat'</code> translate identically; <code>'X'</code> stays a constant, distinct from the variable <code>X</code> .
<code>test_unsupported_syntax_is_rejected_not_misread (7 cases),</code> <code>test_typed_statements_and_includes_are_rejected (2),</code> <code>test_a_symbol_at_two_arities_is_rejected</code> <code>test_benchmark_records_unsupported_input</code>	Fail-closed parsing: <code><~></code> , <code><=</code> , <code>~\ </code> , <code>~&</code> , <code>\$</code> distinct, numbers, strings, <code>tff</code> , <code>include</code> and mixed arities raise an error in both translators. The harness classifies such inputs as <i>unsupported input before</i> Vampire runs (the test supplies a Vampire binary that does not exist).

Test	What it checks
test_variables_cannot_capture_lean_names (5 cases)	Variables named T, True, False, Nat, Decidable compile; T and True failed on the translator before the fix.
test_include_expansion	bench.expand inlines a plain include (with or without TPTP's optional spaces); leaves includes in % and /* */ comments and inside quoted names untouched; leaves a selective include in place, which problem() then refuses; and raises on unresolved, nested and non-Axioms/includes.
test_block_comments_are_rejected	A /* ... */ comment, which cm.py does not strip, is refused rather than read as formulas; '/*' inside a quoted name is just a name.
test_shallow_certificate_proves_a_nonempty_domain	emit refuses a 0-element model, and every committed shallow certificate states $\text{Nonempty } T \wedge \dots$
test_benchmark_inputs_match_their_sources	Every benchmark input equals bench.expand applied to its TPTP original, byte for byte, and originals, expansions and axiom files match the SHA-256s in bench/inputs.json. Skipped if the inputs have not been rebuilt (bench/inputs.py).
test_every_record_is_accounted_for (7 cases), test_annotations_are_not_formulas	Hand-counted files: the reviewer's two contradictory files (2 records each), spaced and commented record syntax, annotations, integer and punctuated quoted names ('thm.bool.TRUTH', 'it\'s'), a % inside a name, and a comment hiding a fake record. statements, the oracle's records and problem must each find exactly the hand count.
test_unreadable_records_are_rejected_not_skipped (9 cases)	Stray text, a record without a formula, an unknown role, tff, include, a missing ., unbalanced parentheses, a /* */ comment and a two-word name: both readers refuse the whole file.
test_a_model_of_part_of_the_problem_is_rejected (2 cases)	The round-4 reviewer's protocol: for each contradictory file, a one-element model of only $\sim p(a)$. The oracle rejects it, and Lean rejects both the shallow and the reflected certificate.
test_source_symbols_cannot_collide_with_generated_names	Symbols named c0, p0, chunk0 and countermodel certify on both paths.
test_a_failing_finder_is_not_a_search_outcome, test_benchmark_reports_a_failing_finder_as_finder_error	A finder that prints no SZS status or termination reason and exits non-zero: cm.py reports <i>vampire failed</i> , the benchmark records <i>finder error</i> , and nothing is cached.
test_a_timeout_is_a_search_outcome_not_a_failure	Vampire 5.1's timeout output (Termination reason: Time limit, exit 1, no SZS status): the benchmark records <i>no model</i> , and nothing is cached.
test_rewriting_is_equivalent_on_every_small_structure (18 cases)	Adversarial forms for the rewrite (independent variables, a shadowed variable, $\forall$ moved across $\vee$, implication flattening, vacuous quantification, $\exists$ under $\forall$, equality, negated conjunction, an equivalence, transitivity clauses): original and rewritten formulas agree on every interpretation of constants a, b, unary p, q and binary r over domains of size 1 and 2 ($8 + 1,024$ structures each). Exhaustive within that finite space, not a proof.
test_clause_search_agrees_with_brute_force	clause_holds, an independent evaluator of cnf clauses that searches for a falsifying assignment, agrees with holds on six clauses over the same 1,032 structures (each clause is false on 192 to 578 of them).

Test	What it checks
<code>test_fidelity_check_flags_a_model_of_part_of_the_problem</code>	<code>bench/fidelity.py</code> reports NOT FAITHFUL for the round-4 reviewer's wrong model, because it reads the whole original file.
<code>test_explicit_finder_errors_are_finder_errors</code> (4 cases), <code>test_recorded_vampire_answers_classify_as_before</code>	Finders printing <code>SZS status Error</code> , an unrecognized termination reason (<code>Internal error, exit 3</code>), an error status with an ordinary reason, or a missing executable: the benchmark records <i>finder error</i> with the diagnostic and caches nothing. Every cached Vampire answer classifies as a definite search outcome, so the table changes no recorded row.
<code>test_formula_readers_reject_malformed_structure</code> (6 cases) <code>test_small_structure_check_detects_a_wrong_rewrite</code>	<code>(p), p(a), ![X] p(X), ![X,]: p(X), (p and p(a, : lean(), tokens() and the oracle's holds() each refuse them.</code> The negative control for the rewrite check: a wrong rewrite of the implication-flattening case (the variable replaced by a constant) disagrees with the original on exactly 156 of the 1,032 structures.
<code>test_ablation_keeps_every_experiment_distinct</code> , <code>test_ablation_refuses_records_it_would_otherwise_drop</code> (3 cases)	Round-4 final review. Records 'a-b', a_b, 'g-h' and g_h give six distinct experiments and six result rows (prover verdicts stubbed). A role outside the ablation language (negated_conjecture, lemma) or a duplicated record name is rejected before any experiment runs.
<code>test_a_cached_error_is_not_a_search_outcome</code>	Closing review: an error response seeded in the cache (as an earlier version could have stored) is ignored; Vampire runs again and its failure is a <i>finder error</i> .
<code>test_ablation_finds_what_a_conclusion_needs</code>	<code>ablate.py</code> labels a needed axiom with a certified counter-model, and never labels an underivable target as needing anything.

The suite has **170 test cases**: 5 tests parametrized over the 15 examples (75), 14 rejection and capture cases, 22 record-reading cases, 18 adversarial rewrite cases, 13 further parametrized cases (3 ablation-scope, 4 finder-error, 6 malformed-formula), and 28 single tests. In the repository, with the TPTP inputs rebuilt, all 170 pass (`bench/test-log.txt`, `pytest` exit status 0); in a tree restored from the review packages, two tests are skipped because their data is not included: the input-integrity test needs the TPTP corpus and the cache-classification test needs the Vampire cache, giving 168 passed and 2 skipped. Separately, `recheck.sh` rebuilds all 30 shipped certificates, prints their axioms and replays Checker, AllShallow and AllReflected through the kernel from scratch with `leanchecker --fresh` (Lean 4.35); its output is `bench/recheck.log`. The benchmark certificates are replayed the same way by `bench/replay.py` (§6.2).

4. The Verified Checker: Data Structures and Trust

`Checker.lean` (namespace CM, 430 lines) is the evaluator behind the reflected path (§3.6). It defines first-order syntax as data; a bounded semantics `Fm.Sem` and a Boolean evaluator `Fm.check`, proved equivalent by `Fm.check_iff`; decoders from packed natural numbers to formulas; ordinary first-order structures with satisfaction `Fm.Sat`; the validity and well-formedness checks `Model.valid` and `Fm.wf`; the bridge `sem_iff_sat`; and the certificate layer `Good` and counter-model-of-good (§4.4). A reflected certificate is one kernel evaluation of check per block, not one elaborated proposition per formula.

4.1 Data structures

Syntax is plain inductive data, with de Bruijn variables:

```
inductive Tm where
  | var (i : Nat)
  | const (c : Nat)
  | app (f : Nat) (args : List Tm)
```

```

inductive Fm where
  | atom (p : Nat) (args : List Tm)
  | eq (a b : Tm)
  | tru | fls
  | not (φ : Fm)
  | and (φ ψ : Fm) | or (φ ψ : Fm) | imp (φ ψ : Fm) | iff (φ ψ : Fm)
  | all (φ : Fm) | ex (φ : Fm)
deriving Repr

```

A model is a domain size and three lookup functions. Elements are $0 \dots n-1$:

```

structure Model where
  n : Nat
  const : Nat → Nat
  fn : Nat → List Nat → Nat
  pred : Nat → List Nat → Bool

```

Terms evaluate under an environment `env : List Nat`. Index 0 is the most recently bound variable:

```

def Tm.val (M : Model) (env : List Nat) : Tm → Nat
  | .var i => env.getD i 0
  | .const c => M.const c
  | .app f args => M.fn f (Tm.vals M env args)
def Tm.vals (M : Model) (env : List Nat) : List Tm → List Nat
  | [] => []
  | t :: ts => t.val M env :: Tm.vals M env ts

```

The semantics is a Prop. Quantifiers range over $x < M.n$:

```

def Fm.Sem (M : Model) : List Nat → Fm → Prop
  | env, .atom p args => M.pred p (Tm.vals M env args) = true
  | env, .eq a b => a.val M env == b.val M env
  ...
  | env, .all φ => ∀ x, x < M.n → φ.Sem M (x :: env)
  | env, .ex φ => ∃ x, x < M.n ∧ φ.Sem M (x :: env)

```

The evaluator mirrors it clause by clause in Bool. Bounded quantifiers become counting loops that stop at the first decisive value:

```

def allBelow (f : Nat → Bool) : Nat → Bool
  | 0 => true
  | k + 1 => f k && allBelow f k
def anyBelow (f : Nat → Bool) : Nat → Bool
  | 0 => false
  | k + 1 => f k || anyBelow f k

def Fm.check (M : Model) : List Nat → Fm → Bool
  | env, .atom p args => M.pred p (Tm.vals M env args)
  | env, .eq a b => a.val M env == b.val M env
  | env, .imp φ ψ => !φ.check M env || ψ.check M env
  ...
  | env, .all φ => allBelow (fun x => φ.check M (x :: env)) M.n
  | env, .ex φ => anyBelow (fun x => φ.check M (x :: env)) M.n

```

`allBelow` and `anyBelow` count down from `n` and build no list. Together with `&&/||`, they give check the pruning that §3.4's miniscoping relies on.

Structure	Role	Encoding	Lookup / evaluation cost
Tm	term syntax	inductive; var is a de Bruijn index	Tm.val: $O(t)$ model lookups; .var i is env.getD i 0, $O(i)$
Fm	formula syntax	inductive, 11 constructors	Fm.check: $O(\varphi \cdot n^2)$ worst case for quantifier nesting depth q; lazy &&, \ \ , allBelow, anyBelow prune cost of the function bodies (§4.2)
Model	finite structure	n plus const, fn, pred as functions	—
Fm.Sem allBelow / anyBelow	meaning (specification) bounded $\forall / \exists$	Prop, never evaluated structural recursion on n	$\leq n$ calls of f; stops at the first false / true
Holds M φ s	certificate statement	$\forall \varphi \in \varphi_s, \varphi.\text{Sem } M []$	—

4.2 Models in generated certificates

emit_reflect writes one `def M : Model`. Elements are shifted from Vampire’s 1 ... n to 0 ... n-1 ($x - 1$). Two helpers from `Checker.lean` do all lookups:

```
def get (packed i w : Nat) : Nat := (packed >>> (i * w)) % 2 ^ w -- w-bit field i
def index (n : Nat) : List Nat → Nat -- tuple → position, base n
| [] => 0
| x :: xs => x + n * index n xs
```

The generated model, with $n, K = 4096$, the element width $w = \max(n-1, 1).bit_length()$ and the mask width mw substituted:

```
def M : Model where
  n := n
  const c := get (if c / 4096 < k then ... else ...) (c % 4096) w
  fn f args := get (if f < k then ... else ...) (index n args) w
  pred p args := get (get MASKS p mw) (index n args) 1 == 1
```

Symbol kind	Python (emit_reflect)	Lean lookup	Cost in symbols
Predicate p	masks[i]: bit idx(a) set iff $p(a)$ is true; propositional symbols are 0-ary predicates with mask 1 or 0; all masks packed into one Nat at width mw	get (get MASKS p mw) (index n args) 1	$O(1)$: two gets, one index ($O(\text{arity})$)
Constant c	cvals packed w bits each, 4096 per block (cblocks)	bsearch('c / 4096', cblocks) selects the block; get ... (c % 4096) w	$O(\log b)$ comparisons for b blocks, then one get
Function f	fn_tables[i]: value $v-1$ at field idx(a), width w	bsearch('f', fn_tables); get ... (index n args) w	$O(\log F)$ comparisons for F functions, then one get

bsearch(var, xs) writes `xs[var]` as a balanced tree of `if var < mid` comparisons. A match on literals would be a chain of $O(n)$ comparisons. Each get is a shift of a packed Nat, and a shift costs time linear in the Nat’s bit length. Blocking bounds that length (§5, pattern 13a).

4.3 Problem encodings

(a) Token streams. A block of non-fact formulas is one Nat, $D < b$, of fixed-width tokens in prefix order. The token width is $tw = \max(\max \text{ token}, 10).bit_length()$, at least 4 bits.

Formulas	Opcode	Operands	Terms	Opcode	Operands
atom	0	$p\ k\ t_1 \dots t_k$	variable	0	i (de Bruijn)
=	1	$t\ t$	constant	1	c
$\top / \perp$	2 / 3	—	application	2	$f\ k\ t_1 \dots t_k$
$\neg$	4	φ			
$\wedge \vee \rightarrow \leftrightarrow$	5 6 7 8	$\varphi\ \psi$			
$\forall / \exists$	9 / 10	φ			

The decoder is mutually recursive and fueled:

Function	Signature	Reads
readTm s w	$\text{fuel} \rightarrow i \rightarrow \text{Option } (\text{Tm} \times \text{Nat})$	one term at position i ; returns the next position
readTms s w	$\text{fuel} \rightarrow k \rightarrow i \rightarrow \text{Option } (\text{List } \text{Tm} \times \text{Nat})$	k terms
readFm s w	$\text{fuel} \rightarrow i \rightarrow \text{Option } (\text{Fm} \times \text{Nat})$	one formula
readFms s w fuel	$k \rightarrow i \rightarrow \text{List Fm}$	k formulas; on none it returns $[\text{.fls}]$ and stops
decode s w k depth	List Fm	readFms s w depth $k\ 0$

Fuel bounds the recursion depth, and each call consumes one unit. Kernel cost grows with the fuel, so `emit_reflect` passes `depth = max(map(nesting, fs)) + 2`, where `nesting` is the token count of a single formula. That is an upper bound on the nesting depth, not the block's token count (§5, pattern 13b: $>9\text{ min} \rightarrow 12\text{ s}$). What the decoder does with bad data is precise, and narrower than “malformed data decodes to $\perp$ ”:

- **Detected failures:** if the fuel runs out, `readFm` returns `none`, and `readFms` emits `.fls`, which can never be certified.
- **Not detected by the decoder:** get reads zeros past the end of the data, and the formula count is supplied by the generator. So truncation, extra tokens and wrong counts can decode to *some* formula list. Such errors may be accepted as a different decoded formula list. The kernel proof remains about that list, not necessarily the intended source problem; detecting the mismatch is the responsibility of the trusted translation and its tests (§4.6).
- **Caught by Fm.wf:** out-of-scope variables, out-of-range constants and wrong function arities (§4.4).

tests/Boundary.lean records these cases as kernel-checked facts.

