hexagon:2611.00373v1

Re-proving `native_decide` facts in Lean's kernel: a study of 16 public packages and a converter

Tejasvi Singh Tomar

October 2026

Abstract

Lean's `native_decide` proves a decidable proposition by running compiled code and records the result as an axiom. Mathlib forbids such axioms, and `lake check` and Comparator in Lean 4.35 reject them, as do the independent proof checkers. NativeTrust re-proves the recorded facts in Lean's kernel, with only the standard axioms and without editing the projects that contain them. Where a definition blocks kernel evaluation and NativeTrust has a version proved equal to it (for example, a fuel-indexed form of well-founded recursion), the converter evaluates a copy that uses that version. It then proves the copy equal to the original, usually by congruence. With its default settings it converts 98.2% of 52,821 native facts in 16 public packages. The three largest packages hold 94% of these facts; outside them it converts 68.5%. Nearly every converted fact needed nothing beyond plain kernel evaluation. Two tools remove the native axioms that the kernel can prove within a budget: a single import that makes each call of the tactic try the kernel first, and a source rewriter.

1 Introduction

The tactic `native_decide` closes a goal p by compiling the decision procedure `decide` p and running it. Since Lean 4.29 each use adds an axiom `<decl>._native.native_decide.ax_... : decide p = true`; the tactics `decide +native` and `bv_decide` add similar axioms of the form $e =$ `true`. A proof that uses one trusts Lean's compiler and runtime and every `@[implemented_by]` and `@[extern]` implementation that the computation reaches. Mathlib forbids the tactic. Both `lake check` and Comparator admit only `propext`, `Classical.choice` and `Quot.sound`, and the independent checkers cannot check compiled evaluation at all. Even so, in October 2026, GitHub code search listed 585 repositories whose Lean files mention the tactic, 575 of which received pushes in 2026.

Other proof assistants make the same trade. Coq's `native_compute` and Isabelle's evaluation by generated code are fast because they trust a compiler, while proof by reflection with the kernel's own evaluation keeps the kernel as the only judge, at the cost of evaluating in the kernel (Section 9). Lean's kernel can evaluate decision procedures itself, as `decide +kernel` does, but code written for compiled evaluation often stops it: recursion defined through `WellFounded.fix`, machine words, hash maps and string processing. The tools we know of act on single uses of the tactic or on single files: they flag it, or replace it by a conventional proof or, where the kernel already succeeds, by `decide +kernel`.

NativeTrust instead re-proves the recorded facts of whole projects in the kernel, without editing the projects. For example:

```
theorem t : (List.range 300).reverse.mergeSort = List.range 300 := by native_decide
#print axioms t          -- [propext, Quot.sound, t._native.native_decide.ax_1_1]
kernelize t              -- adds t.kernel, with the statement of t
#print axioms t.kernel   -- [propext, Quot.sound]
```

`List.mergeSort` is defined by well-founded recursion, which Lean turns into a term that the kernel cannot evaluate (Section 2). The converter evaluates copies in which that term is replaced by a version indexed by fuel, which falls back to the original definition when the fuel runs out and is proved equal to it (Section 3.4). It then proves that the copies equal the originals, by congruence or, where that fails, by comparing them directly (Section 3); here the whole conversion takes 0.4 s. Where no replacement applies, it names the constant that blocks evaluation if it finds one. For a whole project, the driver writes the new theorems into a kernel supplement, one module for each module that names a native tactic. A theorem in any other module that uses an original theorem keeps its native axioms, since the supplement does not rebuild it (Section 3.7).

We ran the converter on the native facts of 16 public packages, chosen mostly for heavy use of the tactic, and of a random sample of 70 repositories from GitHub code search. Each fact ran in its own process with fixed limits of time and memory, and we classified the failures. The converter converts 98.2% of the packages' 52,821 facts. In the random sample the median package converts 99.6% of its facts and the pooled rate is 58.5% (Section 4.5). Most facts need no copy at all. For them the converter's contribution is to handle whole projects, including the facts inside definitions and instances. Rewriting decision procedures by hand, so that the kernel can evaluate them, is the usual alternative; the kernel versions that we built by hand for our own earlier results show how much that costs and where the converter can replace it (Section 8).

Contributions.

- NativeTrust, a converter that re-proves native facts in Lean's kernel and writes a kernel supplement: for each module that names a native tactic, kernel-checked counterparts of the module's theorems whose native facts all convert. Its fuel-indexed replacement for well-founded recursion is proved equal to the original without reducing any accessibility proof.
- An evaluation on 52,821 native facts in 16 packages and on a random sample of repositories, with an ablation of the converter's transformations and a classification of the failures.
- Tools that remove unnecessary `native_decide`: a one-import switch, a linter and a source rewriter with a per-fact budget. Opt-in options and add-ons for the facts that the default settings leave unconverted, among them a kernel-checked pipeline for `bv_decide`.
- Independent checks of three supplements and of samples from twelve packages, and a comparison with the hand-built kernel versions of our published results.

Section 2 lists what stops kernel evaluation, Section 3 describes the converter and Section 4 evaluates it. Sections 5 to 8 cover the removal tools, the independent checks, the `bv_decide` pipeline and the comparison with hand-built kernel versions, and Sections 9 to 11 discuss related work, lessons and limitations.

2 Background

Lean's kernel evaluates closed terms by reduction, with built-in arithmetic on natural-number literals, and `decide +kernel` relies on this evaluation. Four things stop kernel evaluation.

1. *Well-founded recursion.* Lean 4.27–4.36 elaborate a definition whose termination measure is not a natural number into `WellFounded.fix (Lean.opaqueId wf) F`. The kernel cannot unfold `opaqueId`, so it cannot evaluate, for example, `List.mergeSort`. Natural-number measures are elaborated into the fuel-based `WellFounded.Nat.fix`, which reduces.

2. *Constants without a usable definition.* These include `partial` functions (opaque in the logic), C primitives declared `@[extern]`, the platform word size behind `USize` and `HashMap`, and definitions that use `Classical.choice` computationally, such as `partial_fixpoint` definitions (`Lean.Order.fix`).

3. *Casts along propositional equalities.* `Eq.rec` along an equality proved with `propext` or `Quot.sound` does not reduce unless its two sides are definitionally equal; compiled code erases such casts.

4. *Cost.* The kernel interprets terms by reduction, which is slower than running compiled code; within one declaration its caches grow with the length of the evaluation, and deep recursion can exhaust its stack.

3 Method

Definitions. A fact is *converted* when NativeTrust produces a theorem with the fact's statement (or its restatement over rebuilt definitions, Section 3.1) that Lean's kernel accepts within the time and memory limits of Section 4.1 and that depends only on the three standard axioms. A *blocking constant* is a constant at which kernel evaluation stops when it is used for computation: an axiom (including `Classical.choice`, which the logic admits but the kernel cannot compute with), an opaque constant (including `partial` and `@[extern]` functions without a definition), or a definition that needs one of these to compute.

Copies and congruence. For a native axiom $e = \texttt{true}$, NativeTrust copies every definition that the evaluation of e can reach and that needs a change. Inside the copies it replaces `WellFounded.fix` (Section 3.4) and any replacement registered by the user (Section 3.2) by versions the kernel can evaluate. The kernel then checks $e' = \texttt{true}$ by `rfl` for the copied term e', and $e = e'$ is proved by congruence from per-definition equations $c = c'$.

Lemma 1. *Let the body of c' be the body of c with constants $d_1, \ldots, d_k$ written as $d'_1, \ldots, d'_k$, where no d_i occurs in the type of c and the body of c with the d_i abstracted is well typed. If $d_i = d'_i$ for every i, then $c = c'$, by a proof whose type check need not evaluate either side.*

Proof. Let M be the body of c with $d_1, \ldots, d_k$ abstracted, so that c unfolds to $M\, d_1 \cdots d_k$ and c' to $M\, d'_1 \cdots d'_k$. Applying `congrArg` to M with the equations $d_i = d'_i$, one argument at a time, proves $c = c'$; checking it unfolds c and c' once and compares their bodies up to the abstracted constants. □

The converter proves the per-definition equations in dependency order by Lemma 1 where it applies; the equations $d_i = d'_i$ are `fix_eq_fixFuel`, the proofs of registered replacements or earlier copy equations. Comparing an original closed term with its copy directly could make the kernel evaluate both, so the converter does so only when congruence fails; for the per-definition equations it then runs under a heartbeat bound. When nothing needs a change, this is plain kernel evaluation of e. In the example of Section 1, the copies replace the `WellFounded.fix` in the definitions of

`List.mergeSort` and `List.merge` by `fixFuel`; `fix_eq_fixFuel` and `congrArg` prove each copy equal to its original, the kernel evaluates the copied statement by `rfl`, and the original statement follows. The converter made four copies there: one of `List.mergeSort` and three of definitions that it uses.

3.1 Facts inside definitions and instances

Declarations such as

```
def cert := ⟨..., by native_decide⟩
instance : Fact (Nat.Prime p) := by native_decide
```

put a native axiom inside data that other facts mention. Such facts are proved first, compiled to their own modules, and reused by every later fact whose module imports them. A statement that mentions such a definition is restated over its rebuilt copy. When the rebuilt declarations are definitions or theorems, the two statements are definitionally equal by proof irrelevance, though not expression-identical; a rebuilt `opaque` constant does not unfold, so a statement over it would be a different proposition (Section 3.7 says what checks this). In the corpus of Section 4, 342 facts sit inside definitions or instances; in the random sample, 34 do.

3.2 Externally proved facts and proved replacements

A declaration `@[kernel_fact] theorem h : P` registers a proof of P. A native fact `decide` $P' =$ `true` uses it when P' is syntactically equal to P, or definitionally equal at reducible transparency. The proof is then `decide_eq_true` h, with the fact's own decidability instance, and the converter rejects a registered proof that uses axioms beyond the standard three. We provide Pratt certificates [28], checked through Mathlib's Lucas primality theorem and a structurally recursive modular exponentiation, for the field primes of common zero-knowledge and cryptographic libraries: BabyBear, KoalaBear, Mersenne31, Goldilocks, BN254 (both fields), secp256k1 (both), BLS12-381 r and Curve25519. The kernel checks a 256-bit certificate in 0.24–0.57 s. Eight corrupted certificates were rejected.