Illustration: `fof(irrefl, axiom, ![X]: ~ r(X,X)).` with `ids["pred"] ["r"] = 0`. tokens returns `[9] + body`, and `X` is index 0:

```

formula :  $\forall X. \neg r(X, X)$ 
tokens   : 9  4  0  0  2  0  0  0  0
           |  |  |  |  |  |  |  |
            $\forall$   $\neg$  atom  $p=0$   $k=2$  var 0 var 0
position: 0  1  2  3  4  5  6  7  8

```

packed (tw = 4 bits, token i at bits $4i \dots 4i+3$, least significant first):

```

bits  35..32 31..28 27..24 23..20 19..16 15..12 11..8 7..4 3..0
token  0      0      0      0      2      0      0      4      9
D = 9 + 4·24 + 2·216 = 131145

```

```

decode 131145 4 1 11 = [.all (.not (.atom 0 [.var 0, .var 0]))]
(depth = 9 tokens + 2 = 11)

```

(b) **Fact tables.** `emit_reflect` diverts every formula that `as_fact` recognizes as a ground literal $(\neg)p(c_1, \dots, c_k)$ over constants into fixed-width records. With K slots for the largest fact arity, each record has $K + 2$ fields of `fw` bits: polarity (1 asserted, 0 denied), predicate, then K constant ids, zero-padded. A block `F<j>` holds 128 records (`facts_per_block`). The arity table `ar` is packed at width `aw = K.bit_length()`.

```

def fact (s w K : Nat) (ar : Nat → Nat) (i : Nat) : Fm :=
  let b := i * (K + 2)
  let p := get s (b + 1) w
  let a := (List.range (ar p)).map fun j => Tm.const (get s (b + 2 + j) w)
  if get s b w == 1 then .atom p a else .not (.atom p a)

def decodeFacts (s w K : Nat) (ar : Nat → Nat) (k : Nat) : List Fm := (List.range k).map (fact s w K
  ↪ ar)

```

fact reads record i directly with $O(K)$ gets. No general decoding and no fuel are needed.

Illustration: the facts of `transitive_cnf.p` (§3.9), with $r = 0, a = 0, b = 1, K = 2$ and $fw = 1$:

record $i = \text{fields } [i \cdot 4 \dots i \cdot 4 + 3] \quad (K + 2 = 4 \text{ fields, } fw = 1 \text{ bit each})$

	polarity	predicate	slot 0	slot 1	
0: $r(a,b)$	1	0	0	1	$\text{as_fact} \rightarrow (1, 0, [0, 1])$
1: $\sim r(b,a)$	0	0	1	0	$\text{as_fact} \rightarrow (0, 0, [1, 0])$

$F0 = \text{pack}([1,0,0,1, 0,0,1,0], 1) = 2^0 + 2^3 + 2^6 = 73$
 $\text{ar} = \text{get } 2 \text{ p } 2 \quad (\text{arity of } r = 2, \text{ aw} = 2)$

theorem `hf0 : Holds M (decodeFacts F0 1 2 ar 2) := holds_facts M _ _ _ _ (by decide +kernel)`

4.4 Soundness

The certificate rests on two theorems, both proved once in `Checker.lean`.

(1) From bounded evaluation to ordinary semantics. Fm.Sem is a *bounded* semantics. Quantifiers range over $x < M.n$, but nothing in `Model` stops a constant or function value from leaving that range, and $M.n$ may be 0. Peer review rightly pointed out the consequence. With $M.n = 1$ and a constant interpreted as 1, the bounded semantics makes $\forall x, x \neq c$ true, although it is unsatisfiable in any structure containing c ; with $M.n = 0, \forall x, \perp$ is vacuously true. The checker therefore defines ordinary structures and satisfaction, and two kernel-checked conditions:

```

structure Structure (α : Type) where
  const : Nat → α
  fn : Nat → List α → α
  pred : Nat → List α → Prop
def Fm.Sat (S : Structure α) (d : α) : List α → Fm → Prop -- ∀/∃ range over all of α

def Model.valid (M : Model) (fns : List (Nat × Nat)) : Bool :=
  0 < M.n && fns.all fun (f, k) => allTuples M.n (fun xs => M.fn f xs < M.n) k
def Fm.wf (M : Model) (fns : List (Nat × Nat)) : Nat → Fm → Bool -- bound variables, constants in
  ↪ range,
  -- functions at checked arities

theorem sem_iff_sat (M) (fns) (hv : M.valid fns = true) (d : Fin M.n) (φ : Fm) :
  ∀ env : List (Fin M.n), φ.wf M fns env.length = true →
  (φ.Sem M (env.map Fin.val) ↔ φ.Sat (M.toStructure (Model.valid_pos hv)) d env)

```

$M.\text{toStructure}$ is the structure on $\text{Fin } M.n$ that M describes. Its values are clipped into the domain, and the proof shows the clipping never changes a value for valid models and well-formed terms (`val_interp`). For a closed formula ($\text{env} = []$), **the bounded semantics coincides with ordinary first-order satisfaction**. The reviewer's examples are rejected: `allNeC.wf Out [] 0 = false` and `EmptyDomain.valid [] = false` are kernel-checked in `tests/Boundary.lean`.

(2) The evaluator decides the bounded semantics:

One theorem connects the evaluator to the semantics. It is proved once, by induction on Fm , and holds for every formula and every model:

```
theorem Fm.check_iff (M : Model) (φ : Fm) : ∀ env, φ.check M env = true ↔ φ.Sem M env
```

What a certificate states. Each block is checked in one pass, $\phi.wf \ M \ fns \ 0 \ \&\& \ \phi.check \ M \ []$, giving $Good \ M \ fns \ block \ (good_of_check, \ good_facts)$. Blocks combine with $good_append$, $M.valid \ fns$ is one more $decide +kernel$, and $countermodel_of_good$ concludes:

```
theorem countermodel : ∀ φ ∈ problem, φ.Sat (M.toStructure (Model.valid_pos valid)) (0,
  < Model.valid_pos valid) [] :=
  countermodel_of_good M fns problem valid good
-- #check @transitive_cnf.countermodel (generated from examples/transitive_cnf.p):
--   ∀ (φ : Fm), φ ∈ transitive_cnf.problem → Fm.Sat (transitive_cnf.M.toStructure ...) (0, ...) [] φ
-- @Fm.Sat : {α : Type} → Structure α → α → List α → Fm → Prop   (dot notation places φ last)
```

That is: **the ordinary first-order structure on $\mathbf{Fin}$ satisfies every formula of the decoded problem.** What it does *not* state is that the decoded problem is the TPTP source; that is the trusted translation (§3.8).

The atom case is rfl . The propositional cases are $simp$ with the induction hypotheses, and imp adds a case split on $\phi.check \ M \ env$. The quantifier cases use

```
theorem allBelow_iff (f : Nat → Bool) : ∀ n, allBelow f n = true ↔ ∀ x, x < n → f x = true
theorem anyBelow_iff (f : Nat → Bool) : ∀ n, anyBelow f n = true ↔ ∃ x, x < n ∧ f x = true
```

each proved by recursion on n . Three lemmas lift $check_iff$ to certificates:

Lemma	Hypothesis (closed by $decide +kernel$)	Conclusion
$holds_of_check \ M \ \phi_s$	$\phi_s.all \ (\cdot.check \ M \ []) = true$	Holds $M \ \phi_s$
$holds_facts \ M \ s \ w \ K \ ar \ k$	$(List.range \ k).all \ (fun \ i \ => (fact \ s \ w \ K \ ar \ i).check \ M \ []) = true$	Holds $M \ (decodeFacts \ s \ w \ K \ ar \ k)$
$holds_append$	Holds $M \ a, Holds \ M \ b$	Holds $M \ (a \ ++ \ b)$

These truth-only lemmas are the inner layer of $good_of_check$, $good_facts$ and $good_append$, which add the well-formedness conjunct. In the generated certificate, $problem : List \ Fm$ is a balanced $++$ tree of the decode $D \dots$ and $decodeFacts \ F<j> \dots$ blocks, a conjecture appears negated (opcode 4), and $good : Good \ M \ fns \ problem$ is a balanced tree of $good_append$ over the per-block proofs h and $hf<j>$; the public statement is $countermodel$ above. NLP263+1's reflected certificate depends on the axioms $propext$ and $Quot.sound$ only. `#print axioms` on the library pins down where $Quot.sound$ enters: through $allBelow_iff$ and $anyBelow_iff$, the lemmas for the quantifier loops. $check_iff$, $holds_of_check$ and $holds_facts$ inherit it from them, and $holds_append$ needs only $propext$. Both are standard Lean axioms, used throughout Mathlib.

Fidelity, and its limit. $decode$, $decodeFacts$ and M are ordinary Lean definitions, so `#eval problem` prints the list of formulas the certificate is about. That is a useful diagnostic, not a proof of fidelity. It shows numeric symbol ids, so comparing it with the TPTP source needs the generator's symbol map and the rewrite. And `#eval` runs compiled code, so it adds nothing to the kernel proof. Fidelity to the source therefore rests on the trusted, tested translation (§3.8): a *verified checker* with a *tested translation pipeline*, not a verified TPTP-to-Lean compiler.

4.5 Shallow vs reflected certificates

	Shallow (emit, §3.5)	Reflected (emit_reflect)
Statement form	one Lean Prop per formula over inductive T ; theorem $c<i>' : \phi_i := by decide$	ordinary satisfaction $\phi.Sat (M.toStructure \dots)$ for every ϕ in problem, over Fm data
Trusted code	<code>lean()</code> , <code>sym()</code> , <code>emit()</code> , <code>rewriter</code> , <code>kernel</code>	<code>tokens()</code> , <code>as_fact()</code> , <code>symbol numbering</code> , <code>rewriter</code> , <code>kernel</code> . <code>Checker.lean</code> is proved, and <code>#eval problem</code> exposes the statement

	Shallow (emit, §3.5)	Reflected (emit_reflect)
Per-formula cost	parse + elaborate + typeclass search + decide for each formula	no per-formula elaboration: one Nat literal per block, evaluated by decide +kernel
Axioms	propext (examples)	propext, Quot.sound
Best use	quantifier-heavy problems	fact-heavy problems

Measured (8 GB laptop, 8 cores):

Workload	Shallow	Reflected
10,000-fact slice, Lean 4.35, median of 3 (§6.3.1)	31.1 s CPU	9.8 s CPU
NLP263+1, 1,026,857 facts, Lean 4.35, median of 3 (§6.3.1)	14.1 min wall	5.2 min wall
PLA025+1, quantifier-heavy, 10-element model, Lean 4.35	4.4 s	>400 s (did not finish); 228 s with the evaluator written using the primitive recursor

Reflection wins on ground facts and loses badly on nested quantifiers. *Hypothesis*: lazy kernel evaluation re-decodes subformulas inside the `allBelow/anyBelow` loops, because the `decode ...` term is unfolded under each quantifier instead of being shared. *Policy*: hybrid dispatch. `bench.py`'s `run_hybrid` reflects when `fact_share ≥ 0.9` and uses the shallow method otherwise. *Future work*: decode once and check the result. That means proving `decode D ... = L` for an explicit list `L`, then running `check` on `L`, so that quantifier loops never touch the packed stream.

4.6 Independent checks of the reflected path (`test_cm.py`)

The reflected path trusts the encoder, not the evaluator. Two tests target it, both against `holds`, a brute-force evaluator of the original TPTP text. It reads records with its own reader and shares only the in-formula lexer and the variable collector with `cm.py`: no record reader, parser, rewriter or translator (§3.10).

Test	Check
<code>test_reflected_certificate</code>	<code>lake build Checker</code> succeeds, and for every <code>examples/*.p</code> the <code>emit_reflect</code> output is accepted by <code>lake env lean</code> .
<code>test_reflected_mutants_agree_with_python</code>	Two random one-entry corruptions per example (seed 1): a predicate entry is flipped, or a function entry is changed to a different element. The list of reflected Lean verdicts must equal <code>[countermodel(p, *m) ...]</code> , where <code>countermodel</code> evaluates <code>raw(p)</code> , the formulas as written in the source, and never <code>problem(p)</code> , the production rewriter. Call chain: <code>countermodel(p, n, consts, tables) → raw(p) → records(text)</code> (the oracle's own lexer and record matcher over the whole file) <code>→ holds(f, n, consts, tables)</code> for each formula. Lean accepts exactly the mutants that Python accepts, and not <code>all(lean)</code> ensures that some mutant is rejected.

A wrong translation in `tokens` or `as_fact` would certify a formula other than the TPTP one. The mutant test exposes this whenever a corrupted model separates the two formulas, which is why its docstring calls it a check on “the part of the reflected path that is trusted”.

5. Scaling: Design Patterns, Data Structures and Measured Effects

Central lesson, for the workloads we measured. For our large ground-fact certificates, the cost was dominated by Lean *elaborating statements* (parsing, elaboration, typeclass search, kernel type checking of large terms) rather than by evaluating them, and reducing proposition elaboration gave the largest gains. This is not a claim about all Lean certificates: quantifier-heavy checking has a separate evaluation bottleneck, which our own reflected regression on PLA025+1 shows (§4.5).

Cost model. Unless stated otherwise, $O(\cdot)$ counts field operations (shifts, masks, comparisons) on fixed-width fields. On Nat data, a shift costs time proportional to the bit length of the number shifted. That is why blocking (pattern 13a) matters, and why an “ $O(1)$ packed lookup” means a constant number of operations on a bounded block, not constant time on an unbounded table.

Profile of one 2,000-fact shallow chunk (Lean 4.13):

Phase	Time	Share
Parsing	3.83 s	41%
Tactic execution	1.37 s	15%
Elaboration	1.34 s	14%
Kernel type checking	1.08 s	12%
Typeclass inference	1.04 s	11%
Other (definitions, linting, sharing, metavariables)	0.59 s	6%
Total	9.25 s	100%

Parsing alone outweighs tactic execution (where *decide* actually runs) by 2.8×. Together, parsing, elaboration and typeclass inference take 66% of the chunk’s time, against 15% for running the tactics.

Machine for all measurements: 8 GB laptop, 8 cores.

5.1 Master table

#	Pattern (classic name)	Where applied	Big-O before → after	Measured effect	Status
1	Backtracking with pruning (miniscoping; lazy $\forall$, bounded $\forall$ in <i>decide</i>)	clause evaluation	$ M ^k$ full enumeration → pruned search tree	NLP219-1: 13-var clause over 7 elements $\approx 10^{11}$ cases → 46,746 atom checks for the whole problem; impossible → certified	kept
2	Greedy join ordering (fewest new variables first)	literal order inside a clause	— (heuristic)	a heuristic, not a guarantee of picking the most selective literal (the density planner, R1, shows selectivity is data-dependent); on SWB017+3 it happens to start from a highly selective guard, <code>rest(S3, nil)</code>	kept

#	Pattern (classic name)	Where applied	Big-O before → after	Measured effect	Status
3	Loop fusion / deforestation (recursive flattening $\forall V(A \rightarrow \forall W(B \rightarrow C))$ → one clause, fresh vars only)	FOF → clause translation	nested loops re-enter outer guards → one fused loop	SWB017+3 368,112 → 5,932 atom checks; native_decide → kernel-only decide in 2–12 s. PLA025+1 91,921 → 31,675	kept
4	Bitset (bitmask predicate tables)	model predicate lookup	$O(n)$ list-membership scan → $O(1)$ bit test	NLP165-1 10.9 s → ~1 s	kept
5	Divide and conquer (one theorem per formula)	certificate layout	one monolithic term → n small independent terms	PLA025+1 47 s → 14 s; peak memory 805 MB → 420 MB	kept
6	Memoization (VARS cache of literal variables)	Python rewriter	recompute per visit → once per literal	SYN832-1 rewrite 21.8 s → 8.4 s, identical output	kept
7	Balanced binary tree	conjunction terms	depth $O(n)$ → $O(\log_2 n)$	linear 10,000-way $\wedge$ overflows kernel (“deep recursion”); tree passes	kept
8	Partitioning / sharding ($\leq 10,000$ declarations per module)	Lean modules	— (hard limit)	an observed, reproducible failure point on Lean 4.13 for our generated pattern: files with more than ~20,600 declarations fail (“deep recursion”); not re-tested on 4.35; Names split for 383,180 constants	kept
9	Segment tree / map-reduce (hierarchical aggregation)	final theorem	restate n formulas in Main → $O(n/\text{chunk})$ chunk conjunctions	10,000-formula slice 109 s → 12.3 s wall; Main 77 s → 0.09 s	kept

#	Pattern (classic name)	Where applied	Big-O before → after	Measured effect	Status
10	Content-addressed cache (SHA-256 of problem+flags+version)	Vampire calls	re-run per invocation → $O(1)$ hit	on a Lean-only rerun, every problem with a deterministic Vampire answer (77 models and 3 definite give-ups) is served from the cache; the 2026-10-15 rerun's median end-to-end time fell from 3 s to 0.65 s per problem; timeouts never cached	kept
11	Index lookup instead of rejection sampling	benchmark selection	~113 requests/problem → 1–2	picking ~165 s → negligible; end-to-end ~174 s → ~8 s per problem (~22×)	kept
12	Producer–consumer pipeline, memory-sized worker pool, fail-fast, LPT scheduling	build orchestration	— (makespan)	~600 MB per chunk → 3–4 workers on 8 GB; <code>native_decide</code> only after timeout; LPT ordering was not measured in isolation	kept
13	Reflection (verified evaluator <code>Checker.lean</code>) with compact encoding	ground-fact certificates	elaborate n propositions → elaborate one data literal + one evaluator	<i>Historical, truth-only certificate</i> (before the validity and well-formedness checks; single runs, Lean 4.35, §5.3): 10,000-fact slice 36.1 s → 5.1 s CPU. <i>Current, controlled</i> (§6.3.1, same Lean, validated certificate, median of 3): 31.1 s → 9.8 s CPU (3.2×)	kept (hybrid)

#	Pattern (classic name)	Where applied	Big-O before → after	Measured effect	Status
13a	Square-root decomposition / blocking (packed Nat per block)	reflected data encoding	lookup in one 557,591-bit number $O(N)$ per shift ($O(N^2)$ total) → $O(\text{block})$	unblocked packing quadratic; array literal of 11,312 numbers timed out in elaboration	kept
13b	Fuel = nesting depth instead of token count	reflected evaluator	kernel cost grows with fuel → bounded by depth	>9 min → 12 s	kept
13c	Direct addressing / binary search instead of match chains	reflected predicate and constant lookup	$O(n)$ literal comparison chain → $O(1)$ packed lookup, $O(\log n)$ blocks	10.4 → 8.5 s CPU	kept
13d	Fixed-width records (fact tables)	reflected facts	—	8.5 → 5.66 s CPU	kept
13e	Block-size tuning	reflected facts	—	128: 5.11 s, 512: 5.66 s, 2048: 7.68 s CPU	kept (128)
13f	decide +kernel instead of decide	reflected proof	skip elaborator-side evaluation	prototype 7.0 s → 3.1 s	kept
R1	Cost-based query planner (order by model table density)	literal order	—	PLA025 −24% atom checks; NLP219 +14%	rejected
R2	Hash-consing (memoized fact lemmas per equivalence class)	shallow facts	9,957 facts → 27 lemmas	build 38.8 s vs 12.3 s (3× slower)	rejected
R3	Opaque def vs abbrev for named propositions	shallow statements	—	no effect	rejected
R4	rfl instead of decide for positive facts	shallow facts	—	6.75 → 5.12 s (1.3×)	measured, not adopted: superseded by reflection
R5	Short-name interning	constant names	—	not measured	untested
R6	Reflection for quantifier-heavy problems	PLA025+1 (10-element model)	—	>400 s (did not finish) vs 4.4 s shallow; primitive recursor 228 s	rejected → hybrid dispatch

5.2 Patterns grouped by the cost they attack

Search size (patterns 1–3, R1). Search size grows as $|M|^k$ in the number of variables k . Backtracking with miniscoping, lazy $\forall$ and bounded $\forall$ cuts NLP219-1 from a $\approx 10^{11}$ -case clause to 46,746 atom checks for the whole problem. Greedy join

order starts each search at the literal that introduces the fewest new variables. That is a heuristic: it often coincides with a selective guard, but selectivity depends on the model’s data (R1). Recursive flattening fuses nested guarded quantifiers into one clause that introduces only fresh variables. Without it, the inner loop re-evaluates outer guards. This alone moved SWB017+3 from `native_decide` to `kernel-only decide`. Once search was this small, it stopped dominating the cost.

Elaboration (patterns 5, 9, 13, R2). These attack the cost that dominates in the profile. Splitting into one theorem per formula keeps each elaboration problem small: PLA025+1 47 s $\rightarrow$ 14 s, with peak memory halved. Hierarchical aggregation removes the second elaboration of every formula in Main. Restating 10,000 formulas in one final theorem took 77 s, of which the statement alone took 59 s and an explicit `And.intro` term 237 s. Reflection removes per-formula elaboration entirely. The certificate becomes data (packed Nat literals) plus one verified evaluator `Checker.lean`, run with `decide +kernel`. Hash-consing failed for the opposite reason: it *added* unification of each fact against a shared lemma (R2).

Kernel limits (patterns 7, 8, 13a–13c). These are hard limits rather than costs: Lean either works or fails.

- A right-nested $\wedge$ of 10,000 conjuncts fails with “deep recursion”. A balanced tree has depth $\lceil \log_2 n \rceil$.
- **On Lean 4.13, with our generated pattern and default options**, files with more than about 20,600 declarations failed with kernel “deep recursion”. Even 30,000 trivial `def c_i : Nat := 1` fail, in a band of ~340 failures around declaration 20,600. The fix caps modules at 10,000 declarations.
- In the reflected path, kernel evaluation is sensitive to representation.
 - A 557,591-bit number shifted on every lookup made decoding quadratic. Blocking fixed it.
 - Structural recursion on fuel = token count made evaluation take more than 9 min. Fuel = nesting depth: 12 s.
 - `match` on literals compiles to an $O(n)$ comparison chain. Packed $O(1)$ lookup and binary search over constant blocks replaced it.

Memory and I/O (patterns 4, 5, 6, 13d–13e). Bitmask tables replace list scans (NLP165-1 10.9 s $\rightarrow$ ~1 s). Each chunk needs ~600 MB, which caps parallelism on 8 GB. Divide and conquer halved peak memory on PLA025+1. Reflection shrank NLP263+1 from 178 MB of source to 25 MB and from 2.0 GB of build output to 65 MB (Table 5.3). Fixed-width fact records with 128 facts per block gave the lowest reflected CPU time.

Pipeline throughput (patterns 10–12). These patterns reduce the time spent outside Lean.

- **Vampire cache.** Keyed by SHA-256(problem, flags, version). Timeouts are never cached, because two problems flipped from model to timeout under CPU load. **Note:** the lower median time of reruns (0.65 s against 3 s per problem) reflects cache reuse, not a Lean improvement.
- **Benchmark selection.** It reads the TPTP ProblemAndSolutionStatistics index: 2,561 eligible problems of 26,990 (CNF SAT 1,305; FOF CSA 888; FOF SAT 368). Rejection sampling took ~113 requests per problem; the index needs 1–2.
- **Build pipeline.** A producer–consumer pipeline, with the worker pool sized by memory (3–4 workers). Jobs run longest first (LPT) to shorten the makespan. `native_decide` runs only after a `decide` timeout, so kernel-only proofs remain the default.

5.3 NLP263+1: shallow vs reflected

NLP263+1: 1,026,857 ground facts, 383,180 constants, 1-element model. Sources: `bench/nlp263–lean4.13.json`, `bench/nlp263–reflected–lean4.35.json`.

Historical, uncontrolled measurements. This subsection records the development-time runs: single runs, different Lean versions, different worker counts, varying load, and a reflected certificate *without* the validity and well-formedness checks added after peer review. They explain how the design was reached; they are **not** the paper’s performance claim. The controlled comparison is §6.3.1 (median 14.1 min shallow against 5.2 min reflected, 2.7 $\times$).

Metric	Shallow (Lean 4.13)	Reflected (Lean 4.35)	Ratio
Wall time	20.7 min (1,241 s)	4.8 min (290 s)	4.3 $\times$
CPU time	54 min (3,262 s)	17.5 min (1,052 s)	3.1 $\times$
Peak memory	1.64 GB	0.50 GB	3.3 $\times$
Lean source	178 MB	25 MB	7.1 $\times$

Metric	Shallow (Lean 4.13)	Reflected (Lean 4.35)	Ratio
Build output	2.0 GB	65 MB	~31×
Modules	555 (Base 1, Names 39, Chunk 514, Main 1)	253 (Base 1, Part 251, Main 1)	2.2×
Granularity	2,000 facts/chunk, 3 parallel	128 facts/block, 32 blocks/part, 4 parallel	—
Axioms	none	propext, Quot.sound	—
Failed modules	0	0	—

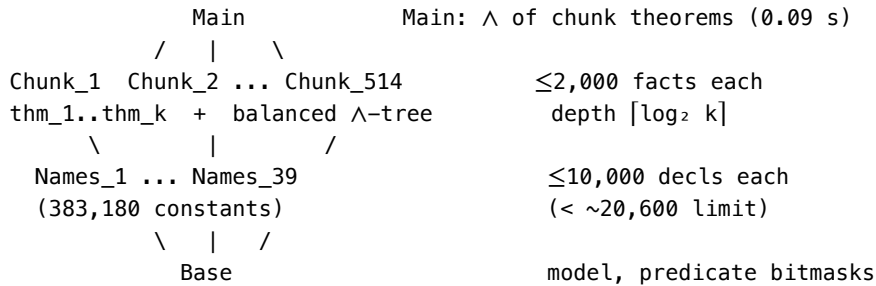
Lean versions differ between the two columns. On the 10,000-fact slice, however, the shallow method costs about the same on both versions (39.6 s CPU on 4.13, 36.1 s on 4.35). This suggested that most of the gap comes from the method, not the Lean version; §6.3.1 tests that directly on one Lean version and finds a smaller gap than this table shows. The version did matter for quantifier-heavy shallow proofs: PLA025+1 took 14–47 s on 4.13 and 4.4 s on 4.35.

10,000-fact slice, Lean 4.35:

Variant	Wall	CPU
Shallow	15.1 s	36.1 s
Reflected, one file	7.1 s	5.1 s
Bare reflection prototype	3.1 s	not recorded

5.4 Illustrations

Hierarchical aggregation (shallow path). Each formula is elaborated exactly once, in its chunk. Main only combines $O(n/2000)$ chunk theorems.



Reflected path: Base (model + Checker.lean) $\rightarrow$ Part_1 ... Part_251 (32 blocks each) $\rightarrow$ Main.