Similarly, a declaration `@[kernel_replace] theorem t : @f = @g` registers g as a replacement for f inside copies. The converter uses the body of g as it stands, so g must itself be evaluable. Long string literals are replaced without any registration: in the decided term and in the copies of the fuel route, a literal of 128 or more characters goes through a version of `String.ofList` that builds the byte array in one step, proved equal to it. Its evaluation is then linear in the length, while on the original path a non-repetitive literal costs time and memory quadratic in the length (Section 4.3).

3.3 Triage, bounded evaluation and splitting

A static pass lists the blocking constants that the evaluation can reach, with the chain of definitions that leads to each. It follows every constant in definition bodies, including higher-order arguments and instances, as the copies would. It does not enter theorems, and it skips panic and trace messages, which kernel evaluation does not reduce. It does follow constants inside the proofs that definitions contain, although evaluation never reaches them. Accessibility proofs, which the fallback route of Section 3.4 uses, are treated separately. The pass is designed to over-approximate, but we have not proved that it does. When it finds a blocking constant, evaluation is capped at 50,000 in the units of Lean's `maxHeartbeats` option ($5 \cdot 10^7$ heartbeats), because a stuck evaluation can exhaust memory before it fails. A capped run can fail for lack of budget and still count as blocked, so we re-ran

a sample without the cap (Section 4.4). Checks of the form $a \,\&\&\, b$ or `List.all` l f are split into separate kernel declarations joined by small lemmas, which reduces the memory each declaration needs. Before the converter searches for such split points, it tries the whole term within a small budget, so that cheap terms need no search, and each step of the search has its own budget. A bounded existential ($\exists\, x < n$ or $\exists\, x \in l$) or a `List.any` is checked at the witness that compiled code proposes, so a wrong witness only makes the check fail.

3.4 Well-founded recursion by fuel

```
def fixFuelAux (hwf : WellFounded r) (F : ∀ x, (∀ y, r y x → C y) → C x)
    (n : Nat) : ∀ x, C x :=
  Nat.rec (fun x => WellFounded.fix hwf F x) (fun _ ih x => F x (fun y _ => ih y)) n
def fixFuel hwf F x := fixFuelAux hwf F 18446744073709551616 x
theorem fix_eq_fixFuel : @WellFounded.fix = @fixFuel
  -- induction on n, WellFounded.fix_eq
```

The auxiliary function `fixFuelAux` unfolds the functional n times and then falls back to `WellFounded.fix`.

Lemma 2. *For all n and x,* `fixFuelAux` *hwf F n x =* `WellFounded.fix` *hwf F x. Hence* `fix_eq_fixFuel` *holds.*

Proof. By induction on n, using only `WellFounded.fix_eq`. For $n = 0$ the two sides coincide. For $n + 1$ the left side is $F\, x\, (\lambda y\, _.$ `fixFuelAux` *hwf F n y*$)$, which the induction hypothesis turns into $F\, x\, (\lambda y\, _.$ `WellFounded.fix` *hwf F y*$)$, the right side of `WellFounded.fix_eq`. Taking $n = 2^{64}$, function extensionality gives the equation of the two constants. □

No accessibility proof is reduced, and `Acc` is not eliminated into a type. Evaluation relies on the kernel's support for natural-number literals, which turns a literal $n > 0$ into `Nat.succ` $(n - 1)$ when a recursor is applied to it. Hence `Nat.rec` on 2^{64} unfolds one step per recursive call actually made, and an evaluation that finishes never reaches the final `WellFounded.fix`. Outputs that rely on this were accepted by nanoda, with its natural-number extension, and by con-ron. We call conversion through these copies the *fuel route*. Lemma 2 applies to every definition built on `WellFounded.fix`, but whether congruence can then bridge the copy depends on the definition (Section 4.6).

Proofs inside copies. A copied body can contain proofs about the constants being replaced: `Multiset.sort` is `Quot.liftOn` with a proof about `List.mergeSort`, and `Nat.find` comes with `Nat.find_spec`. A copy equals its original only propositionally, so substituting inside such a proof makes it ill-typed. The converter keeps these proofs as they are and transports them along the copy equations (`Eq.mp`), and it proves the copy equation by dependent congruence, abstracting each replaced constant together with its equation. This runs as a retry of the fuel route, after a first attempt without it fails (Section 4.6). The retry also keeps the original of a definition whose copy would equal it (the unchanged-copy rule). Copies of definitions that also occur in types, such as an instance inside the type of another definition, cannot be bridged this way; Section 3.5 describes how the retry handles most of them.

The previous version of NativeTrust used a lazily unfolding accessibility proof ($2^{64} - 1$ layers of `Acc.intro`, as in Coq's `Acc_intro_generator`), bridged by proof irrelevance. It needs the kernel to reduce `Acc` proofs. Core PR #14892 (merged 14 October 2026, "define well-founded recursion without large elimination of `Acc`") [13] removes that reduction, and con-ron rejects the construction. When the fuel route fails even with its retry and with the parametric route of Section 3.5, the

converter falls back to the accessibility route, which works up to Lean 4.36, and records the fact as `acc_fallback`.

3.5 Retries

Proof transport (Section 3.4) is one of several transformations that run only after a fact's first attempt fails, so a fact that converts at its first attempt converts as it would without them. This subsection describes the others.

Iterator-based string functions loop through $\texttt{WellFounded.extrinsicFix}_3$, and their iterator state types are defined through an opaque constant (`Std.Internal.idOpaque`), so copies of the iterator functions cannot be bridged by congruence. The library therefore replaces the string functions one level up: it gives list-level versions of the iterator-based `String` functions that the failing facts use (`toNat?`, `foldl`, `replace` with a string pattern, `contains`), proved equal to the originals from core lemmas such as `String.Slice.foldl_eq_foldl_toList`. A function generic in its pattern is replaced for one pattern type: its applications are first folded into a specialized constant, a definitional step.

A cast along an equality proved with `propext` or `Quot.sound` stops the kernel when its two sides are not definitionally equal (Section 2), although no constant is missing, so the static pass does not report it. A diagnostic follows the kernel's weak head normal form into stuck recursors and names the cast and the definition that contains it. When the cast sits inside a `Decidable` or `Fintype` instance, the converter builds a cast-free version of the instance (by splitting the proposition with `casesOn`, by `decidable_of_iff` over the fields of a structure, by `Fintype.ofFinset`, or by turning the cast into `decidable_of_iff`), proves it equal to the original by `funext` and `Subsingleton.elim`, and retries. Both are kernel-checked, and any value of a `Decidable` or `Fintype` type is correct by construction, so nothing new is trusted.

Lemma 1 requires that no copied constant occurs in the type of c. A definition that must be copied can occur in types, as the instance `instSizer` does here through the field type of `HasFmt`:

```
instance instSizer : Sizer Nat where size n := ack 1 n  -- ack: well-founded recursion
structure Fmt (α : Type) [Sizer α] where enc : α → Nat
class HasFmt where fmt : Fmt Nat                        -- field type: @Fmt Nat instSizer
```

The copy of such a definition is not definitionally equal to it, so every type that mentions it changes, and the copies around it no longer fit their original types. A further route, tried when the fuel route and its retry fail, therefore makes the replacement parametric. The replaced constants r (`WellFounded.fix`, registered replacements and cast-free instances) become variables: every definition c whose value or type mentions one of them, or a definition rewritten in this way, gets a generic version

$$c.\texttt{gen} : \forall\,(F : T)\,(h : r = F),\ T_c[r \mapsto F,\ d \mapsto d.\texttt{gen}\,F\,h],$$

whose type and value are rewritten together, so that the generic definitions fit each other for every F. Closed proofs are kept and cast along h. Inductive types are not copied: where a constructor or field type mentions a rewritten definition, values are re-wrapped field by field, which is the identity up to structure eta at the original constants. The decided term e becomes $E\,F\,h$.

Lemma 3. *Let $h_r : r = r'$ be the equation of the replacement r' of each replaced constant r. Then $e = E\,r\,\texttt{rfl}$ and $E\,r\,\texttt{rfl} = E\,r'\,h_r$.*

Proof. With $F = r$ and $h = \texttt{rfl}$, every generic definition unfolds to its original, since casts along `rfl` reduce and the re-wrapping of fields is the identity up to structure eta; so e and $E\,r\,\texttt{rfl}$ are definitionally equal. The generic definitions are well typed for every F, so abstracting one replaced

constant together with its equation and applying dependent congruence along h_r replaces r by r'; one constant at a time, this proves the second equation. □

The kernel then evaluates $E\,r'\,h_r$ to `true`, and the original fact follows. The retry of the fuel route also copies neither type formers nor proofs, and it transports proofs nested inside types as in Section 3.4.

`USize` depends on the opaque word size `System.Platform.getNumBits`, and so do `HashMap` and `HashSet` through it; core proves that the word size is 32 or 64. A retry evaluates a term that reaches it once for each value and joins the two kernel proofs by cases; a result that differs between the two word sizes is not proved. Because the kernel computes with numerals only in closed terms, each case is first checked on a copy in which the hypothesis on the word size is replaced by a closed stand-in, and this copy never enters a proof. Hash maps keyed by strings remain out of reach, since `String.hash` has no definition.

3.6 Options and add-ons

Four options are off by default. With `stage`, intermediate values of a computation are computed by compiled code, proved one by one in the kernel as separate lemmas, and joined by a kernel-checked bridge, so that no single declaration evaluates the whole computation; compiled code only proposes the values, as it proposes witnesses in Section 3.3, and the axioms of the final theorem are checked. The option `csimp` uses the proved equations that Mathlib registers for its compiled code (a product formula for binomial coefficients, trial division up to the least factor, binary-splitting factorials) as replacements. With `propSplit`, a proposition is checked along its logical structure: conjunctions and bounded quantifiers are split into parts, and a disjunction is checked at the disjunct that compiled code names. With `shards`, the parts of a split check go into separate modules, compiled in parallel and joined by a module that checks every imported statement.