Blocking (reflected encoding).

unblocked: facts $\rightarrow$ one Nat of 557,591 bits
lookup(i) = (N >>> (i.w)) &&& mask shift touches all N bits
 $\Rightarrow O(N)$ per lookup, $O(N^2)$ per pass (quadratic)

blocked: facts $\rightarrow$ [B_0][B_1][B_2] ... [B_m] 128 facts per block
one packed Nat literal per block, fixed-width records
lookup(i) = decode(B_{i/128}, i mod 128)
 $\Rightarrow O(\text{block})$ per lookup; whole pass linear in N
constants: binary search over sorted blocks, $O(\log m)$

Block size is a trade-off between per-block overhead and per-lookup shift cost: 128 $\rightarrow$ 5.11 s, 512 $\rightarrow$ 5.66 s, 2048 $\rightarrow$ 7.68 s CPU.

5.5 Negative results

- **Cost-based query planner.** Ordering literals by model table density cut atom checks by 24% on PLA025 but raised them by 14% on NLP219. A planner that wins on one problem and loses on another adds complexity without a reliable gain. The greedy fewest-new-variables order was kept.

- **Hash-consed fact lemmas.** Grouping 9,957 facts into 27 shared lemmas by equivalence class made the build 3× slower (38.8 s vs 12.3 s). Unifying each fact with a shared lemma costs more than deciding the fact directly. This is the clearest confirmation of the central lesson: fewer terms do not help when each one needs more elaboration.
- **Opaque def vs abbrev** for named propositions had no measurable effect.
- **rfl instead of decide** for positive facts gave 1.3× (6.75 → 5.12 s). This is a small gain next to the order-of-magnitude effects above.
- **Short-name interning** was not measured.
- **Reflection on quantifier-heavy problems.** PLA025+1 (10-element model) did not finish in 400 s reflected, against 4.4 s shallow on Lean 4.35. The primitive recursor brought it to 228 s, still 50× slower. Hypothesis: lazy kernel evaluation re-decodes subformulas inside quantifier loops. Policy: hybrid dispatch, which reflects when $\geq 90\%$ of formulas are ground facts and uses the shallow method otherwise.

5.6 Lessons for builders of large Lean certificates

- **Profile before optimizing, and expect elaboration to dominate.** Parsing alone was 41% of a chunk. `decide` itself was a minority of the cost.
- **Never state a formula twice.** Prove each one once, locally, and aggregate proofs hierarchically. Restating 10,000 formulas in `Main` cost 77 s; aggregating chunk conjunctions cost 0.09 s.
- **Keep every term shallow and every file small.** Use balanced trees for big conjunctions ($\log_2 n$ depth) and keep modules at $\leq 10,000$ declarations. Lean 4.13 had an undocumented failure band near 20,600.
- **Reduce the search before you reduce the proof.** Miniscoping, lazy connectives, greedy join order and flattening turn 10^{11} cases into tens of thousands.
- **For ground data, use reflection, and design the encoding for the kernel.** Use packed blocks (~128 records), $O(1)$ lookups instead of match chains, fuel bounded by depth, and `decide + kernel`. For quantifier-heavy problems, stay shallow and dispatch per problem.
- **Distrust sharing that adds unification.** Hash-consed lemmas made the build 3× slower. Fewer terms help only if each one costs the same to elaborate.
- **Engineer the pipeline as carefully as the proof.**
 - Content-addressed caching. Never cache timeouts, because they depend on load.
 - Pick problems from an index instead of sampling.
 - Size worker pools by memory, not cores.
 - Schedule longest jobs first.
 - Fall back to `native_decide` only on a timeout.

6. Evaluation and Case Studies

All measurements were taken on one 8 GB, 8-core laptop. Wall-clock times vary by about 2× with system load. Atom counts and verdicts are exact. Lean versions are stated per result.

6.1 Case study A: a published Lean formalization (Spinoza’s *Ethics*)

Question. Nakamura [22] formalizes *Ethics* I in Lean 4 and proves two non-derivability results with hand-built counter-models:

- **S-A12:** A12 does not follow from `StatedAxioms` together with `PSRSubstance`;
- **S-A15:** A15 does not follow from `StatedAxioms` together with `plenitude`.

Can the tool find these counter-models by itself?

Method.

1. **Read** their `Definitions.lean` and `Axioms.lean` at commit 6d4c046.
2. **Transcribe** them into TPTP: primitives become predicates, defs become biconditionals, the 8 nontrivial fields of their `StatedAxioms` register become axioms (the register has 12 fields; 4 are `True` placeholders for A3–A6, and the 8 combine stated-axiom content with an exclusivity rider and their bridge axioms A8–A11, so they are not Spinoza’s seven axioms one-for-one), the weaker principle is an axiom and the target is the conjecture.
3. **Validate:** the transcription must reproduce their *positive* results first. Weak P5 follows from `StatedAxioms + PSRSubstance`, and A15 from `StatedAxioms + A25 + A26`.
4. **Search blind:** run Vampire on their axioms only, without consulting their `A12CounterModel`.

5. **Certify twice:** once on the transcription, and once as instances of their own `EthicaWorld` and `StatedAxioms` classes, checked against their unmodified library (nakamura/, pinned to Lean 4.13).

Results.

Claim	Nakamura’s model (hand-built)	Ours (automatic)	Checked against their library
S-A12	4 elements (A12CounterModel)	2 elements , 0.1 s	yes; axioms [propext]
S-A15	3 elements (A15CounterModel)	2 elements , 0.1 s	yes; axioms [propext]

The models.

- **S-A12:** two substances that *share* one attribute and *differ* in another. The sufficient-reason principle holds, and A12 fails. This is the shape of the classic objection to *Ethics* I P5.
- **S-A15:** two gods, one lacking an attribute of the other.

Caveats. In our models one object can be both a substance and an attribute. That is not an artifact of the encoding: in Nakamura’s reconstruction, A10 makes an attribute conceived through itself and A8 identifies that with being in itself, so any attribute *is* a substance. A sorted encoding that kept the two apart would therefore *conflict* with A8 and A10, not just refine them; it would need a revised signature or revised bridges. Also, their definition of God requires at least one attribute, not infinitely many (their source records the stronger cardinality content as a gap). Our smaller models are an automation result within this reconstruction, not evidence of a more faithful reading of Spinoza.

6.1.1 The assumption-dependency map `ablate.py` asks, for each target proposition, whether it follows from the full register, and then, for each axiom, whether it still follows without it. It reads the theory with the same total record reader as `cm.py`, keys every experiment by a collision-free identifier, and refuses a theory containing a role outside its language (definitions, axioms or hypotheses, conjectures) rather than drop that record. “Needed” means that removing the axiom yields a finite counter-model, **certified by Lean’s kernel**. “Not needed” means Vampire still proves the target; that rests on a Vampire proof, which is not certified here. The definitions of substance, attribute and God are held fixed throughout. The run covers 36 experiments, takes 105 s, and is recorded in `ablation/`.

Without ↓ / target →	weak P5	A12	A15
(<i>nothing removed</i>)	follows	does not follow (2-element model)	follows
any of the 8 stated axioms	not needed	still does not follow	not needed
A22 (sufficient reason for substances)	needed (2-element model)	still does not follow	not needed
A25 (plenitude)	not needed	still does not follow	needed (2-element model)
A26 (God is unique)	not needed	still does not follow	needed (2-element model)

Subset-minimal sufficient sets (greedy deletion, each step a Vampire proof; this finds *a* minimal set, not necessarily a smallest or the only one, and a Vampire timeout would leave minimality unresolved): weak P5 needs {A22} alone; A15 needs {A25, A26}. These are more than leave-one-out results, since leave-one-out alone cannot show that all individually unneeded axioms can be dropped *together*. They also have direct support upstream: Nakamura’s own Lean proofs derive weak P5 from `PSRSubstance` and A15 from the two fields of `PSRPlenitude`, neither using `StatedAxioms (ModalForm.lean at 6d4c046)`.

Findings:

1. **None of the eight nontrivial fields of Nakamura’s stated-axiom register is needed for these results.** Relative to the fixed definitions, the logical work is done by the added sufficient-reason principles. This is a precise form of the Bennett–Della Rocca question for this reconstruction.
2. **A12 is irreducible even against the whole register.** Nakamura shows A12 does not follow from `StatedAxioms` + A22. A certified 2-element model shows it also fails with plenitude (A25) and God’s uniqueness (A26) added.

3. The “needed” results confirm Nakamura’s positive Lean proofs (weak P5 from A22; A15 from A25 + A26) and show that each premise in them is necessary. A counter-model shows non-derivability, not full independence. Independence would also need the negation to be non-derivable, which we do not test. Both claims rest on the soundness of first-order logic, not on any internal Lean theorem about consistency.

Caveats:

- This is the first-order layer of the register only, not its S5 modal layer.
- “Not needed” rests on uncertified Vampire proofs.
- As in §6.1, the counter-models let one object be both a substance and an attribute, which the register permits.

6.2 Case study B: three samples of TPTP

Question. Does the tool work on problems nobody chose for it? When it fails, is it ever *wrong*?

Method.

1. **Index:** from TPTP’s ProblemAndSolutionStatistics, keep the 2,561 fof/cnf problems marked Satisfiable or CounterSatisfiable.
2. **Sample** 100 problems, never hand-picked, but **not a uniform sample overall**. The first 20 came from an earlier picker that chose a random *domain*, then a problem in it, which favors problems from small domains. The last 80 were drawn uniformly from the index. The set was also our **development set**: bugs found on it became fixes and regression tests, and it was rerun after each fix. So results on it measure performance on development-exposed problems. §6.2.1 reports two further uniform samples.
3. **Search:** 30 s of Vampire per problem. Results are cached by content, and timeouts are never cached.
4. **Certify** with hybrid dispatch: reflected when at least 90% of formulas are ground facts, shallow otherwise. **Budgets** (bench.py): a shallow certificate gets 60 s for its single file with decide; if that times out, a second 60 s attempt with native_decide, reported separately as native-assisted. A one-file reflected certificate gets 60 s. A modular reflected certificate gets 600 s for each module (Base, each Part, Main), with four Parts built in parallel and no cap on the total; NLP263+1 is the only problem that used it.

Problems were developed in Fibonacci batches (1, 1, 2, 3, 5, 8). Every bug found became a regression example, and after the last fix every problem was rerun with the final compiler.

Results (Lean 4.13, shallow; the baseline):

Outcome	Problems
Certified, strict kernel only	67
Certified with native_decide (also trusts Lean’s compiler)	3
Lean rejected (limits, not wrong models)	6
Vampire found no model within 30 s	23
cm.py error (NLP263, before the quoted-name fixes)	1

Inputs. Every benchmark input is rebuilt from the official TPTP v9.3.1 release (bench/inputs.py; tarball SHA-256 in bench/inputs.json), with includes expanded by the tested bench.expand (§3.8). The rebuild found a defect in the earlier scraped inputs: two of them, RNG025–9 (development) and B00008–3 (second sample), contain an include that the TPTP file comments out (% include('Axioms/...').), and the old expander had inlined it, adding the axiom file’s clauses to the problem (40 against 25 and 43 against 21 cnf lines). Both had been reported as *no model*. On their correct inputs Vampire finds models of 3 and 2 elements, and both are now certified. Two benchmark no-model-found outcomes concerned altered inputs and are superseded by the corrected runs; the defect produced no certificate.

Results. The first two columns are development history. The **final** column is the repaired implementation at commit 66c8b32: validated reflected certificates, fail-closed parsing, primed variables, Nonempty T in shallow statements, inputs from the TPTP release, and total record reading (§6.2.2). It was run from a separate git worktree with the same 30 s Vampire limit, and it is the result this paper reports. That rerun was not an idle-machine execution (§6.2.2).

Outcome	Lean 4.13 (shallow)	Lean 4.35 (hybrid, before review)	Final, 66c8b32
Models found by Vampire	77	77	78
Certified, strict kernel only	67	70	71
Certified with native_decide	3	2	2
Lean rejected	6	5	5
Vampire found no model	23	23	19
Unsupported input (rejected by the parser)	—	—	3
cm.py error	1	0	0
Method	shallow ×100	shallow ×98, reflected ×2	shallow ×95, reflected ×2, none ×3
Median end-to-end per problem	3 s	0.65 s (Vampire answers cached)	

Strict kernel certification: 71 of the 78 models Vampire found (91.0%). Two more were certified with `native_decide`, which enlarges the trusted base: 73 of 78 (93.6%) including those. Over all 100 problems, the full pipeline succeeds strictly on 71 (71%). These answer different questions, so all three are reported.

Development subset	Problems	Models found	Strict kernel	+ native	Rejected	No model
First 20 (domain-first picker)	20	17	17 (100% of models)	0	0	2 (+1 unsupported)
Last 80 (uniform from the index)	80	61	54 (88.5%)	2	5	17 (+2 unsupported)
All 100	100	78	71 (91.0%)	2	5	19 (+3 unsupported)

The uniformly drawn 80 certify 88.5% strictly, against 17 of 17 for the domain-first 20. That is a descriptive breakdown, not evidence that the picker caused the difference; the established problem is the non-uniform design itself.

Four verdicts changed between the versions:

- **NLP263+1:** a `cm.py` error on 4.13, certified on 4.35 through the reflective checker (§6.3);
- **HWV048-2:** a 16-element model that overflowed the stack on 4.13 now certifies in 2.7 s. This is consistent with Lean 4.33 bounding kernel recursion by `maxRecDepth`;
- **NLP018-10:** needed `native_decide` on 4.13 and certifies kernel-only on 4.35;
- **RNG025-9:** *no model* on the altered input, certified on the release input (above).

The failures, each explained. On Lean 4.35, **all 5 rejections are one family:** deeply nested modal-logic encodings (LCL647, LCL659, LCL673, LCL683, LCL685). They fail in Lean’s *typeclass instance search* for `Decidable`, within seconds, because the formulas are nested too deeply. The reflective checker needs no instance search, so it is the natural route for these (§8).

Of the 19 problems with no model in the final run, 10 are rated 1.00 in TPTP: no prover solves them in TPTP’s reference evaluations. *No model* means only that none was found within 30 s; it is not a semantic result. **Four earlier “no model” results turned out to be our own gap.** Vampire had answered them by deleting unused predicates without printing a model. The flag `-updr off` recovered three of them, and the fourth (NLP263) became case study C.