Add-ons extend the converter for particular facts. The kernel-checked pipeline for `bv_decide` (Section 7) rebuilds the user's theorem with a kernel-checked proof in place of its native fact. The add-on for `@[csimp]` equations and the package add-ons, each written for one package, are modules that the driver imports on request; they register proofs with `@[kernel_fact]` or proved replacements (Section 3.2), for example primality proofs for the primes that a package uses, or a replacement for a package's tokenizer.

3.7 Driver and supplement

The driver, `lake exe ntrust`, runs one process per fact under a wall-clock limit and an address-space cap, proves facts inside definitions and instances first, and writes three outputs. The first is a report with one line per fact. The second is a *kernel supplement*: one module per source module that names a native tactic, with a theorem T.`kernel` for every theorem T of that module that depends, directly or through other declarations, on a native fact, provided all those facts were converted; theorems of other modules get none. The third is the list of pairs $(T, T.\texttt{kernel})$ for Comparator. Inside a file, the command `kernelize` T adds T.`kernel` for one theorem, as in the example of Section 1. The static pass also drives a linter (Section 5).

What is trusted. The converter itself is not trusted. By the definition at the start of this section, the theorem of a converted fact is accepted by Lean's kernel and uses only the standard axioms, and registered proofs, replacements and add-ons enter only as kernel-checked theorems. Whether a new theorem states the original fact is a separate question. The converter declares each T.`kernel` with

T's type, except for statements restated over rebuilt definitions (Section 3.1); the statement check of Section 6 confirms the statements of the three supplements checked there, and the pairs written for Comparator allow the same comparison for any supplement. Lemmas 1–3 explain why the proofs exist and why their checks are cheap; soundness does not depend on them. The static pass, the diagnostics and the budgets decide only what is tried, so an error in them can cost conversions or mislabel a failure, but it cannot produce a wrong theorem.

4 Evaluation

4.1 Setup

Corpus. We examined 30 public repositories (Appendix A). They were the repositories with the most files that mention `native_decide` in code search, plus research developments, a corpus of AI-written formalizations, and software-verification projects. Sixteen packages contain native facts that build at their pinned toolchain (Lean 4.28–4.35); rs-mca has 8 sub-projects, which we count as one package. In our setup, 546 modules did not build, 263 of them in TrainVerify, and the artifact lists all of them. Facts in those modules are not counted.

Unit and limits. The unit is one native fact: one native axiom on Lean $\geq$4.29, or one use of `Lean.ofReduceBool` on Lean 4.28. A single tactic call can produce several facts (e.g. under `all_goals`). The per-fact driver measured 52,725 facts: 52,523 from `native_decide`, 197 from `decide +native` and 5 from `bv_decide`. A per-theorem driver measured the other 96, in six rs-mca sub-projects on Lean 4.28, where the facts are uses of `Lean.ofReduceBool` rather than axioms. Each fact ran in its own process with a 900 s wall-clock limit and a 48 GiB address-space cap, with many processes in parallel (Appendix B); facts that failed for time or memory were re-run with 3,600 s and 96 GiB (the heavy pass).

Runs. The numbers in this section come from the library in the artifact (the final run). The tables also give, as a baseline, an earlier run of the same facts with the same driver and limits but older builds of the library, which lacked the long-literal rewriting, the search budgets, the witness checks and most of the retries, and differed in the triage (Appendix B). All runs used 96-core cloud machines with 743 GiB of memory.

4.2 Results

Table 1 lists the packages and the facts converted in the final run and in the baseline run. In the final run, 10 packages convert at least 95.1% of their facts, and the other 6 convert at most 23.1%. Per package, the median rate is 98.4% and the mean 64.3%. The pooled rate is 98.2%, but TrainVerify contributes 86.6% of all facts. Without it the pooled rate is 86.4%, and without the three largest packages it is 68.5%. In both the corpus and the random sample, every fact that the baseline run converted converts again. A converted fact took a median of 2 s including the import (90th percentile 3 s), with a median peak memory of 3.3 GiB. These medians mostly reflect TrainVerify. No evaluation in the corpus that finished reduced to `false`.

4.3 What stops the rest

Table 2 classifies the 969 facts that the final run does not convert, next to the baseline run's 1,057. A failure is *traced* when a reachable blocking constant or a stuck construct was identified, and its

package	domain	facts	baseline	final run	rate
TrainVerify (distributed-training graphs)	software	45,720	45,707	45,714	99.99%
cvc5/logos (checker for cvc5 proofs)	software	2,185	2,185	2,185	100%
Humanfia IChO 2026 (AI-written)	chemistry	1,864	1,860	1,863	99.9%
formal-conjectures	mathematics	690	664	677	98.1%
longfellow-zk (Lean model)	software	690	626	656	95.1%
asi-stack-book	software	392	383	391	99.7%
rs-mca (8 sub-projects)	mathematics	107	104	106	99.1%
sounio	software	77	70	76	98.7%
lido-srv3 proof closure	software	70	70	70	100%
cedar-spec	software	26	10	26	100%
BerryEsseen	mathematics	579	2	5	0.9%
Strata	software	286	66	66	23.1%
clean (zk circuits)	software	120	17	17	14.2%
aks	mathematics	7	0	0	0%
sorting-network	mathematics	7	0	0	0%
aeneas	software	1	0	0	0%
total		52,821	51,764	51,852	98.2%

Table 1: Native facts converted automatically per package, in the baseline run and in the final run; the rate is the final run's.

class names that constant or construct. In the final run, 231 failures are traced and 738 are not.

Strings. In the baseline run the largest class came from string processing at check time. BerryEsseen stores its cover certificates as strings and parses them inside the decided term, and 569 of its 579 facts were blocked there. The 16 cedar-spec failures were parsing tests, while every native fact that cedar-spec uses inside proofs converted. The class is empty in the final run. With the string replacements of Section 3.5, all 16 cedar-spec facts convert, and BerryEsseen's facts are no longer blocked but run out of memory. Each of its checks parses certificate literals of up to 100 KB in the kernel, and memory grows steadily until the cap: the checks stop at 48 GiB after a median of 121 s. In the heavy pass they stop at 96 GiB after a median of 233 s, and one of 567 converts. Package add-ons and sharding convert all of them, although the add-ons have runs longer than the time limit (Section 4.7). In a trial that applied the replacements in the first attempt instead of a retry, Strata copied formatting code that its facts reach but never run, and 32 of the 66 Strata facts that convert timed out.

String literals. A literal is unfolded to `String.ofList`, whose bytes come from `List.toByteArray`, a left fold of `Array.push`; each `Array.push` appends to the end of the array's list. For prefixes of one of BerryEsseen's certificate literals, in files that import nothing, computing the UTF-8 size in the kernel takes 21 s and 4.4 GiB at 1,000 characters; at 2,000 characters it takes 84 s and 15.6 GiB (with a raised `maxRecDepth`). A synthetic non-repetitive literal of 1,000 characters in a file that imports nothing behaves the same (19 s, 4.4 GiB), and so does a fold of `Array.push` over the same bytes without any string. Peak memory fits 0.45 GiB plus 48.6 KiB per byte plus 8.27 KiB per distinct substring of the bytes (64 runs, worst error 2.1%), so repetitive literals are cheap and non-repetitive ones quadratic; by this fit, BerryEsseen's 7,824-character literal would need about 240 GiB. Replacing only `List.toByteArray` by the one-step version of Section 3.2 makes the cost linear: for `d.utf8ByteSize > 0` on the same literals, the kernel needs 23, 36, 67, 131 and

class (constants reached)	baseline	final	packages (final run)
iterator-based string functions (`WellFounded.extrinsicFix`$_3$; iterator state types defined through `Std.Internal.idOpaque`)	585	0	
Lean's metaprogramming layer (`partial` parser, `PersistentHashMap`, unsafe implementations, `partial_fixpoint`), `HashMap`, `USize`	220	220	Strata
casts along equalities proved with `propext` or `Quot.sound`, inside decidability instances*	13	0	
C primitives and platform constants (an `mmap` data loader, `Float`, `USize`)	6	6	aks 2, sorting-network 2, sounio, aeneas
the LRAT parser of `bv_decide` (`partial`, hashing)	1	5	clean 4, asi-stack-book 1
time, memory or recursion-depth limit reached	230	738	BerryEsseen 574, clean 99, longfellow-zk 34, formal-conj. 13, TrainVerify 6, aks 5, sorting-network 5, Humanfia 1, rs-mca 1
not classified (earlier driver version)	2	0	
total	1,057	969	

Table 2: The facts not converted, in the baseline run and in the final run. The first five classes are traced. *Found by a diagnostic added after the baseline run.

257 MiB at 500, 1,000, 2,000, 4,000 and 8,000 characters, against 1.0, 4.0 and 15.9 GiB at 500, 1,000 and 2,000 characters on the original path, and at most 1 s. Both sets of figures are measured above the 1.5 GiB of a file that imports Lean and does nothing else. The cause is therefore the `Array.push` fold in `List.toByteArray`, which the library's one-step version (Section 3.2) avoids. Core's string-equality simprocs were changed in March 2026 so that their proofs no longer make the kernel run the UTF-8 encoding [47], but other evaluations of literals still take this path.

Casts. Thirteen facts stopped at a cast in the baseline run. The decidability instances involved were built with `simp <;> infer_instance` (asi-stack-book), come from Mathlib's `Fintype` instance for the edge sets of simple graphs, or are derived for a quotient type (`Nat.Partition`). With cast-free instances (Section 3.5), all 13 convert in the final run. In formal-conjectures' copy of Mathlib, 61 of the 84 such instances that hold a cast of data get a cast-free version.

The rest. Strata's checker runs through Lean's parser and environment infrastructure, and the kernel cannot evaluate its `partial` functions, persistent hash maps, unsafe implementations and `partial_fixpoint` definitions. A package add-on converts the 34 Strata facts that are blocked by hash maps keyed by strings (Section 4.7). The six C-level failures load data through a project-specific `mmap` extern, call `Float` functions, or use `USize`. All five `bv_decide` facts stop at their LRAT parser, and the pipeline of Section 7 converts them.

Resources. For resource failures the driver records no blocker: either none is reachable, or the run reached its limit before the static pass reported. Besides BerryEsseen's certificate parsing (above), clean's facts depend on instances `Fact (Nat.Prime p)` proved by `native_decide` and stop at the kernel's recursion depth, as do 21 longfellow-zk facts that evaluate Mathlib's binomial coefficient by Pascal's rule, about 8.7 million levels deep. In the baseline run, 569 of the 638 time and memory failures outside our repository (228 in the corpus, 410 in the sample) had run out in the meta-level search for split points rather than in the kernel; the budgets of Section 3.3 address this. Of these 638, the final run converts 276, and the options and add-ons raise the number to 534 (Section 4.7).

The other 104 are mostly beyond kernel evaluation as it stands. In 32 of them a decision procedure builds an array by repeated `push`, and since an array is a list in the kernel, each push copies it (one cSHAKE circuit of 607,242 gates takes about $1.8 \cdot 10^{11}$ list steps to build). Another 18 decode a serialized term of 566 KB (about 10^{13} steps), ten facts of aks and sorting-network read certificate strings of up to 2.95 GB, and six facts evaluate multi-scalar multiplications on an elliptic curve; four of these six perform 2,048 scalar multiplications of about 12 s each. Seven longfellow-zk facts check certificates in exact rational arithmetic.

4.4 Validation and ablation

Labels without the cap. With the cap switched off, 20 randomly sampled Strata facts labeled blocked fail again in the final run. In the baseline run the check covered 56 facts: 20 sampled from each of BerryEsseen and Strata, and all 16 of cedar-spec. All of them failed; the BerryEsseen facts failed at once, with the kernel stuck on the same opaque constant. The BerryEsseen and cedar-spec facts are no longer blocked.

Ablation. We re-ran the final run's conversions outside definitions and instances with well-founded replacement and splitting switched off: 51,376 of its 51,513 such conversions, all but those of the per-theorem driver and those made only in the heavy pass. This is close to applying `decide +kernel` to each fact, except that facts inside definitions and instances are still rebuilt first. In this run, 51,344 converted again. The 32 that did not are six TrainVerify facts that need splitting and 26 facts blocked at `WellFounded.fix` (18 in formal-conjectures, five in cedar-spec, two in BerryEsseen and one in cvc5/logos). Without splitting, five of the TrainVerify facts are capped and run out of budget, because the static pass finds a constant of Mathlib's real numbers reachable from the whole check, and the sixth runs out of memory. A fact that converts without the replacement never evaluates a definition built on `WellFounded.fix`, since the kernel cannot unfold `opaqueId` (Section 2). All 1,772 of these conversions that made well-founded copies were also run without the replacement, and the same 26 failed. The other 49,604 conversions made no such copies.

4.5 A random sample of repositories

The corpus is a selection that favors heavy users. As a check, we sampled from the 585 repositories that GitHub code search returned for `native_decide language:Lean` in October 2026. We excluded 35 repositories we had examined or inspected before (24 of the 30 in the corpus and 11 others) and 40 repositories whose size, as GitHub reports it, is at least 400,000 KB. There were no forks or archived repositories. From the remaining 510 we drew 70 with a fixed seed (20261023). Of the six corpus repositories not excluded, two are not in the frame (Humanfia's and alphaproof-nexus) and four could be drawn. Only cedar-spec was drawn, and its results in the sample match those in the corpus. Packages pinning Lean older than 4.27 are not supported and were skipped (14 packages in 9 repositories). The others were run with the same driver and limits, except one repository,

which our scripts did not run because they did not recognize its toolchain name (three uses of `native_decide`). Native facts that build occur in 26 packages, 2,018 facts in all. In the final run the per-package median is 99.6% and the mean 70.3%; the pooled rate is 58.5%. Our own package dominates the pooled rate, because 420 of its 534 facts stop at a resource limit; without it the rate is 75.9%. Of the 838 failures, 509 hit memory, time, stack or recursion-depth limits, and 420 of these are in our own package. The traced ones reach the LRAT parser of `bv_decide` (203, in VeIR, Radix and tacenta), `partial` functions and while loops (70), iterator-based string functions (25) and `WellFounded.fix` (17). The 25 iterator-string facts also reach partial functions, which have no equations in the logic, such as Lean's JSON parser. The retries of Section 3.5 convert all 14 facts of ninegates, which reach the platform word size, and the four facts of Zag and minidregg whose copied definitions occur in types; BobDyLean's 13 such facts stay blocked. Lean-regex's two facts, which the baseline run had among the iterator-string failures, now run out of memory, and a package add-on converts them (Section 4.7). One fact, in a test module for compiled overrides, evaluates to `false` in the kernel; NativeTrust reports it and produces no proof.

4.6 Robustness to core PR #14892

Among the evaluations in the final run, core PR #14892 can affect only those that run well-founded recursion. The converter made well-founded copies for 1,775 converted facts. Each of these facts copies definitions of its own project, and 71 also copy definitions of core or Mathlib (29 and 13 distinct definitions), such as `String.splitOn`, `String.trimAscii`, `List.merge`, `List.mergeSort` and `Array.qsort`. Of the 1,775 facts, 28 evaluate the recursion: the 26 of the ablation (Section 4.4), one BerryEsseen fact that converts only in the heavy pass, and one rs-mca fact of the per-theorem driver, which the ablation does not cover; each of the last two fails without the replacement. Without the proof transport of the retry (Section 3.4), congruence cannot bridge the copies of 15 formal-conjectures facts, which would then convert only through the accessibility route. With it, 51,824 of the 51,852 conversions never evaluate a definition built on `WellFounded.fix`, and the other 28 do so on the fuel route; none uses the accessibility route.

The corpus packages pin Lean 4.28–4.35 and cannot be built on a nightly. On the nightly of 22 October 2026, which contains PR #14892, the fuel route converts `List.mergeSort` and a lexicographic Ackermann function. On that nightly the accessibility route fails on `List.mergeSort`, tested by itself in a file of its own, where it gets stuck at `Classical.choice`; on Lean 4.34 it succeeds. For comparison, plain `decide +kernel` fails on both toolchains.

4.7 Options and add-ons

We ran the options and add-ons of Section 3.6, with the final library, on the facts that the final run does not convert. Table 3 shows their combined effect (a) and the facts that each of them converts (b). With all of them, and counting BerryEsseen's runs beyond the time limit, 52,599 of the corpus's 52,821 facts convert (99.6%); without the three largest packages, 92.8% convert.

Most of what the options achieve comes from `csimp`. With `stage`, `csimp` and `propSplit` together and up to 16 parts per check, 108 facts convert, and `csimp` alone converts 105 of them, among them clean's 99 primality facts, through the equation that Mathlib proves for its compiled least-factor search. Mathlib's binomial coefficient needs a better formula than Pascal's rule. A package add-on with a fast binomial coefficient converts 18 of longfellow-zk's 21 binomial facts, and the `@[csimp]` add-on converts 15 of these 18 through Mathlib's proved product formula. The other three convert with `csimp` together with `propSplit` and 16 parts. Given 3,000 s instead of 900 s, the options convert three more TrainVerify facts; the heavy pass or sharding also converts each of them. Staging

(a) Share of the facts converted as groups of extensions are added from left to right

	baseline	final run	general add-ons	options	package add-ons
corpus, pooled	98.0%	98.2%	98.2%	98.4%	99.6%
corpus without the three largest	65.9%	68.5%	69.2%	73.0%	92.8%
corpus, per-package median	90.8%	98.4%	98.5%	99.0%	99.5%
random sample, pooled	44.9%	58.5%	67.9%	68.1%	69.0%
random sample, per-package median	77.8%	99.6%	100.0%	100.0%	100.0%

(b) Facts converted by each extension

	extension	facts	packages
general add-ons	`bv_decide` pipeline	196	VeIR 188, clean 4, tacenta 3, asi-stack-book 1
	`@[csimp]` add-on	16	longfellow-zk 15, formal-conjectures 1
options	`propSplit`	4	rimalloc 3, Humanfia 1
	`stage`, `csimp`, `propSplit`, up to 16 parts	108	clean 99, formal-conjectures 5, longfellow-zk 3, goya 1
	`shards`	15	BerryEsseen 8, longfellow-zk 4, TrainVerify 3
package add-ons	tokenizer replacement	307	BerryEsseen
	covers parsed once per certificate	259	BerryEsseen
	primality proofs by `norm_num`	99	clean
	hash-map lookups as array scans	34	Strata
	fast binomial coefficient	18	longfellow-zk
	`DecidableEq` instance	12	BobDyLean
	list version of a graph linearization	4	minidregg
	Pratt primality certificates	3	formal-conjectures
	linear checks of a case-fold table	2	lean-regex

Table 3: Opt-in extensions, run on the facts of the corpus and the random sample that the final run does not convert. In (a) the final run includes the heavy pass (3,600 s, 96 GiB). The add-ons and options ran at 900 s and 48 GiB per process, sharded checks at these limits per module; many of BerryEsseen's package add-on runs took longer, and two VeIR facts converted by the `bv_decide` pipeline as modules of 3,600 s each. In (b) some facts convert with more than one extension; (a) counts each fact once.

alone converts none of these facts, and on two facts each of aks and sorting-network it stops the process, because compiled code for one of their externs is not available to it. Nor does it reach the large computations of Section 4.3: in most of the 80 facts of the baseline run that compare large computed values, the staged parts are themselves too expensive.

Sharding splits BerryEsseen's eight chunks of exact rational checks into their 176 conjuncts each. It splits four longfellow-zk certificates and three TrainVerify facts into at most 64 modules each (33 to 64 in these runs). Each combined theorem depends only on the standard axioms, checked in a fresh process. Sharding does not help facts whose first part already exhausts memory, such as BerryEsseen's covers and our own package's boxes: in each, all 8 sampled facts failed in their first part, and the other 970 such facts were not sharded. It also cannot use a native fact nested inside the checked one (12 minidregg facts).

For BerryEsseen, two package add-ons, a replacement for the tokenizer and one that parses the covers once per certificate, convert all 566 facts that are left after the final run and sharding. In all, they converted 569 facts (310 with the tokenizer replacement and 259 covers), three of which the final run also converts. These conversions took 740.5 CPU-hours, and no process needed more than 42.4 GiB, but most ran longer than the time limit: 240 of the 310 tokenizer conversions took longer than 900 s, 144 of them more than an hour, and 255 of the 259 covers took longer than 900 s along their longest chain of parts compiled in parallel (median 1,073 s). All the parts of a cover took a median of 7,144 s together.