6.2.1 The second and third samples **Second sample: initially held out, then used for debugging.** We drew 100 problems uniformly without replacement from the 2,461 eligible problems not in the development set, with seed 20261015 (`bench/heldout-sample.json`), before any result on them was seen. We then used it: earlier runs on it exposed a crash

on include (1a55aef), two harness defects and a load-induced timeout (a23fe9c), and the variable-capture defect in GEG001+1 (62d43d6); the input rebuild then changed B00008-3. Each was fixed before the final run. These are general defects, not tuning to particular problems, but the sample’s results did inform the implementation, so **we report the second sample as a descriptive validation sample, not as an untouched test of generalization**. Superseded logs are kept in bench/superseded-*.jsonl.

Third sample: drawn and frozen before any result. We drew 100 further problems uniformly without replacement, seed 20261016, from the 2,357 eligible problems in neither earlier set nor among four exploratory problems used during development (bench/third-sample.json). The list was committed together with the frozen code (900818d) before any result on it existed. Two launches did not complete, and both are recorded:

1. At 900818d, the run stopped on its first development certificate because a fresh worktree lacked an output directory; acb343b creates it. No third-sample problem had been reached.
2. At acb343b, the development and second samples completed, and the third stopped when KRS290-1, whose formulas nest deeper than Python’s recursion limit, raised an exception the harness did not catch. No third-sample verdict was written. Before it stopped, the run had already run Vampire on the third-sample problems it reached, and may have compiled certificates for some; the Vampire answers stayed in the content-addressed cache, and we did not inspect them or any intermediate result. To locate the crash, we reran only the translator’s pre-check on the third sample; it showed that KRS290-1 was the only problem raising the exception (bench/superseded-acb343b-third-attempt1.txt).

The one change made after that, 30ad4f3, is in the harness: any translator exception is now recorded as a *cm.py error* instead of stopping the run. *cm.py*, *Checker.lean*, the dispatch rule, the time limits and the inputs did not change, so KRS290-1 is reported as a failure, not repaired. The third sample was then run in full at 30ad4f3. It is therefore not strictly untouched: one piece of information from it, that a problem crashes the pre-check, led to a harness change.

All three samples were then rerun after the round-4 repair (§6.2.2), which peer review prompted by auditing the code, not by any sample’s results. That repair changed what three third-sample inputs mean to the translator. **The reported results are therefore a corrected evaluation on samples that have all been exposed to development**, with the original sampling design retained. On the third sample, selected before its first execution, the solver and translator stayed frozen until the round-4 repair; a harness-only change after an aborted attempt recorded translator exceptions as failures. The completed evaluation strictly certified 59 of 67 found models.

Outcome	Development (100)	Second sample (100)	Third sample (100)
Unsupported input (rejected by the parser)	3	4	3
Translator crash before a model outcome (KRS290-1)	0	0	1
Vampire found no model within 30 s	19	34	29
Models found	78	62	67
Certified, strict kernel only	71	55	59
Certified with native_decide	2	2	3
Lean rejected (time or instance-search limits)	5	5	4
Model found, certificate generation crashed (SY0595-1) ¹	0	0	1
Strict, of models found	91.0%	88.7%	88.1%
Strict or native, of models found	93.6%	91.9%	92.5%
Full pipeline, strict, of all sampled problems	71%	55%	59%
Full pipeline, strict, of supported inputs	71 of 97 (73.2%)	55 of 96 (57.3%)	59 of 97 (60.8%)

All three samples at commit 66c8b32. The rows are mutually exclusive and each column sums to 100; models found = strict + native + Lean rejected + generation crash. ¹ SY0595-1 and KRS290-1 are the two *cm.py* error records of the third sample: one after a model was found, one before any model outcome. Both stay in the supported-input denominator.

Reading. On the third sample, the repaired implementation strictly certified 59 of the 67 models Vampire found (88.1%), against 91.0% on the development set and 88.7% on the second sample. The conditional rates are similar; three samples of this size do not establish that they are equal. Full-pipeline success differs more (71%, 55%, 59%), and that difference coincides mainly with how many problems Vampire found a model for within 30 s (78, 62, 67), not with certification. These are descriptive results; the second and third samples are uniform samples of the eligible problems, while the development set is not.

Failures, each listed.

- **Lean rejected.** Development: the five LCL instance-search failures above. Second sample: LCL649+1.015, LCL681+1.010 and GEG001+1 fail in instance search on deeply nested formulas; NLP055-1 and HWV076+1 exceed the 60 s Lean limit. Third sample: LCL663+1.010 and LCL649+1.020 fail in instance search; NLP213+1 exceeds 60 s; and ALG310-1, native-assisted in round 3 (54.8 s against the 60 s limit), exceeded the native limit in this rerun.
- **Native-assisted** (seven in all, reported separately from strict certificates). Development: ALG306-1, ALG334-1. Second: NLP179+1, SWB022+3. Third: NLP016+1, NLP186+1, NLP243+1.
- **Unsupported input.** Development: BI0001+1, HWV042+1, KRS244+1. Second: <~> in KRS182+1, KRS199+1, KRS210+1, and SYN000+2, which uses the role unknown and a selective include. Third: <~> in KRS224+1, KRS249+1, HWV105+1.
- **Translator crashes (third sample).** KRS290-1: formula nesting exceeds Python’s recursion limit in *cm.py*’s recursive-descent translator. SY0595-1: Vampire found a model, but the problem has more than 2,000 formulas, so *cm.py* emits a multi-module certificate, and the benchmark’s shallow path calls it without the output directory that requires; it crashes before writing the certificate. Both are limitations of the implementation as evaluated, reported rather than fixed (§8.2).

Why no model was found. Two kinds of *no model* outcome occur, and both are search outcomes. Most are timeouts at 30 s: Vampire 5.1 reports Termination reason: Time limit. The rest are fast give-ups (SZS status GaveUp, “Cannot represent all propositional literals internally”): the problem exceeds the finite-model builder’s internal encoding. Of the 82 across the three samples (19, 34 and 29), the TPTP headers say that 18 have no finite model at all (4, 8 and 6: “Has only infinite models”, “No finite models”, or Infix’s verdict), so no finite-model finder can succeed on them. GRP773-1’s smallest model has 64 elements. Nine are encoding give-ups (PLA041-1, SYN832-1; SYO593-1, SYN817-1, KRS281-1; SYN821-1, SYO603+1, SYN864-1, HWV070+1). The other 54 timed out without a documented cause; 28 of them are rated 1.00 by TPTP, meaning no prover solved them in TPTP’s reference runs. *No model* means only that none was found within the limit; it is not a semantic result.

6.2.2 Round-4 repair: record omission, and what it changed **The defect.** Peer review audited the delivered code and showed that *cm.py* found records with a regular expression that *searched* the file. A record it could not match was skipped without error: a quoted name with punctuation, or a comment or whitespace between *fof* and *(*. The test oracle had copied that expression, so it agreed with the translator on the weakened problem. With two contradictory two-record files and a deliberately wrong model of only the surviving record, both certificate paths accepted it. Records are now read by the total, fail-closed statements (§3.3), the oracle has its own reader (§3.10), and those two files are regressions that both paths must reject.

Its effect on the benchmark (*bench/compare_parsers.py*, *bench/parser-comparison.json*): - **297 of the 316 inputs read identically.** These are the 300 benchmark problems, the 15 examples and the ablation theory; the 297 include NLP263+1, compared as an exact list of 1,026,857 records. - **10 are rejected by both versions.** - **9 changed.** Each has quoted record names with punctuation ('EQ-var', 'thm.bool.TRUTH'): COM150+1 (28 → 60 records) and eight SWW problems (5 → 10 to 44). COM150+1 had been *no model*, which stands, since Vampire read the full file. **Eight earlier certificates omitted source premises and were withdrawn:** SWW883+1 (development), SWW882+1, SWW902+1, SWW938+1, SWW942+1 (second), and SWW865+1, SWW889+1, SWW921+1 (third). **The repaired pipeline subsequently certified their complete source problems,** each with a 2- or 3-element model. Unchanged strict totals do not validate the superseded certificates. One previously native-assisted result, ALG310-1, timed out in the rerun and is reported as a failure.

Running the rerun. The first rerun, at 8d46e97, required an SZS status from Vampire. Vampire 5.1 reports a timeout without one, so that run labelled its 17 development timeouts *finder error*. We stopped it after the development sample and kept the partial ledger (*bench/superseded-8d46e97-dev-timeout-misclassified.jsonl*). 66c8b32 also accepts Vampire’s termination reason, and all three samples were rerun there. A row-by-row comparison (*bench/timeout-*

correction-comparison.json) finds exactly those 17 rows different, all *finder error* → *no model*; the other 83, including all 71 strict certificates, are identical, and the models-found denominator is 78 in both. All three samples were rerun with a background process occupying one CPU core: an orphaned worker of our own parse comparison, found and stopped afterwards. Strict certification totals were unchanged. ALG310-1 exceeded the native-assisted time limit; background contention may have contributed, but causation was not established. The timeout is retained in the reported outcomes, and this load qualification does not apply to the controlled timings of §6.3.1.

Source-fidelity coverage of every certified result (bench/coverage.py, bench/coverage.md). Each check is reported separately, and *unchecked* means neither verified nor failed.

For every one of the 192 certified results, the test oracle reads the **original** problem file with its own record reader and evaluates every original formula, by brute force, in the model Lean certified (bench/fidelity.py, cap 300 s per problem).

	Strict (185)	Native-assisted (7)
Original-formula satisfaction	184 faithful , 1 unchecked	5 faithful, 2 unchecked
Rewrite agreement (below)	176 agree, 9 unchecked	4 agree, 3 unchecked
Compilation and fresh kernel replay	185 passed	not replayed (see below)
Failed, in any check	0	0

Three results are **unchecked** for original-formula satisfaction, because brute-force evaluation of their original formulas did not finish in 300 s: - NLP219-1 (development, strict); - NLP179+1 (second sample, native-assisted); - NLP186+1 (third sample, native-assisted).

For these three, two supplementary checks are reported separately and do **not** establish that the original formulas hold. First, record identity: cm.statements and the oracle's reader consume the whole file and agree record by record on kind, name, role and formula tokens (124, 67 and 67 records). Second, satisfaction of the rewritten formulas: the oracle confirmed it for NLP179+1, while it was unchecked at 300 s for NLP219-1 and NLP186+1. An independent clause-search evaluator, validated against the brute-force one on every small structure, also did not finish on NLP219-1 within 10 minutes. **Original-formula satisfaction of these three results therefore remains unchecked, so the source-fidelity claim covers 189 of the 192 certified results.** bench/coverage.md lists every exception with all its checks.

Bounded validation of the rewrite. - Adversarial forms. On 18 hand-written formulas (15 fof, 3 cnf) covering independent variables, a shadowed variable name, $\forall$ moved across $\vee$, implication flattening, vacuous quantification, $\exists$ under $\forall$, equality, negated conjunction, an equivalence that must not be flattened, and transitivity clauses, the original and rewritten formulas agree on every interpretation of the signature (constants a, b; unary p, q; binary r) over domains of size 1 and 2. That is $8 + 1,024 = 1,032$ structures per formula, 18,576 comparisons in all (test_rewriting_is_equivalent_on_every_small_structure). A deliberately wrong rewrite (a variable replaced by a constant) disagrees on exactly 156 of the 1,032 (test_small_structure_check_detects_a_wrong_rewrite). - **Benchmark.** bench/rewrite_check.py pairs every source record (read by the oracle) with its rewritten formula, for every problem whose outcome is certified, native-assisted or Lean-rejected (206 problems; this selector leaves out SY0595-1, whose certificate generation crashed after a model was found, and so no certified result), and compares each rewritten formula with its original on the model, up to 50 one-entry mutations and 10 random structures of the same signature (domains of 1 to 3 elements). Of those 206, **192 were checked in full**: 7,165 rewritten formulas (out of 1,036,470 formulas in those problems), 364,181 original-against-rewritten evaluations (7,165 on the models, 285,366 on mutations, 71,650 on random structures), **0 disagreements**. **14 are unchecked** at the 60 s cap per problem, because brute-force evaluation of their original formulas is too slow: 9 with strict certificates (NLP184-1, NLP219-1, SWB017+3, SWB023+3 in development; NLP192-1, NLP199-1, SWB007+3 in the second sample; NLP238-1, SWB025+3 in the third), 3 native-assisted (ALG306-1, ALG334-1, SWB022+3) and 2 Lean-rejected (NLP055-1, ALG310-1) (bench/rewrite-check.jsonl).

This is exhaustive within the stated finite test space and bounded validation on the benchmark, not a proof that the rewrite preserves meaning.

Code after the evaluation run. The benchmark results are those of commit 66c8b32. The final round-4 review then found three further defects, fixed in the released code: - **Vampire answer classification:** answers outside a documented table (an SZS Error, an unrecognized termination reason, a missing executable) were reported as *no model*. They are now *finder errors*. - **Malformed brackets and empty fields** were accepted. They are now rejected by both record readers and by the formula translators. - **abl**ate.py could merge experiments for distinct record names that sanitize alike, and silently dropped records with roles outside its language. It now uses collision-free identifiers and rejects such records.

None of these changes what valid input means: - **Reading:** against 66c8b32, every one of the 316 inputs reads identically or is rejected by both versions (306 and 10; bench/parser-comparison-r4b.json). - **Certificates:** the 30 shipped certificates are byte-identical. - **Recorded answers:** every recorded Vampire answer classifies as before (test_recorded_vampire_answers_classify_as_before). - **Ablation:** the Nakamura map reruns unchanged, with all 36 verdicts and sizes and the greedy sets identical (evidence/round4b/ablation-rerun-report.json).

So the evaluation was not rerun.

Fresh replay of the strict benchmark certificates. bench/replay.py recompiled every strict certificate of the rerun as a Lean module, printed #print axioms for each countermodel, and replayed one aggregate module per sample from scratch with leanchecker --fresh, imports included (Lean 4.35, same laptop). **All 185 compiled and passed:** development 71 in 910 s, including the million-fact NLP263+1 certificate; second sample 55 in 47 s; third sample 59 in 55 s (bench/replay.log, bench/replay-{dev,second,third}.jsonl; replay.py exited 0). Axioms: propext for 182, propext and Quot.sound for the two reflected certificates (NLP263+1 and AGT042+1), and none for SYN092-1.003. The seven native-assisted certificates are not replayed: replay cannot validate their compiled-code axioms, so they stay a separate, weaker category. leanchecker --fresh replays through Lean’s own kernel; it is not an independent kernel implementation. A successful replay shows that the kernel accepts the emitted statements; it says nothing about whether those statements are faithful to the source, which the fidelity check above addresses.