5 Removing unnecessary `native_decide`

The ablation (Section 4.4) leads directly to a tool. Importing `NativeTrust.KernelFirst` registers elaborators for `native_decide` and `decide +native` that Lean tries before its built-in ones. Each call first runs the static pass and, if no blocking constant is reachable, builds the same auxiliary lemma as `decide +kernel` and lets the kernel check it within a heartbeat budget (2,000 in `maxHeartbeats` units by default). If the kernel proves the goal, that is the proof, with no native axiom. Otherwise the state is restored and Lean's native path runs, so the call behaves as before, and an info message gives the reason. No call is edited. The kernel attempt adds only a declaration that Lean's kernel checks.

On unmodified sources, adding only this import removed every native axiom from lido-srv3 (723 modules, 70 to 0) and from cvc5/logos's 10 largest modules (1,043 to 0). Building took about the same time with and without the import: 343 s and 345 s for lido-srv3, and 187.6 s and 184.7 s for the cvc5/logos modules. Where kernel checks are expensive it costs more: codegraff, a small package from the random sample, took 17 s instead of 2.6 s. The module imports Lean, and through it Lean's `Std` library, into every importer. In codegraff, six structure literals written in the shorthand `{ a, b }` were then elaborated as set notation, because `Std` has generic `Insert` and `Singleton` instances; importing Batteries, Aesop or Mathlib has the same effect, and a type ascription fixes each literal. For projects that cannot take that import, `NativeTrust.KernelFirstLite` is the same switch as two macros that import nothing beyond Lean's prelude, without the static pass, options or reports: codegraff builds unchanged with it (88 to 23 native axioms), and lido-srv3 loses the same 70 axioms. Its build then takes 421 s instead of the 360 s of that run's unmodified build (343 s in the run of the full switch), because a call that the kernel cannot check spends the budget before it falls back.

For projects that want no dependency on NativeTrust, a source rewriter (`ntrust migrate`) replaces a `native_decide` by `decide +kernel` when plain kernel evaluation proves all its facts within a per-fact budget (2,000, as for the switch, by default), so expensive facts stay native. Unlike

package	native axioms before	after	defaults build wall time (s)	after	without limits build wall time (s)
lido-srv3	70	0	7.1 → 7.8	0	7.8 → 15.6
cvc5/logos	2,185	3	1,740 → 1,693	1	1,855 → 1,717
Humanfia IChO 2026	1,864	194	93.6 → 95.3	145	94 → 470
asi-stack-book	392	10	22.2 → 21.9	10	29.7 → 28.7
sounio	77[a]	35	4.3 → 4.2	7	6.8 → 205
formal-conjectures	690	126	31.6 → 33.9	111	45 → 211
total	5,278	368		274	

Table 4: Rewriting `native_decide` as `decide +kernel` (`ntrust migrate`) in six corpus packages, with the default limits (a per-fact budget of 2,000 and a slowdown limit of 2) and without them, in separate runs; both runs also limit the time of a build round. Build times are clean rebuilds of the affected modules, with the same job count before and after. [a]Not counting 664 uses in modules that the original build does not build.

	native axioms removed	uses replaced	builds	build time
`ntrust migrate`, defaults	30.1% / 25.0%	28.1% / 47.2%	23 of 24[a]	0.99 / 1.00
without budget or slowdown limit	53.6% / 75.9%	50.0% / 80.7%	23 of 24[a]	5.4 / 1.9
one-import switch, budget 2,000	35.5% / 27.3%		21 of 24[b]	1.3 / 1.2
budget 20,000 where needed	48.4% / 39.8%			

Table 5: Removing native axioms from the random sample: 24 packages (all with native facts except the two of our repository), 1,472 native axioms and 2,361 uses, with an earlier build (Appendix B). Pooled rate / per-package median; build time is the ratio of clean-build wall time after to before, pooled / median. [a]The 24th, zcash, fails only in its own checks that some theorems keep a native axiom. [b]zcash and minidregg fail only in such checks, and codegraff builds with `KernelFirstLite`.

AXLE's tool for single files (Section 9), `ntrust migrate` works on a whole project: it rewrites the sources in place, keeping the originals, and rebuilds the project. It reverts each replacement that fails, that leaves a module building past a round's time limit (900 s by default), or that makes a module's build more than twice as slow, and then checks the remaining axioms. The per-fact budget and the slowdown limit can be switched off, but the time limit cannot. Table 4 shows six corpus packages with the default limits and without them. Under the default limits the rewriter replaced 2,058 of the 2,339 uses, including those in examples, and without the limits it replaced 2,220; with the heartbeat limits of four declarations lifted as well, Humanfia goes from 1,864 native axioms to 4. Every rewritten package builds, and the native axioms that remain are exactly those of the facts whose uses were kept. Without the limits, build time fell on cvc5/logos, where compiling the decision procedures for `native_decide` cost more than the kernel checks. It was level on asi-stack-book and rose where single facts are expensive to evaluate in the kernel, about thirtyfold in sounio. With the defaults no build took more than 1.1 times as long.

To measure this without the selection of the corpus, we ran both tools on every package of the random sample except ours (Table 5). Run without its limits, the rewriter leaves 683 native axioms, and 208 of them come from `bv_decide`, which a rewrite to `decide +kernel` cannot handle (they convert by the pipeline of Section 7). Among the rest, 120 belong to facts too expensive for plain kernel evaluation within the per-fact limits, 101 to rewrites that were reverted because their modules took too long or too much memory to build, 80 to modules whose checks require the native

axioms, and 174 to other causes, mostly constructs that plain kernel evaluation cannot handle, chiefly iterator-based strings and well-founded copies. Six of the 24 packages check axiom footprints in their sources, and four require some theorems to keep native axioms (one of them only `bv_decide` axioms), for instance with `#guard_msgs in #print axioms` or with a command that fails when a theorem marked as native "reaches no native_decide axiom". In those projects removing a native axiom is a change their tests reject, and the variants built without the audit modules all succeed. Without the limits, one small package took 143.5 times as long to build (1.34 to 192.29 s), because its rewritten facts are expensive in the kernel. With the defaults no package's build took more than 1.7 times as long; the slowest went from 1.03 to 1.74 s. The budget keeps expensive facts native, and the slowdown limit reverted modules in six packages where many cheap facts added up. The one-import switch fails on the two packages with axiom checks and on codegraff.

For new code, the linter (`set_option linter.nativeTrust true`) checks each new native fact the same way. When the static pass finds a reachable blocking constant, it warns about it, although we have not measured how often such warnings are false. Otherwise, when the kernel proves the fact within the budget, it warns that `native_decide` is unnecessary, with a "Try this: `decide +kernel`" fix. A strict option makes the second warning an error for continuous integration, and `#native_decide_audit` prints a verdict for every native fact of a module. On cvc5/logos and lido-srv3 modules the linter added at most 4.4% to elaboration time.

6 Independent checks

Method. The supplements and conversions checked in this section were made with an earlier build (Appendix B). We export kernel theorems with their full dependency closure, using `lean4export` [25] built for the package's toolchain. We then run two checkers distributed with the Lean 4.35 toolchain: nanoda 0.4.19 [26], a separate implementation of the kernel, with the three standard axioms permitted and any other a hard error; and con-ron, which never reduces proofs. An empty export counts as a failure. We report inputs that con-ron does not support as declined. Neither checker compares T.`kernel` with T, so for the supplements we also run a separate statement check. For every T.`kernel` in a supplement module, a fresh Lean process compares its type with T's by expression equality; when they differ, it adds a theorem with T's type and the value T.`kernel`, which Lean's kernel accepts only if the two types are definitionally equal. It also lists any axioms beyond the standard three, and as a positive control it confirms that each original T depends on a non-standard axiom. We did not, however, run Comparator.

Supplements. We ran `lake exe ntrust` on cvc5/logos, lido-srv3 and Humanfia, and checked every module of each supplement separately, because some packages' modules cannot be imported together. Table 6 gives the results. In all three, every T.`kernel` uses only the standard axioms and every original uses a native axiom. The two modules con-ron declined exceed its expression arena (cvc5/logos) or use the `Nat.div` and `Nat.mod` definitions of Lean 4.31, which it does not support (lido-srv3). One Humanfia module ran con-ron out of address space at 64 GiB and was accepted at 200 GiB.

Samples. For twelve packages, including the three above, we proved a random sample of converted facts again (seed 20261023) and checked them the same way (Table 7). The sample sizes were chosen per package to keep each export within the checkers' memory. Both checkers accepted every sample they could read. The checker con-ron declined two packages, for the same two reasons as for the supplements. The two checkers count declarations differently, and we report the counts as printed.

package	facts	theorems	modules	same / defeq statement	nanoda	con-ron
cvc5/logos	2,185	1,115	205	1,115 / 0	205	204 (1 declined)
lido-srv3	70	84	9	84 / 0	9	8 (1 declined)
Humanfia IChO 2026	1,860	395	34	375 / 20	34	34

Table 6: Kernel supplements (Section 3.7). Facts converted; kernel theorems produced; supplement modules; statements expression-identical / only definitionally equal (all 20 confirmed by the kernel); modules accepted. Humanfia has 407 theorems with native facts; 12 depend on its 4 unconverted facts. Producing the supplements took 25 minutes (cvc5/logos, 24 jobs), 15 s (lido-srv3, 8 jobs) and 24 minutes (Humanfia, 16 jobs); the independent checks of the cvc5/logos supplement took 70 minutes on 8 jobs.

package	sample	nanoda (declarations)	con-ron (declarations)
cvc5/logos	40	5,455	4,749
Humanfia IChO 2026	40	7,435	6,368
Strata	20	13,868	12,359
lido-srv3	30	1,847	1,412
cedar-spec	10	719	632
rs-mca (m31)	9	663	548
asi-stack-book	30	2,613	declined
sounio	30	1,953	declined
longfellow-zk	30	4,920	4,385
formal-conjectures	30	4,839	4,367
clean (with the four primality proofs)	30	4,249	3,796
TrainVerify (one fact from each of 24 module groups)	24	24 groups accepted	24 groups accepted

Table 7: Independent checks of random samples of converted facts, made with the build used for the supplements (Appendix B). The cvc5/logos exports contain copies built on `fixFuel` and the lemma `fix_eq_fixFuel`.