6.3 Case study C: one million facts (NLP263+1)

NLP263+1 is a natural-language knowledge base: **1,026,857 ground facts over 30 predicates and 383,180 constants**, 70 MB of TPTP. Vampire’s model has a single element: every constant denotes it. Checking each fact is therefore trivial, and the problem is pure scale.

The first measurements (Lean 4.13 shallow against Lean 4.35 reflected, single runs, under varying load) suggested about 5× in wall time. Peer review rightly called that comparison uncontrolled, and the reflected certificate has since gained validity and well-formedness checks. §6.3.1 replaces it; the old figures remain only in §5.3, labelled as history.

6.3.1 Controlled comparison bench/perf.py at commit 62d43d6. Lean 4.35.0; Vampire 5.1.0 answers from the cache; Apple M2, 8 cores, 8 GB, macOS 14.4. Each run is a clean build of the generated certificate with **4 parallel Lean workers for both methods**; the checker library is prebuilt and not timed. Shallow and reflected runs alternate, three repetitions each, on an otherwise idle machine (a macOS update service was active in the background). The reflected timings include the validity and well-formedness checks of §4.4.

Version. The timings were taken at 62d43d6. Regenerated with the final evaluation code (cm.py as at 66c8b32) from the same cached Vampire model, **all 253 reflected modules of NLP263+1 are byte-identical** to the timed ones. Of the 555 shallow modules, 40 are byte-identical and 515 differ. They differ only by the round-4 renaming of the emitter’s helper declarations (c_i, P_i, C_j, chunk_j gain a prime) and, in Main.lean, the Nonempty T conjunct (§3.5). Applying exactly that renaming to the timed output makes all 555 byte-identical (bench/perf-equivalence.json). The reflected timing therefore measured exactly the certificate the final code produces; the shallow timing measured a certificate that differs by those changes, which were not re-timed, and whose cost we have not measured. The rebuilt input differs from the timed one only by trailing blank lines.

NLP263+1 (1,026,857 facts)	Shallow: median (range)	Reflected: median (range)	Ratio of medians
Wall time	14.1 min (12.6–17.2)	5.2 min (5.2–5.8)	2.7×
CPU time	52.5 min (52.3–55.3)	20.3 min (20.2–21.0)	2.6×
Peak memory, largest single process	1.42 GB (1.28–1.48)	0.50 GB (0.49–0.51)	2.8×
Sampled maximum memory, all concurrent Lean processes	2.67 GB (2.67–2.69)	1.86 GB (1.85–1.87)	1.4×
Axioms	none	propext, Quot.sound	

10,000-fact slice of NLP263+1	Shallow: median (range)	Reflected: median (range)	Ratio of medians
Wall time	10.9 s (9.8–11.4)	4.4 s (4.1–4.7)	2.5×
CPU time	31.1 s (31.0–32.0)	9.8 s (9.5–10.0)	3.2×
Peak memory, largest single process	0.90 GB (0.82–0.96)	0.46 GB (0.46–0.46)	2.0×
Sampled maximum memory, all concurrent Lean processes	2.02 GB (1.94–2.12)	1.33 GB (1.32–1.34)	1.5×

What this supports. Under matched conditions, reflection certifies this ground-fact workload about **2.5–3×** faster. Its largest single process uses about 2.8× less memory on the full problem (1.42/0.50 GB) and about 2.0× less on the slice (0.90/0.46 GB); summed over concurrent Lean processes, the sampled maximum is only about 1.4–1.5× lower. The speedup is roughly half what the uncontrolled comparison suggested. CPU time varies by under 6% across repetitions, while shallow wall time varies by 37% (12.6–17.2 min). Memory pressure on an 8 GB machine and the background update service are plausible explanations for that spread; we did not record the system measurements that would establish either. Both problems have one-element models, so these are **throughput stress tests**, not evidence about larger domains or quantifier depth, where reflection is slower (§4.5, §8).

Definitions (also stored in each record of `bench/perf.jsonl`). Wall time runs from invoking `cm.py` to the end of the final `#print axioms` check; model search is excluded (cached). CPU time is user plus system time over all child processes. Memory is reported two ways: the largest single process’s maximum resident set size (`/usr/bin/time -l`, an exact peak per process), and the largest sum of resident set sizes of concurrently running Lean processes, sampled every 0.2 s (a sampled maximum, which can miss a shorter peak). A fresh kernel replay (`leanchecker --fresh`) is timed separately (§3.10), not included.

How this problem forced the scaling recipe. It crashed `cm.py`, because of % inside a quoted gloss. It then crashed Lean’s kernel twice, on a 10,000-long conjunction and on a file with more than about 20,600 declarations. And it exposed a variable-capture hazard, through a constant named `'T'`. Each failure became one of the patterns in §5.

6.4 Summary

	Evidence
Generality	Strict certification of 71 of 78 models (development), 55 of 62 (second sample) and 59 of 67 (third sample), at one commit after the round-4 repair; 2, 2 and 3 more native-assisted, reported separately. All three samples have been exposed to development, so these are descriptive results. Uncertified models produced no certificate; two third-sample inputs crashed the translator (§6.2.1)
Scale	1,026,857 facts certified in a median 5.2 min reflected, against 14.1 min shallow, under matched conditions (§6.3.1)
Usefulness on real formal work	Automatic, smaller counter-models for a published Lean formalization

	Evidence
Trust	Strict certificates use kernel checks and standard axioms only (7 more native-assisted across the three samples, reported separately); reflected certificates prove ordinary first-order satisfaction (§4.4); four real bugs in trusted layers caught by independent checks, and three more found during the review: silent misreading of unsupported syntax; variable capture, which failed closed; and include expansion inside comments, which changed two inputs and made two benchmark no-model-found outcomes concern altered inputs (superseded by the corrected runs). An artifact audit in round 4 found that the record reader could skip a legal record: eight certificates had omitted source premises and were withdrawn; the repaired pipeline certified their complete source problems, and every certified result is now checked against its original source file (§6.2.2)

7. Related Work

This work sits where four lines of research meet: counter-model finders attached to proof assistants, automated provers connected to Lean, proof by reflection with verified checkers, and very large machine-generated Lean certificates. Each is mature on its own. Below we say what each line does, where it differs from `lean-countermodel`, and what we could and could not verify. Every claim about another work was checked against the source listed in [References](#) on 2026-10-15. Where a source did not settle a point, we say so.

7.1 Counter-model finding in proof assistants

Isabelle: Nitpick and Nunchaku. Nitpick [3] is a counterexample generator for Isabelle/HOL. It translates higher-order goals into relational logic and solves them with Kodkod [4], a SAT-based relational model finder. Nitpick covers far more than we do: higher-order logic, (co)inductive datatypes and (co)recursive functions. It does not produce a certificate for Isabelle’s kernel, however. A model it reports as genuine is trusted on the strength of the translation and of Kodkod. Nunchaku was announced as a successor to Nitpick that several proof assistants could use [5, 6]. Its repository still describes it as “in initial development phase, not ready for use” [6], and we found no Lean front end for it.

HOL4: HolRefute. HolRefute [2] is an open pull request against HOL4, opened on 2026-09-04 and **not merged** when we checked on 2026-10-15. It adds Quickcheck, narrowing and a Kodkod model finder that it describes as a “module-for-module port of Nitpick (Isabelle2025-2)”. It also replays counterexamples: “every counterexample is replayed and, where replay succeeds, turned into a HOL theorem”. Replay fails closed. Its trust policy still accepts a `Genuine` verdict that has no certificate, as Nitpick does with Kodkod, and the commit history shows this policy was argued over before being kept [2]. So HolRefute is the closest analogue in another system: an untrusted finder whose output is, where possible, re-proved by the kernel. It differs from our work in three ways. It targets HOL4. It replays by kernel evaluation (`computeLib`, `EVAL`) and by decision procedures rather than through a verified first-order evaluator. And certification is best-effort, whereas we report a model only when Lean has certified it.

Equational Theories Project (ETP). The ETP [1] settled all 22,028,942 implications between 4,694 equational laws of magmas and formalized the results in Lean. Its finite-magma counterexamples came from brute force, SAT solvers (Glucose, Kissat), Mace4 [7] and Vampire [24], among other methods [1, §5]. A custom `Facts` command packs one finite magma that refutes many implications into a single theorem. The theorems are proved with a decision tactic (`decideFin!` in the paper’s Figure 6), and “tactics such as `native_decide` that relied on external tools were not permitted into the codebase” [1, §3]. The formalized generating set has 10,657 positive and 586,925 negative implications, most of the negative ones held in `Facts` theorems [1, §3]. The ETP is the largest precedent we know of for kernel-checked finite counter-models in Lean. Its machinery is specific to one signature: one binary operation, with equations indexed by number. It is not a general first-order tool. The project also reconstructed Vampire *proofs* in Lean [1, §7, Fig. 10].

Lean: Lean Disprove and plausible. Lean Disprove (Jan Ondras and Cameron Freer, 3rd AI for Math Workshop at ICML 2026) [10] is a `/disprove` command for coding agents in the `lean4-skills` package [11]. It runs a bounded

six-phase search with a registry of methods: “decision procedures, finite enumeration, random sampling, and external scripts such as SAT/SMT queries”. It reports REFUTED only “when Lean accepts the refutation” [10]. We share its rule that external witnesses are never trusted. As described, its search works over the types a Lean goal already mentions. Its method registry does not include a first-order model finder that invents a new carrier and interpretations for uninterpreted symbols. `plausible` [12] is property-based testing for Lean 4. It samples and shrinks counterexamples over types with `SampleableExt` or `Arbitrary` instances. It reports counterexamples and does not produce a proof of the negation [12].

Model finders. Mace4 [7] searches for finite models of first-order formulas by grounding and equational rewriting. Its manual describes it as complementary to theorem provers, as a searcher for countermodels. Vampire’s finite-model builder [24], which we use, plays the same role. Neither finder’s output is checked by a proof assistant unless a tool like ours does it.

Finite models turned into proof-assistant certificates in Coq. The closest predecessor of this work’s core idea is Goubault-Larrecq’s *Finite Models for Formal Security Proofs* [32]. It extracts finite models of first-order clause sets that represent security protocols, model-checks them, and instruments the checker to output a formally checkable proof, for example in Coq. It differs from this work in input (protocol clause sets), prover integration and target system, and it does not use reflection in Lean or work at the scale reported here. But the finite-model-to-certificate idea precedes us. Separately, Kirst and Larchey-Wendling mechanize first-order finite model theory constructively in Coq [33]: semantic background rather than an external-certificate tool, but it further limits any novelty claim about the checker concept itself.

7.2 Prover integration in Lean

Lean can already import proofs. Duper [18] is a proof-producing superposition prover written in Lean. Lean-auto [16] translates Lean goals for external automated provers. LeanHammer [17] combines a neural premise selector with these components into what its authors call “the first end-to-end domain general hammer for the Lean proof assistant”. “Lean on Vampire Proofs” (Bodingbauer, Hajdu, Kovács, Polaczek and Rawson) [8] reconstructs Vampire proofs as trusted Lean proofs. A companion library, `vprover/vampllean` [9], is “required by Lean for checking generated proof files”. The paper appeared at ITP 2026 (LIPIcs 382, article 36), and its publisher record lists `vprover/vampllean` and Vampire’s `leancheck` branch as supplementary software [8].

All of this work moves **proofs** from a prover into Lean, so it certifies *validity*. We move **models**, which certify *non-validity* or consistency. The two directions complement each other. Run on the same TPTP problem, Vampire either finds a refutation, which [8] can replay, or a finite model, which we can certify. To our knowledge, none of [8, 9, 16, 17, 18] imports models.

7.3 Certificates by reflection

Proof by reflection has a standard shape. Write a checker as a Lean function, prove it sound once, and let evaluation carry each instance. Recent Lean work applies this shape to *proof* certificates.

- **PBLean** [13] checks VeriPB pseudo-Boolean proofs with a Boolean checker whose soundness is “fully proved in Lean and executed as compiled native code”, that is through `native_decide`. This puts Lean’s compiler, and the axiom `Lean.ofReduceBool`, in the trusted base.
- **Streaming LRAT** [14] turns a SAT solver’s LRAT certificate into a Lean theorem. It streams the certificate through Lean core’s verified LRAT checker, made resumable, and reports importing 174 TB of certificates in a scaling experiment. The abstract does not say whether the final check is kernel-only or compiled. We did not establish which.
- **bv_decide** (Lean 4.12) [15] reduces `BitVec/Bool` goals to SAT. CaDiCaL refutes the instance, and “the resulting LRAT proof is checked in Lean”. The release notes state that its proofs “use `Lean.ofReduceBool`”, so the compiler is trusted.

Our reflected checker (`Checker.lean`, §4) follows the same pattern for a different object. It verifies that a **finite structure satisfies a first-order formula**, under Tarski semantics, rather than that a derivation is correct. It also makes a different trust choice: certificates are checked with `decide +kernel` and use only `propext` and `Quot.sound`. Neither `native_decide` nor `Lean.ofReduceBool` is used. That rule matches the ETP’s ban on `native_decide` [1]. Measured costs of the choice are in §5–§6.

7.4 Large finite certificates in Lean

Two papers by Florath show the proof-carrying-data pattern in Lean 4 for covering codes. In [19], a database stores bounds together with traces that “replay to Lean proofs”. In [20], $K_8(4, 2) = 23$ is proved from an explicit 23-word code and two LRAT refutations, with “no external SAT solver used during proof replay”. The ETP [1] remains the largest example of finite counter-models in Lean: about 587,000 negative implications, packed into Facts theorems because “explicitly formalizing all 22 028 942 implications as theorems would lead to an infeasible compilation time in Lean” [1, §3]. That observation agrees with our central measurement in §5: for large generated Lean files, elaboration dominates the cost. None of [1, 19, 20] gives a per-phase profile or a declarations-per-file limit, which is what we contribute there.

Two further items bear on scaling. Lean 4.33 [27] “bounds kernel type checking by `maxRecDepth` rather than the physical stack”, making “(kernel) deep recursion detected deterministic across platforms and builds”. This is why our Lean 4.13 limits may not carry over to later Lean versions (§8). Our search reductions reuse classic first-order preprocessing, in particular miniscoping, which pushes quantifiers inward as in standard clause-normal-form computation [26]. We claim no novelty for these patterns, only for measuring them on Lean certificates.

7.5 Formalized philosophy

Nakamura [22] formalizes Part I of Spinoza’s *Ethics* in Lean 4 and gives two counter-models. The abstract names “a four-element counter-model” for Proposition V and a second one for axiom A15. The abstract also notes that the non-derivability claim is “itself a meta-logical consequence of kernel consistency”. Our §6.1 reproduces both counter-models from the axioms alone, with 2 elements in each case, and certifies them against Nakamura’s unmodified library.

Benzmüller [21] ports the Isabelle/HOL dataset on Gödel’s and Scott’s ontological arguments to Lean 4 and to TPTP THF/SMT-LIB. The result is 294 theorems plus 45 statements that the original refutes or leaves open. None of the provers evaluated proves any of the 45. The ancillary files include `tools/thf/run_nitpick.py`, so Nitpick appears to be used on the THF side, but the material we read does not say how it is used. Those problems are higher-order (THF), which is outside our untyped `fof/cnf` scope. Their first-order fragments are a natural target for future work.

7.6 Comparison

	Logic / inputs	Finds models?	Kernel-certified?	General FOL?	Reusable tool?	Reflection?
Nitpick [3]	Isabelle/HOL goals (higher-order)	Yes (Kodkod)	No: Kodkod trusted	Yes, and more (HOL)	Yes	No
HolRefute [2]	HOL4 goals (higher-order)	Yes (Kodkod port of Nitpick; also Quickcheck, narrowing)	Partly: replayed to a HOL theorem when replay succeeds; uncertified Genuine verdicts allowed	Yes, and more (HOL)	Yes, but unmerged PR as of 2026-10-15	No: replay by evaluation and decision procedures
ETP [1]	Equational laws on magmas	Yes (brute force, SAT, Mace4, Vampire, other methods)	Yes: decide-style tactic, native_decide banned	No: one signature	No: project-specific	Not described
Goubault-Larrecq [32]	First-order clause sets (security protocols)	Yes (finite-model extraction)	Yes: Coq-checkable proofs	Clause sets of that domain	Research system	No (instrumented checker)

	Logic / inputs	Finds models?	Kernel-certified?	General FOL?	Reusable tool?	Reflection?
Lean Disprove [10]	Lean goals	Yes, within the goal’s types (enumeration, sampling, SAT/SMT)	Yes: REFUTED only when Lean accepts	No first-order model finder as described	Yes (lean4-skills)	Not described
plausible [12]	Lean props over sampleable types	Yes (random testing)	No: no proof produced	No	Yes	No
Lean on Vampire Proofs [8, 9]	Vampire proofs (FOL, HOL, theories)	No (proofs)	Yes	Yes	Yes	Not stated in abstract
PBLean / LRAT [13, 14]	PB / CNF proof certificates	No (unsatisfiability proofs)	PBLean: compiled (native_decide);/ PB) LRAT: Lean core’s verified checker, mode not verified by us	No (propositional)	Yes	Yes (verified checker)
This work	TPTP fof/cnf problems (untyped; §8 lists unsupported features)	Yes (Vampire -sa fmb)	Yes, under a strict kernel-only policy (propext, Quot.sound); reflected certificates prove ordinary satisfaction via sem_iff_sat	Yes (untyped FOL)	Yes	Yes (reflected mode); shallow mode also

7.7 What is new here

No single technique here is new. Finite model finding [4, 7, 24], kernel-checked finite counter-models in Lean [1], untrusted finders checked by replay [2, 10], provers connected to Lean [8, 16, 17, 18] and verified reflective checkers [13, 14, 15] all exist. The combination does not appear to: models of **arbitrary untyped first-order problems** found by an external prover, certified by Lean’s **kernel alone** through a **verified Tarski-semantics evaluator**, scaling to a million-fact problem.

We found no direct precedent for the exact combination, but our first search missed a close Coq predecessor [32], found in peer review. That is a reminder that a focused search supports “we found no implementation with this combination”, never “first ever”. The search had limits. On 2026-10-15 we searched the web for combinations of “Lean 4”, “finite model finder”, “counter-model”, “TPTP”, “Vampire”, “Mace4”, “certificate” and “kernel”. We searched for Lean ports of Nitpick or Nunchaku, including Lean Zulip discussions, and for verified first-order model checkers in Coq, Isabelle and Lean. We also read the sources above for mentions of model import. The closest results were the works in this section. The searches also surfaced SMTCoq (SMT certificate checking in Coq) and a February 2026 arXiv preprint on rewrite-system certificates for *infinite* counter-models in the unit equational fragment. We did not read either in full, so we do not cite them. We did not search the Lean Zulip archive directly or non-English literature.

8. Limitations and Future Work

8.1 Supported input

`cm.py` reads self-contained, untyped TPTP problems. Since peer review, unsupported features are **rejected with an error**, never translated: an earlier version silently misread several of them (§5).

Feature	Status
<code>fof</code> , <code>cnf</code> statements	supported
Every record read in full: whitespace and % comments between any two tokens; record names that are lower-case words, single-quoted words with any punctuation (<code>'thm.bool.TRUTH'</code>) or integers; optional annotations after the formula	supported, tested; text that is not a complete record makes the whole file rejected (§3.3)
Roles: <code>axiom</code> , <code>hypothesis</code> , <code>definition</code> , <code>assumption</code> , <code>lemma</code> , <code>theorem</code> , <code>corollary</code> , <code>plain</code> , <code>negated_conjecture</code> ; <code>conjecture</code> (negated in the certificate)	supported (non-conjecture roles are treated as assumptions)
Other roles (<code>type</code> , <code>unknown</code> , <code>question</code> , <code>fi_*</code> , ...)	rejected (no assertion meaning to certify)
Connectives <code>~</code> & <code>\ </code> <code>=></code> <code><=></code> ; quantifiers <code>!</code> <code>?</code> ; <code>=</code> , <code>!=</code> ; <code>\$true</code> , <code>\$false</code>	supported
Plain and quoted names (<code>cat</code> and <code>'cat'</code> are one symbol; <code>'X'</code> is never a variable)	supported, tested
% comments, including % inside quoted names	supported, tested
<code>/* ... */</code> block comments	rejected (<code>cm.py</code> strips only % comments; none of the 300 benchmark inputs uses them)
Connectives <code><~></code> , <code><=</code> , <code>~\ </code> , <code>~&</code>	rejected
<code>\$distinct</code> , other <code>\$</code> -defined symbols, numbers, strings	rejected
<code>tff</code> , <code>thf</code> , <code>tcf</code> , <code>tpi</code> statements	rejected (the whole file)
<code>include(...)</code>	rejected by <code>cm.py</code> . The benchmark harness expands includes beforehand with <code>bench.expand</code> (trusted, now tested by <code>test_include_expansion</code>): each plain <code>include('Axioms/F.ax')</code> outside comments and quoted names is replaced by that axiom file. A <i>selective</i> include, <code>include('Axioms/F.ax', [names])</code> , is left in place, so the input is rejected and recorded as <i>unsupported input</i> (SYN000+2). An unresolved include, an include of a file that itself includes, or one outside <code>Axioms/</code> is refused. Originals, expansions and axiom files are recorded with SHA-256s against the official TPTP v9.3.1 release (<code>bench/inputs.json</code>)
One symbol used at two arities (TPTP's <code>p/1</code> and <code>p/2</code> are distinct symbols)	rejected (the model tables are keyed by name)
Purely propositional problems Vampire settles without printing a finite model	not handled (no model to certify)

8.2 Limitations

Limitation	Effect	Path forward
No front end inside Lean	Users write TPTP by hand	A Lean command that exports the current goal (first-order part) to TPTP and applies the certificate. This is the step that makes the tool adoptable.

Limitation	Effect	Path forward
Reflected checking of nested quantifiers is slow	PLA025+1 did not finish in 400 s reflected, against 4.4 s shallow, so we dispatch by fact share	Decode once, then check (decode $D = L$, then check L), so that lazy kernel evaluation stops re-decoding subformulas inside quantifier loops
Trusted translation and rewriting	A bug here can certify the wrong statement. It happened once (§5) and was caught by an independent check; peer review prompted a second class (silent misreading of unsupported syntax), now closed by fail-closed parsing. An audit of the archived inputs at 1a55aef (bench/input-audit.json, on the earlier scraped inputs) found 3 development inputs with unsupported syntax; all 3 were <i>no model</i>, so no certificate was affected. The final runs apply the same checks to the rebuilt inputs before Vampire runs, and their <i>unsupported input</i> verdicts supersede that audit. The frozen run of the second sample then exposed variable capture: GEG001+1 binds a variable named T, the domain type's name; Lean rejected the certificate, and variables are now emitted primed (X'). The input rebuild from the TPTP release (§6.2) exposed include expansion inside comments, which had changed two inputs: two benchmark no-model-found outcomes concerned altered inputs and are superseded by the corrected runs. Round-4 review: record omission. The regular expression that found records skipped any record it could not match, and the test oracle had copied it, so both agreed on a weakened problem. It dropped records from 9 benchmark inputs, 8 of which had been reported certified; Lean had checked only the formulas it was given (§6.2.2). Records are now read by a total, fail-closed reader, and the oracle has its own. The oracle still shares the in-formula lexer and the variable collector with cm.py, so correlated errors in those two helpers remain possible.	Prove the rewrites in Lean, or emit both the original and the rewritten formula and prove them equivalent

Limitation	Effect	Path forward
Translator robustness	The third sample exposed two crashes, reported as failures: <code>cm.py</code> 's recursive-descent translator exceeds Python's recursion limit on very deep nesting (KRS290–1), and the benchmark's shallow path does not pass the output directory that <code>cm.py</code> needs for problems over 2,000 formulas (SY0595–1)	An iterative parser; pass the output directory and build the modules, as the reflected path already does
Untyped first-order logic only	No sorts, arithmetic or higher-order input	TPTP <code>tff</code> sorts, using Vampire's multi-sorted models
Vampire's limits are our limits	Within 30 s, no model for 19, 34 and 29 of the three 100-problem samples. By the TPTP headers, 18 of those 82 have no finite model at all and one needs 64 elements; 9 hit the finite-model builder's encoding limit; the other 54 timed out without a documented cause (§6.2.1)	Longer limits; other finders such as Mace4 or SAT-based search
Some formats are unread	Purely propositional problems that Vampire settles without printing a finite model	Parse Vampire's other model formats
Lean-version effects	Lean 4.13's limit of about 20,600 declarations per file may not hold on $\text{Lean} \geq 4.33$, which changed how kernel recursion is bounded	Re-test on current Lean before reporting upstream
Spinoza encoding	Our 2-element models let objects be both substances and attributes, as Nakamura's axioms allow	A sorted encoding; as §6.1 explains, separating substances from attributes would revise the relevant bridges or the signature, so it would be a different reconstruction, not a refinement of this one
Sample, not census	Three 100-problem samples: a development set (the first 20 picked by domain), a second sample used for debugging, and a third drawn and frozen before any result (§6.2.1)	The full 2,561, now cheap with caching and indexing
Hybrid dispatch threshold	The 90% fact-share threshold was chosen during development, and only a few problems use reflection	Validate it on problems with intermediate fact shares
Non-derivability, not independence	A counter-model shows a target does not follow; independence also needs its negation not to follow	Run both directions in <code>ablate.py</code>

9. Conclusion

Counter-models are how one shows that something does *not* follow. With `lean-countermodel`, Lean 4 users can get them for supported untyped first-order inputs for which the finder produces a checkable finite model: Vampire finds them, Lean's kernel certifies them, and Vampire is never trusted. The interface takes arbitrary signatures; what limits coverage is the supported TPTP fragment (§8.1) and the finder, which is incomplete and cannot help with theories that have only infinite models.

The tool is small. The work was in making certificates finish. One lesson recurred on the ground-fact workloads we measured: **Lean's cost lay in elaborating statements, not in evaluating them.** Each pattern that reduced elaboration paid off, among them balanced trees, hierarchical aggregation, compact encodings and reflection through a once-verified

evaluator. The patterns that added elaboration or unification work backfired, and we report those too.

The verified checker turns a certificate into a single statement: the ordinary first-order structure the model describes satisfies every formula of the problem. It depends only on Lean’s standard axioms, and every shipped certificate replays through `leanchecker --fresh`; so do all 185 strict certificates of the three benchmark samples. Separately, 189 of the 192 certified results (strict and native-assisted) were checked against their original source files; for the other three, that check did not finish and they are listed by name (§6.2.2). With it, a million-fact problem is certified in a median of 5.2 minutes on a laptop, about 2.7× faster than one theorem per formula under matched conditions, measured at an earlier commit (§6.3.1). The tool also reproduces, automatically and with smaller models, the hand-built counter-models of a published Lean formalization.

The remaining gap is the user’s side: a command inside Lean that takes a goal and returns a certified counter-model. The checker and the scaling recipe are the foundation for it.