7 Kernel-checked `bv_decide` facts

A `bv_decide` fact states that an LRAT certificate, stored as a string, checks against the CNF of a bit-blasted goal. The certificate parser uses `partial` functions and the CNF construction hashes with opaque constants, so the kernel cannot evaluate these facts as they stand. We keep the user's theorem and replace the native fact inside its proof by a kernel-checked proof that the bit-vector expression is unsatisfiable (core's `BVLogicalExpr.Unsat`), which is what core's reflection lemma derives from the native fact. As with the supplements, the new theorem is declared with the original theorem's type.

Core's bit-blaster does not suit kernel evaluation: its and-inverter graph is built with arrays and hash maps, which the kernel evaluates as lists. On a 16-bit goal with 312 graph nodes, bit-blasting alone took 34 of the 69 s that core's pipeline needed to build the CNF. We wrote a bit-blaster for the kernel instead. It works on lists by structural recursion, builds each repeated word operation once (so x/y and x mod y share one divider), and emits Tseitin clauses over node indices. Its soundness theorem states that if this CNF is unsatisfiable, so is the expression; it covers every operation of `bv_decide`'s expression language, including bit reversal, counting leading zeros and population count.

An external SAT solver (CaDiCaL) produces an LRAT certificate for this CNF, which is stored

as definitions and is not trusted. The original and the learned clauses sit in one balanced tree. The kernel first checks the original clauses against the bit-blaster in an in-order pass, and then it checks the certificate's steps. For large goals the first pass is split: one declaration runs the bit-blaster, one per segment of 30,000 graph nodes checks well-formedness and agreement, and proved lemmas join them. Each clause and the partial assignment are pairs of natural-number bitmasks, so unit propagation uses `Nat.land`, `Nat.lor`, `Nat.xor`, shifts and subtraction, which the kernel evaluates with GMP. Each step carries its unit literal, computed outside the kernel and checked with a subset test, because `Nat.log2` is not accelerated. The steps are checked in chunks of at most 20,000 hints, one declaration each, and the chunks can go into separate modules compiled in parallel. The soundness proofs use core's definitions and lemmas; they depend only on the standard axioms (checked on Lean 4.31, 4.33 and 4.35).

All five `bv_decide` facts of the corpus convert (Table 8). The largest, clean's ByteDecomposition `soundness`, has 65,720 clauses and a certificate with 959,442 hints; its kernel checks take 247 s (261 s with solving and preparation) in a process that peaks at 7.8 GiB. Split into four modules compiled in parallel, they take 113 s of wall time. We also ran every `bv_decide` fact of the corpus and the random sample (Section 4.5), and all 213 convert. They come from six packages, 188 of them from VeIR and 17 from Radix. In the final run, with the final library loaded, the 196 facts outside Radix converted again, one process per fact at the limits of Section 4.1, except six VeIR facts. Two of these ran out of time in one process and convert as four modules compiled in parallel (each limited to 3,600 s and 48 GiB); the other four ran out of memory and convert with the split first pass. Radix pins Lean 4.29, which the current version of the pipeline does not support. Its 17 facts convert there with two earlier versions: the one used for this sweep, whose soundness proof built on Lean 4.29 after a small change and uses only the standard axioms, and one that evaluates core's bit-blaster in the kernel. A fresh process confirms that each converted fact depends only on the standard axioms, and each rebuilt declaration has the statement of the original. Where theorems use several facts converted in different modules or packages, those facts are converted together, in one module, and after these joint runs every rebuilt declaration depends only on the standard axioms. The checker nanoda accepts a sample of them, including VeIR's `Veir.RISCV.Combine` rebuilt over 30 converted facts (29,187 declarations), and rejects the native originals. A `bv_decide` fact inside a definition yields a rebuilt definition without the axiom, but statements that mention the original definition, rather than the rebuilt one, still depend on it.

On synthetic goals the first pass costs about 0.5 ms per clause, and the steps cost about 0.22–0.25 ms per hint on the multiplication and division goals of Table 8, where steps carry many hints. With the first pass split, a goal with 1.18 million clauses converts at 20.4 GiB; the declaration that runs the bit-blaster is then the limit, at about a million graph nodes by extrapolation. Certificates of about ten million hints take 30–40 minutes; larger ones exhaust memory in the elaborator, which holds the parsed certificate, before the kernel starts.

A drop-in tactic. The same pipeline is also a tactic, `bv_kdecide`, that can be written instead of `bv_decide`. It runs `bv_decide` on a copy of the goal in a scope whose changes to the environment are discarded, takes the reflected expression from the resulting proof term, proves its unsatisfiability as above and closes the goal with core's reflection lemma, without adding a native axiom. Of the 208 `bv_decide` axioms that the rewriter left in the random sample (Section 5), 190 convert this way. Of the other 18, one VeIR axiom is proved by the project's own tactic, and Radix's 17 axioms stay because of its Lean 4.29 pin. VeIR calls `bv_decide` through a macro of its own, at 247 call sites; changing that macro alone made one proof module build with 68 kernel-checked uses and no native axiom. On 195 facts of four packages the tactic took a median of 2.0 s per fact (90th percentile 53 s,

goal	clauses	hints	kernel time	peak memory
clean `not64_eq_sub`	2	2	0.05 s	2.6 GiB
clean `And8` / `Or8`	1,187 / 1,205	7,503 / 5,319	1.7 / 1.5 s	2.9 GiB
asi-stack-book	1,424	840	0.8 s	1.6 GiB
`admitted_layer_contract_is_complete`				
clean ByteDecomposition `soundness`	65,720	959,442	247 s	7.8 GiB
$(x \,\&\, y) + (x \mid y) = x + y$, 1,024 bits	73,697	232,083	191 s	
8,192 bits	589,793	1,863,909	2,054 s	47.9 GiB
$x \cdot y = y \cdot x$, 9 bits	3,161	5,865,891	1,352 s	
$x \cdot (y \cdot z) = (x \cdot y) \cdot z$, 7 bits	3,587	9,040,986	2,006 s	
$(x/y) \cdot y + x \bmod y = x$, 12 bits	8,909	5,639,962	1,380 s	

Table 8: Kernel-checked `bv_decide` goals: the corpus's five facts, then synthetic goals. Kernel time is the first pass plus the certificate steps (single first-pass declaration); one core, on a shared machine. Peak memory is the maximum resident size of the process and is given for the rows where the process checked one goal.

at most 48 minutes and 16.8 GiB). A limit of 60 s per fact would let a rewrite keep 176 of the 195, for 1,018 s of added build time.

8 Comparison with hand-built kernel versions

Many of our published results use `native_decide`, and for most of them we later built kernel-checked versions by hand: we rewrote the decision procedures for the kernel, generated check modules from the certificates and built the modules in shards. By their build records, the hand-built versions of 23 results took about 1.03 million CPU-hours; nineteen of them prove the published bounds, and four prove only a weaker bound or a part (the artifact lists them with their sources). We ran the converter on the native versions, at the limits of Section 4.1 and again at 3,600 s and 96 GiB for the facts that ran out of time or memory, and compared the outcome with the hand-built versions (Table 9 in Appendix C). Where a class of facts was too large to run in full, we ran a sample, and we count the facts of a sampled class that we did not run as unconverted.

A light theorem. Theorem 1.1 of the quantum error-correction paper [32] has 61 native facts, and all of them convert. On the same machine and toolchain, the native checks of the theorem took 0.07 CPU-hours, its hand-written kernel version took 1.61 and the conversion took 1.68. The hand-written version evaluates the same checking functions with `decide +kernel`, through 390 check modules and their aggregator, written by two generators; the conversion needs none of these files. The converted theorem has the original's statement and uses only the standard axioms, and nanoda checked the 21,775 declarations of its export; con-ron, which had accepted an earlier conversion of the theorem, aborted on this export after a failed memory allocation.

A heavy proof. Our proof that sorting networks on 13 inputs need at least 45 comparators, $S(13) \geq 45$, rests on 1,724 native facts, and none converted in our 22 probe runs. Its checker keeps memo tables in `Std.HashMap`s keyed by byte arrays, whose hash functions are C primitives without a definition, and the static pass, run on one fact of each class, finds such blocking constants in the classes that hold 1,684 of the facts. Every fact also reads its certificate data from string literals of up to 64 MiB. Even with a 32 GiB stack, the kernel rejected the copies of one fact's fifteen 64 MiB

chunks for recursion depth, while it accepted its last chunk, of 4.8 MiB, in 18 s. Small inputs are expensive too: two test certificates of under 1 KB take 79 and 99 s in the kernel, where the native module with all six test checks builds in 1.1 s. The hand-built release rewrote the checker over tables of natural-number literals, and a fresh build of it took 8,805.9 CPU-hours over 102,624 modules [31]. The search facts of the quantum error-correction paper's other theorems behave alike. A single head did not finish within an hour, whereas the hand-built version, which spends 662.3 of its 715.6 CPU-hours on the heads, checks one in 4.3 s.

So the conversion replaced a hand-built version where the native checking functions were already within the kernel's reach, in about the hand-built version's CPU time. Facts that search large spaces or read large data still need checkers written for the kernel, and most of the cost of the hand-built versions lies there. A probe tells the two apart: for $S(13) \geq 45$, the static pass and the probe runs took 2.2 CPU-hours.

9 Related work

Proof by reflection [5] underlies `decide`. Coq evaluates by compiled code inside the kernel's conversion check (`vm_compute` [20], `native_compute` [4]); proofs that use `native_compute` trust the OCaml compiler in a similar way, and Coq's kernel was given machine integers and persistent arrays for checking SAT certificates by reflection [37]. HOL4 derives computation theorems by rewriting (`computeLib` [3]). Isabelle's normalization by evaluation [1] runs compiled ML code and admits its results through an oracle, and Isabelle distinguishes evaluation by the simplifier (`code_simp`), which produces theorems, from evaluation by generated code (`eval`), which is trusted [22, 48]. Candle adds a verified in-logic computation primitive [7]. The four-color theorem [19] was checked by Coq's kernel, while Flyspeck's enumeration of tame graphs [27] ran through Isabelle's code generator [23]. Primality certificates checked in a proof assistant go back to Caprotti and Oostdijk [8] and Grégoire, Théry and Werner [21]; Isabelle's Archive of Formal Proofs has Pratt certificates [16], and Mathlib's `norm_num` decides the primality of small numbers. General recursion in type theory has a long literature, including the Bove–Capretta method [6] and Equations [29]; our fuel lemma follows the standard pattern of unfolding a fixpoint a bounded number of times.

SAT certificates. Verified checking of SAT and SMT certificates (Cruz-Filipe et al. [17], Lammich [24], cake_lpr [30], SMTCoq [2]) is the model for Section 7. The tactic `bv_decide` runs its verified LRAT checker as compiled code [39], and so does LRAT-Catcher, which imports large certificates into Lean in chunks [41]. Mathlib's `lrat_proof` [40] has Lean's kernel check an LRAT proof through a proof term built step by step. Two recent prototypes evaluate a checker in Lean's kernel. LratK [42] is an RUP checker with a bitset assignment and a soundness proof. The prototype lean-dedep [43] translates `bv_decide`'s whole pipeline into a non-dependent program with a generated proof relating the two; its tactic `in_kernel bv_decide` proves $x \cdot 3 = x + x + x$ for 6-bit x in 48 s, and the 16-bit case does not finish within ten minutes. Our pipeline differs in its bit-blaster written for the kernel, its clauses stored as bitmask pairs with unit witnesses, and its chunks in separate modules. On the same goal, with the same settings (one thread, a 40 GiB address-space limit) but on a different machine, it takes 1.1 s for the whole file at 6 bits, 2.3 s at 16 bits, 12 s at 64 bits and 651 s at 1,024 bits.

Lean. Core Lean 4 [18] offers `decide +kernel`, the fuel-based `WellFounded.Nat.fix`, and `cbv`/`decide_cbv`, which rewrite with equation lemmas and therefore also handle well-founded definitions, more slowly. A core issue (#5806 [15]) plans to rewrite recursive functions into `Nat.rec`

with large fuel, and Mathlib has fuel rewrites for individual functions. The prototype lean-dedep [43] also proves, by the same induction from `WellFounded.fix_eq` as our fuel lemma, that a well-founded fixpoint whose relation is bounded by a natural-number measure equals `WellFounded.Nat.fix`; our lemma needs no measure, because after 2^{64} unfoldings it falls back to `WellFounded.fix`. Parth Shastri proposed a lazily generated accessibility proof inside `WellFounded.fix` itself (PR #9944 [14], not merged). For removing the tactic, ReplaceNativeDecide [44] is a prompt for AI agents that replaces individual uses with conventional proofs, and the `repair_proofs` tool of AXLE [38] replaces `native_decide` by `decide +kernel` within a submitted file, keeping the original where the kernel fails. Mathlib's linter [46] flags every use of the tactic. The tool lean4checker [11] re-checks environments with Lean's own kernel; nanoda [26] and con-ron [45] are independent implementations, and Lean4Lean [9] reimplements the kernel in Lean, closely following the C++ one. SafeVerify [12], Comparator [10] and `lake check` are gates that refuse native axioms. We know of no other tool that converts the existing native facts of whole projects without editing them, or of an earlier measurement of what prevents conversion.

10 Discussion

For authors of verified software. Where the kernel proves a fact, `decide +kernel` is the better choice. The constructs in Table 2 make facts unconvertible with current core: code that goes through the metaprogramming layer, transcendental `Float` functions and other primitives that exist only as compiled code, and the certificate parser of `bv_decide`. The platform word size behind `USize` and `HashMap` blocks only results that depend on it (Section 3.5). Data stored as `String` and parsed at check time no longer blocks evaluation, but the parse can cost more memory than the check is worth. Decidability instances obtained by a cast, such as `simp <;> infer_instance`, block kernel evaluation; `decidable_of_iff` avoids the cast, since its proof is never evaluated. Native facts inside instances should carry certificates.

For Lean core. Defining `List.toByteArray` by the one-step version that copies use (Section 3.2) would make kernel evaluation of string literals linear in core itself. Linear evaluation is not enough for very long literals, which unfold to lists of characters as deep as the literals are long: the kernel ran out of recursion depth on the 64 MiB literals of Section 8. Lemmas that characterize `String.replace` and the other pattern-generic string functions by lists would let proved replacements reach every pattern type, and iterator state types the kernel can see through would make iterator-based string functions convertible without replacements. Logical definitions for the hash primitives would unblock `HashMap` keyed by strings, and arrays that the kernel evaluates without going through lists would let core's `bv_decide` pipeline run in the kernel and make arrays built by repeated `push` cheap to build. None of these changes reaches Strata's `partial` parser, unsafe implementations and `partial_fixpoint`.

Limitations.

- The corpus is selected, and the random sample comes from a code-search frame (Section 4.5). Neither counts facts in modules that did not build or that name no native tactic, such as uses through a project's macro.
- The retries, options and add-ons were developed against the failures of the baseline run, in the corpus and in the random sample, so the final run measures them on the packages they were built for, not on unseen ones.

- The time and memory limits are fixed, and heavy facts may convert with more resources; the options and add-ons ran without a heavy pass (Table 3).
- Supplements do not rebuild theorems of modules that name no native tactic (Section 3.7).
- The independent checks of Section 6 were made with earlier builds and do not cover every conversion of the final run.
- The source rewriter does not rewrite uses inside macro definitions, and neither removal tool handles `bv_decide` (Section 7).
- The evidence for robustness to PR #14892 is indirect: no package of the corpus could be built on a toolchain that contains it (Section 4.6).

11 Conclusion

Most native facts in the packages we studied never needed compiled evaluation: Lean's kernel re-proves them as they are, so `decide +kernel` would have served for them, including the facts inside definitions. Proved replacements and cast-free instances, which add nothing to what a proof trusts, emptied the corpus's classes of iterator-based string functions and of casts, and what still fails is mostly computation that is large for the kernel. The changes to core in Section 10 would bring more of it within reach; for the heaviest computations, as on our published results, checkers written for the kernel remain necessary. A probe with the converter is cheap, so a project can learn which of its facts need such checkers before writing any.

A Repositories

package	repository and commit	status
TrainVerify	`Tianshi-Xu/tainverify_lean` 9cfbfc78048e	studied
cvc5/logos	`cvc5/logos` 4cd3a10e302a	studied
Humanfia IChO 2026	`humanfia/hoa` 1c918ec9a96c	studied
formal-conjectures	`google-deepmind/formal-conjectures` c83a94d785d0	studied
longfellow-zk	`longfellow-zk/longfellow-zk` de5f9490c3d8	studied
BerryEsseen	`CoolRmal/BerryEsseen` bfb22608ad18	studied
asi-stack-book	`corbensorenson/asi-stack-book` 5ae6da1ef0b6	studied
Strata	`strata-org/Strata` 649cba18f6f7	studied
clean	`Verified-zkEVM/clean` 1a6c28426372	studied
rs-mca	`przchojecki/rs-mca` 93fba1be3f32	studied
sounio	`Sounio-lang/sounio` 3ef3dbaa6e94	studied
lido-srv3	`lfglabs-dev/lido-srv3-proof-closure` 2a182aa1d9fa	studied
cedar-spec	`cedar-policy/cedar-spec` c9a03407a96a	studied
aks	`girving/aks` c7fb62ed80a4	studied
sorting-network	`VarunVejalla/sorting-network` aad4ffcef720	studied
aeneas	`AeneasVerif/aeneas` 74aabd52f770	studied
FX	`GrigoryEvko/FX` e01b6e40bb6c	no native facts
evm-asm	`Verified-zkEVM/evm-asm` 7e65e4d02471	no native facts
brill-noether	`npflueger/brill-noether-graphs` 68c6891b0a99	no native facts
tengoku	`competemath/tengoku` 1487abe2fe69	no native facts
alphaproof-nexus	`google-deepmind/alphaproof-nexus-results` 0647711a7118	no native facts
ternary-goldbach	`gersh/ternary-goldbach-lean` 27df23af6a71	none in built modules
program_guide	`lusing/program_guide` b5eadb6b4c8a	no native facts
sparkle	`Verilean/sparkle` 1614cee83ac5	no native facts
ArkLib	`Verified-zkEVM/ArkLib` d023c5d31d76	no native facts
UFT	`iarnoldy/UFT` a90ce55c5773	did not build
tetrasemaphore	`IT-Help-San-Diego/tetrasemaphore` 833c2438fc3e	did not build
veil	`verse-lab/veil` 02791b2d1576	did not build
FloatLib	`lean-dojo/FloatLib` 1f5771b3e36a	did not build
llm-rust-lean-pipeline	`tobiaskrogh-ship-it/llm-rust-lean-pipeline` 89b8ef6d0a0c	did not build

B Reproducibility

The library is available as a Lake package at https://github.com/tejasvi/native-trust. The artifact (version 1.1.0, doi:10.5281/zenodo.23818393) holds the library, the add-ons, the tests with their expected outputs, the drivers and aggregators, the per-fact results of every run cited here and the logs of the facts that did not convert, the supplements and checker outputs, the sample frame and the script that draws the sample (seed 20261023), and an index that gives, for each number in this paper, the files and the command that reproduce it.

Source manifests are SHA-256 digests over the per-file digests of the library and Lake files. The library in the artifact, `6b378a8e`, is the final library. It produced the final run of Section 4 with its options, add-ons, sharding, ablation and label check, the re-run of 196 `bv_decide` facts in Section 7, and the runs of Section 8. For the final run it was built for each of the 26 toolchains that the packages pin. On Lean 4.27–4.29, whose core lacks lemmas from which the string replacements are proved and keeps `@[csimp]` equations in an older table, the module with these replacements and the module that looks up `@[csimp]` equations are replaced by stubs, so the string replacements and the `csimp` option are not available there; of the packages with facts that the final run does not convert, this concerns aks, sorting-network and Radix (32 facts). In the final run the checks of a package ran in groups that shared a memory limit, with the same per-fact limits as the baseline

run; a check stopped by that shared limit rather than by its own limit was re-run alone, and only the re-run, which the artifact records, counts.