Appendix A: Reproducing Every Number

Result	Command	Data
Unit and integration tests	<code>pytest test_cm.py</code> (Vampire on PATH or in \$VAMPIRE)	170 tests, all passing (bench/test-log.txt); 82 run without Vampire; in a restored package, without the TPTP corpus and the Vampire cache, 168 pass and 2 are skipped
Axioms and fresh kernel replay of all shipped certificates	<code>./recheck.sh</code>	writes bench/recheck.log
Examples (shallow)	<code>lake build; lake env lean Axioms.lean</code>	examples/, Cm/
One reflected certificate	<code>python3 cm.py --reflect examples/group_comm.p > G.lean && lake env lean G.lean</code>	—
Nakamura, against their library	<code>cd nakamura && lake build && lake env lean Axioms.lean</code>	spinoza-ethica-lean @ 6d4c046
Benchmark inputs, from the official TPTP v9.3.1 release (§6.2)	<code>python3 bench/inputs.py verify TPTP-v9.3.1 [TPTP-v9.3.1.tgz]</code> (checks against the committed manifest; record writes a candidate manifest instead)	bench/inputs.json (SHA-256 of the tarball, every original, expansion and axiom file)
TPTP development sample (original sampling)	<code>python3 bench.py N [WORKERS]</code>	bench/eligible.tsv, bench/results*.jsonl
Frozen runs (§6.2): all three samples at 66c8b32 (round-4 rerun); earlier runs kept as bench/superseded-*	<code>python3 bench/frozen.py <repo>/bench [dev second third],</code> from a worktree of that commit (usage in the file)	bench/heldout-sample.json, bench/third-sample.json, bench/results-final-{dev,second,third}.jsonl, bench/superseded-*.jsonl
Record-reading comparison with an earlier cm.py (§6.2.2)	<code>python3 bench/compare_parsers.py <commit> OUT.json</code>	bench/parser-comparison.json (against 30ad4f3), bench/parser-comparison-r4b.json (against 66c8b32)
Source fidelity of every certified result (§6.2.2)	<code>python3 bench/fidelity.py <repo>/bench 300</code>	bench/fidelity.jsonl, bench/fidelity.log
Bounded rewrite validation on the benchmark (§6.2.2)	<code>python3 bench/rewrite_check.py <repo>/bench 60</code>	bench/rewrite-check.jsonl, bench/rewrite-check.log
Coverage report, each check separate (§6.2.2)	<code>python3 bench/coverage.py <repo>/bench 300</code>	bench/coverage.md, bench/coverage.json

Result	Command	Data
Negative controls (§6.2.2)	bench/negative-controls.sh [TPTP-v9.3.1]: runs a selection of the regression tests and five failure controls	bench/negative-controls.log
Fresh kernel replay of every strict benchmark certificate (§6.2)	python3 bench/replay.py <repo>/bench, in the same worktree, after frozen.py see method in the file	bench/replay- {dev,second,third}.jsonl, bench/replay.log bench/input-audit.json
Input audit on the earlier scraped inputs (§8), superseded by the final runs' classifications		
Controlled timings (§6.3.1)	python3 bench/perf.py bench/problems/NLP263+1.p 3 4; python3 bench/perf.py bench/slice10000.p 3 4 (the first 10,000 statements of NLP263+1)	bench/perf.jsonl
NLP263	python3 cm.py --reflect bench/problems/NLP263+1.p DIR, then build the modules	bench/nlp263-*.json

Requirements: Lean 4.35 via elan (nakamura/ uses 4.13), Vampire 5.1.0, and Python 3.12 with pytest. Vampire 5.1.0 is a single binary from its GitHub release, for example `gh release download v5.1.0 -R vprover/vampire -p vampire-macOS-ARM64.zip` (other platforms have their own assets); put it on PATH or set VAMPIRE. The Lean project has no dependencies beyond Lean's core library, so `lake build` needs no network access after the toolchain is installed.

References

All sources were accessed on 2026-10-15. Marks used below:

- **[verified]**: we read the source itself (abstract page, paper text, repository or release notes), or its publisher metadata through Crossref.
 - **[partly verified]**: one part of the entry could not be confirmed. The entry says which part.
1. M. Bolan, J. Breitner, J. Brox, N. Carlini, M. Carneiro, F. van Doorn, M. Dvorak, A. Goens, A. Hill, H. Husum, H. Ibarra Mejia, Z. A. Kocsis, B. Le Floch, A. Livne Bar-on, L. Lucciolli, D. McNeil, A. Meiburg, P. Monticone, P. P. Nielsen, E. O. Osazuwa, G. Paolini, M. Petracci, B. Reinke, D. Renshaw, M. Rossel, C. Roux, J. Scanvic, S. Srinivas, A. R. Tadipatri, T. Tao, V. Tsyklevich, F. Vaquerizo-Villar, D. Weber, F. Zheng. *The Equational Theories Project: Advancing Collaborative Mathematical Research at Scale*. arXiv:2512.07087, 2025. <https://arxiv.org/abs/2512.07087> · PDF: https://teorth.github.io/equational_theories/paper.pdf **[verified]**: §3, §5 and §7 were read in the PDF text.
 2. HOL-Theorem-Prover/HOL pull request #2051 (author lukaszcz). *HolRefute: Quickcheck, narrowing and a Nitpick-style Kodkod model finder*. Opened 2026-09-04; **open and unmerged** as of 2026-10-15. <https://github.com/HOL-Theorem-Prover/HOL/pull/2051> **[verified]**: we read the PR description and the first part of the page. The remaining 60k characters, mostly commit history, were not read.
 3. J. C. Blanchette, T. Nipkow. *Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder*. Interactive Theorem Proving (ITP 2010), LNCS, pp. 131–146, Springer, 2010. https://doi.org/10.1007/978-3-642-14052-5_11 **[verified]** (Crossref metadata; the description is from the authors' page at <https://www21.in.tum.de/~nipkow/pubs/itp10.pdf>)
 4. E. Torlak, D. Jackson. *Kodkod: A Relational Model Finder*. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007), LNCS, pp. 632–647, Springer, 2007. https://doi.org/10.1007/978-3-540-71209-1_49 **[partly verified]**: title, authors, venue and pages come from Crossref. The Crossref record has no year; 2007 is inferred from the TACAS 2007 volume.
 5. A. Reynolds, J. C. Blanchette, S. Cruanes, C. Tinelli. *Model Finding for Recursive Functions in SMT*. IJCAR 2016. <https://homepage.divms.uiowa.edu/~tinelli/papers/ReyEtAl-IJCAR-16.pdf> **[partly verified]**: authors, title, and the fact that the method is implemented in CVC4 and the “higher-order model finder” Nunchaku were confirmed from a search result describing this PDF. We did not open the PDF itself.

6. Nunchaku (repository). <https://github.com/nunchaku-inria/nunchaku> [verified]: the README states “currently in initial development phase, not ready for use”. No last-commit date was visible.
7. W. McCune. *Mace4 Reference Manual and Guide*. Argonne National Laboratory Technical Memorandum ANL/MCS-TM-264, 2003. arXiv:cs/0310055. <https://arxiv.org/abs/cs/0310055> [verified]
8. J. Bodingbauer, M. Hajdu, L. Kovács, A. Polaczek, M. Rawson. *Lean on Vampire Proofs (Short Paper)*. 17th International Conference on Interactive Theorem Proving (ITP 2026), LIPIcs 382, 36:1–36:9, 2026. <https://doi.org/10.4230/LIPIcs.ITP.2026.36> (also arXiv:2603.26342) [verified] (the publisher record lists vprover/vampllean and Vampire’s leancheck branch as supplementary software)
9. vprover/vampllean: “Library required by Lean for checking generated proof files”. <https://github.com/vprover/vampllean> [verified]: we confirmed the repository and its description; the publisher record of [8] lists it as that paper’s supplementary software.
10. J. Ondras, C. Freer. *Lean Disprove: Certified Counterexample Search for AI-Assisted Formal Mathematics*. 3rd AI for Math Workshop: Toward Self-Evolving Scientific Agents, ICML 2026. <https://icml.cc/virtual/2026/82610> · OpenReview: <https://openreview.net/forum?id=5ck1jRE65S> [verified]: we read the ICML listing. The first author’s full name is Jan Ondras. We did not open the OpenReview page.
11. C. Freer et al. lean4-skills (repository; includes the disprove workflow). <https://github.com/cameronfreer/lean4-skills> [verified]
12. plausible: a property-based testing framework for Lean 4. <https://github.com/leanprover-community/plausible> [verified]
13. S. Szeider. *PBLean: Pseudo-Boolean Proof Certificates for Lean 4*. arXiv:2602.08692, 2026. <https://arxiv.org/abs/2602.08692> [verified]: we read the abstract. The use of native_decide was confirmed from a search summary of the paper’s HTML version.
14. S. Szeider. *Streaming LRAT Certificates into Lean Theorems*. arXiv:2607.00815, 2026. <https://arxiv.org/abs/2607.00815> [verified]: abstract only. Whether the final check runs in the kernel or as compiled code is not settled by the abstract.
15. Lean 4.12.0 release notes (bv_decide). <https://lean-lang.org/doc/reference/latest/releases/v4.12.0/> [verified]
16. Y. Qian, J. Clune, C. Barrett, J. Avigad. *Lean-Auto: An Interface Between Lean 4 and Automated Theorem Provers*. Computer Aided Verification (CAV 2025), LNCS, pp. 175–196, Springer, 2025. https://doi.org/10.1007/978-3-031-98682-6_10 [verified] (Crossref metadata only)
17. T. Zhu, J. Clune, J. Avigad, A. Q. Jiang, S. Welleck. *Premise Selection for a Lean Hammer*. arXiv:2506.07477, 2025. <https://arxiv.org/abs/2506.07477> [verified]
18. J. Clune, Y. Qian, A. Bentkamp, J. Avigad. *Duper: A Proof-Producing Superposition Theorem Prover for Dependent Type Theory*. ITP 2024, LIPIcs 309, 10:1–10:20, 2024. <https://doi.org/10.4230/LIPIcs.ITP.2024.10> [verified]
19. A. Florath. *Formal Foundations and Proof-Carrying Certificates for q-ary Covering Codes in Lean 4*. arXiv:2606.09600, 2026. <https://arxiv.org/abs/2606.09600> [verified] (abstract)
20. A. Florath. *A Lean-Certified Proof of $K_8(4, 2) = 23$* . arXiv:2606.16688, 2026. <https://arxiv.org/abs/2606.16688> [verified] (abstract)
21. C. Benz Müller. *Gödel’s and Scott’s Variants of the Ontological Argument in Lean 4 and TPTP THF*. arXiv:2609.26806, 2026 (v3, 2026-10-01). <https://arxiv.org/abs/2609.26806> [verified]: we read the abstract and the ancillary file list. How the Nitpick script tools/thf/run_nitpick.py is used is **unverified**.
22. Y. Nakamura. *Bennett’s Conjecture in Lean 4: Counter-Models for the PSR-Reducibility of Spinoza’s Propositions V and XIV*. arXiv:2605.02331, 2026. <https://arxiv.org/abs/2605.02331> · Code: <https://github.com/Nakamura/spinoza-ethica-lean> [verified]: we read the abstract. The 4- and 3-element model sizes come from our own reading of the Lean source (§6.1).
23. G. Sutcliffe. *The TPTP Problem Library and Associated Infrastructure: From CNF to TH0, TPTP v6.4.0*. Journal of Automated Reasoning 59(4), pp. 483–502, 2017. <https://doi.org/10.1007/s10817-017-9407-7> [partly verified]: Crossref confirms the title “The TPTP Problem Library and Associated Infrastructure”, the author, volume 59, pp. 483–502, and 2017. The subtitle and the issue number (4) were not in the Crossref record.
24. L. Kovács, A. Voronkov. *First-Order Theorem Proving and Vampire*. Computer Aided Verification (CAV 2013), LNCS, pp. 1–35, Springer, 2013. https://doi.org/10.1007/978-3-642-39799-8_1 [verified] (Crossref metadata)
25. L. de Moura, S. Ullrich. *The Lean 4 Theorem Prover and Programming Language*. Automated Deduction – CADE 28, LNCS, pp. 625–635, Springer, 2021. https://doi.org/10.1007/978-3-030-79876-5_37 [verified] (Crossref metadata)
26. A. Nonnengart, C. Weidenbach. *Computing Small Clause Normal Forms*. In A. Robinson, A. Voronkov (eds.), *Handbook of Automated Reasoning*, pp. 335–367, Elsevier, 2001. <https://doi.org/10.1016/B978-0-444-50813-3/50008-4> [partly verified]: Crossref confirms the title, authors, book, pages and year. The editors and the statement that the chapter covers miniscoping are from general knowledge; we did not open the chapter.

27. Lean 4.33.0 release notes (#13956: kernel type checking bounded by `maxRecDepth`). <https://lean-lang.org/doc/reference/latest/re> [verified]
28. P. Grim, D. Singer. *Computational Philosophy*. Stanford Encyclopedia of Philosophy, first published 2020, substantive revision 13 May 2024. <https://plato.stanford.edu/entries/computational-philosophy/> [verified]
29. B. Fitelson, E. N. Zalta. *Steps Toward a Computational Metaphysics*. *Journal of Philosophical Logic* 36(2): 227–247, 2007. <https://doi.org/10.1007/s10992-006-9038-7> [verified via [28]’s bibliography]
30. P. E. Oppenheimer, E. N. Zalta. *A Computationally-Discovered Simplification of the Ontological Argument*. *Australasian Journal of Philosophy* 89(2): 333–349, 2011. <https://doi.org/10.1080/00048401003674482> [verified via [28]’s bibliography]
31. J. K. Horner. *A Computationally Assisted Reconstruction of an Ontological Argument in Spinoza’s The Ethics*. *Open Philosophy* 2(1): 211–229, 2019. <https://doi.org/10.1515/opphil-2019-0012> [verified via [28]’s bibliography]
32. J. Goubault-Larrecq. *Finite Models for Formal Security Proofs*. *Journal of Computer Security* 18(6): 1247–1299, 2010. <https://doi.org/10.3233/JCS-2009-0395> [verified]: extracts finite models of first-order clause sets for security protocols, model-checks them, and can output a formally checkable proof, e.g. in Coq.
33. D. Kirst, D. Larchey-Wendling. *Trakhtenbrot’s Theorem in Coq: A Constructive Approach to Finite Model Theory*. IJCAR 2020. Extended version: arXiv:2104.14445. <https://www.ps.uni-saarland.de/extras/fol-trakh/> [verified]
34. G. Sutcliffe. *The SZS Ontologies: Success*. <https://szs.tptp.org/success> [verified]: *Satisfiable* and *CounterSatisfiable* do not imply a finite model; *FinitelySatisfiable* and *FinitelyCounterSatisfiable* are separate statuses.
35. Lean 4.35 Language Reference: *Validating a Lean Proof*. <https://lean-lang.org/doc/reference/4.35.0/ValidatingProofs/> [verified]: since Lean 4.29, native evaluation adds one dedicated axiom per computation (names containing `_native`); the `Lean.trustCompiler` machinery was removed in 4.35.