Earlier builds produced the baseline run, the supplements and the sampled re-checks of Section 6, and the measurements of Section 5. The build used for the supplements, the sampled re-checks and the baseline run's ablation and random sample has the fuel route without its retry; only its library modules were kept, so its manifest cannot be recomputed. In the baseline run, a later build with the retry ran cvc5/logos, Strata and formal-conjectures, and two older builds ran the other 13 corpus packages; the older ones differ from the first in the treatment of well-founded recursion and in the triage, the step that decides where the evaluation cap applies. In the baseline run, a process that the machine stopped for lack of memory counts as the fact's memory failure. The trial of string replacements in the first attempt (Section 4.3) comes from a work build that was not adopted. The artifact keeps the library sources of the builds described here, except the two older ones, under their manifests, with the runs that used them (`study/library_versions`).

C More on our published results

Table 9 lists the main results of Section 8. Our 3SUM/APSP result [35] behaves like Theorem 1.1 of the quantum error-correction paper (Section 8), from the other side: its kernel version had rewritten the checker, in a 163-line file that also proves the rewrite equal to the original, and the converter proves the 11 native facts about the original checker in about the same time (52 s against 42.8 s, as sums of per-module wall times). Our paper on range-criterion counterexamples to the PPT^2 conjecture [34] had no kernel version; 573 of the 998 native theorems that it cites now have kernel-checked versions. We also ran 97 facts from thirteen further results, nine of which have hand-built versions, and 3 converted. We did not run our proof of $S(14) \geq 50$, whose native build alone took 1,274.5 CPU-hours. For nine results we ran, on copies of the modules, the library's tactic `kdecide`, which proves a goal by the same procedure as the driver, with compiled code switched off.

result	native facts	facts converted	theorems converted	CPU-hours native	CPU-hours NativeTrust	CPU-hours hand-built
with a hand-built kernel version						
QEC threshold, Theorem 1.1[a]	61	61	1 of 1	0.07	1.68	1.61
QEC threshold, all theorems	634	77[b]	1 of 16	17.2	9.4[b]	715.6
sorting networks, $S(13) \geq 45$	1,724	0[c]	0 of 1	1,185	2.2[c]	8,805.9
six-body central configurations	546[d]	56	18 of 97	–	11.1	1,178.3
3SUM/APSP lower bounds	11	11	14 of 14	<0.01	0.01	0.01
Hamming-cube Poincaré constant	3	2	0 of 4	<0.01	1.26	0.33
without one						
PPT^2 counterexamples, core	751	540	543 of 770	0.70	20.1	–
PPT^2 counterexamples, rows	1,039[e]	271	304 of 1,219	4.11	27.0	–

Table 9: NativeTrust on our published results that use `native_decide`. The native column counts the native check modules alone for Theorem 1.1, 3SUM/APSP and the Hamming cube, and otherwise the whole native build, as recorded in the release for $S(13) \geq 45$. Hand-built versions are as recorded in their releases; for the quantum error-correction paper, the kernel part of Theorems 1.2 to 8.1. [a]All columns measured on the same machine. [b]525 of the 541 search facts were not run after 16 sampled ones timed out; NativeTrust's CPU-hours cover the facts that were run. [c]22 probe runs, and the static pass on one fact of each class. [d]In the modules that build; the native headline theorem has 3,158. [e]679 run, those of the cited theorems first.

References

[1] K. Aehlig, F. Haftmann, T. Nipkow. A compiled implementation of normalisation by evaluation. *J. Funct. Program.* 22(1), 2012.

[2] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. CPP 2011.

[3] B. Barras. Programming and computing in HOL. TPHOLs 2000.

[4] M. Boespflug, M. Dénès, B. Grégoire. Full reduction at full throttle. CPP 2011.

[5] S. Boutin. Using reflection to build efficient and certified decision procedures. TACS 1997.

[6] A. Bove, V. Capretta. Modelling general recursion in type theory. *Math. Struct. Comput. Sci.* 15(4), 2005.

[7] O. Abrahamsson, M. O. Myreen. Fast, verified computation for Candle. ITP 2023.

[8] O. Caprotti, M. Oostdijk. Formal and efficient primality proofs by use of computer algebra oracles. *J. Symb. Comput.* 32(1–2), 2001.

[9] M. Carneiro. Lean4Lean: verifying a typechecker for Lean, in Lean. arXiv:2403.14064v3, 2025.

[10] Comparator. `https://github.com/leanprover/comparator` (archived; part of Lean since 4.35 as `lake comparator`).

[11] lean4checker. `https://github.com/leanprover/lean4checker` (archived; distributed with Lean as `leanchecker` since 4.28).

[12] SafeVerify. `https://github.com/GasStationManager/SafeVerify`.

[13] J. Breitner. Lean 4 pull request #14892, "define well-founded recursion without large elimination of `Acc`". `https://github.com/leanprover/lean4/pull/14892`, 2026.

[14] P. Shastri. Lean 4 pull request #9944, "experiment: definitional well-founded recursion" (not merged). `https://github.com/leanprover/lean4/pull/9944`, 2025.

[15] J. Breitner. Lean 4 issue #5806, "Optimizing terms for kernel reduction (meta issue)". `https://github.com/leanprover/lean4/issues/5806`, 2024.

[16] S. Wimmer, L. Noschinski. Pratt's primality certificates. *Archive of Formal Proofs*, 2013.

[17] L. Cruz-Filipe, M. J. H. Heule, W. A. Hunt, M. Kaufmann, P. Schneider-Kamp. Efficient certified RAT verification. CADE 2017.

[18] L. de Moura, S. Ullrich. The Lean 4 theorem prover and programming language. CADE 2021.

[19] G. Gonthier. Formal proof—the four-color theorem. *Notices of the AMS* 55(11), 2008.

[20] B. Grégoire, X. Leroy. A compiled implementation of strong reduction. ICFP 2002.

[21] B. Grégoire, L. Théry, B. Werner. A computational approach to Pocklington certificates in type theory. FLOPS 2006.

[22] F. Haftmann, T. Nipkow. Code generation via higher-order rewrite systems. FLOPS 2010.

[23] T. Hales et al. A formal proof of the Kepler conjecture. *Forum Math. Pi* 5, 2017.

[24] P. Lammich. Efficient verified (UN)SAT certificate checking. CADE 2017.

[25] lean4export. https://github.com/leanprover/lean4export.

[26] nanoda. https://github.com/ammkrn/nanoda_lib; version 0.4.19 as distributed with the Lean 4.35 toolchain.

[27] T. Nipkow, G. Bauer, P. Schultz. Flyspeck I: tame graphs. IJCAR 2006.

[28] V. R. Pratt. Every prime has a succinct certificate. *SIAM J. Comput.* 4(3), 1975.

[29] M. Sozeau, C. Mangin. Equations reloaded: high-level dependently-typed functional programming and proving in Coq. *Proc. ACM Program. Lang.* 3(ICFP), 2019.

[30] Y. K. Tan, M. J. H. Heule, M. O. Myreen. cake_lpr: verified propagation redundancy checking in CakeML. TACAS 2021.

[31] T. S. Tomar. A kernel-checked Lean proof that sorting networks on 13 inputs need 45 comparators. Zenodo, 2026. doi:10.5281/zenodo.23552145. Native Lean version: doi:10.5281/zenodo.23504361.

[32] T. S. Tomar. A matching threshold of 2×10^{-3} for the rotated surface code under circuit-level noise. Zenodo, 2026. doi:10.5281/zenodo.23498520. Native Lean version: doi:10.5281/zenodo.23485205; kernel-only Lean version: doi:10.5281/zenodo.23498518.

[33] T. S. Tomar. Smale's sixth problem for six bodies in the plane: finiteness of central configurations for generic masses, with a proof formalized in Lean. Zenodo, 2026. doi:10.5281/zenodo.23472600. Supplement: doi:10.5281/zenodo.23472605.

[34] T. S. Tomar. On the minimal dimension of range-criterion counterexamples to the PPT^2 conjecture. Zenodo, 2026. doi:10.5281/zenodo.23651994. Lean checks: doi:10.5281/zenodo.23651991.

[35] T. S. Tomar. Which fine-grained lower bounds survive the refutation of 3SUM and APSP? Zenodo, 2026. doi:10.5281/zenodo.23609699. Supplement: doi:10.5281/zenodo.23609700.

[36] T. S. Tomar. The L^1 Poincaré constant of the Hamming cube. Zenodo, 2026. doi:10.5281/zenodo.23533350. Supplement: doi:10.5281/zenodo.23533351.

[37] M. Armand, B. Grégoire, A. Spiwack, L. Théry. Extending Coq with imperative features and its application to SAT verification. ITP 2010.

[38] J. Xin, A. Schneidman, C. Cummins, K. Ram, S. Ganesh, J. Limperg. AXLE: a cloud infrastructure for Lean 4 theorem proving utilities. arXiv:2606.26442, 2026.

[39] H. Böving, S. Bhat, L. Cicolini, A. Keizer, L. Frenot, A. Mohamed, L. Stefanesco, H. Khan, J. Clune, C. Barrett, T. Grosser. Interactive bitvector reasoning using verified bit-blasting. *Proc. ACM Program. Lang.* 9(OOPSLA2), 2025.

[40] M. Carneiro. `Mathlib.Tactic.Sat.FromLRAT` (the `lrat_proof` command). Mathlib, 2022.

[41] S. Szeider. Streaming LRAT certificates into Lean theorems. arXiv:2607.00815, 2026.

[42] LratK, an LRAT checker evaluated by Lean's kernel (file `CoveringLean/LratK.lean`). https://github.com/thiagopatzdorf/Matematica, 2026.

[43] lean-dedep (proof of concept). https://github.com/nomeata/lean-dedep, 2026.

[44] ReplaceNativeDecide. https://github.com/GasStationManager/ReplaceNativeDecide.

[45] con-ron. https://github.com/leanprover/con-ron.

[46] Mathlib pull requests #25297, "lint against using `native_decide`" (2025), and #43194 (`linter.style.native`, 2026). https://github.com/leanprover-community/mathlib4/pull/25297.

[47] J. Breitner. Lean 4 pull request #12887, "optimize string literal equality simprocs for kernel efficiency". https://github.com/leanprover/lean4/pull/12887, 2026.

[48] F. Haftmann. Code generation from Isabelle/HOL theories. Isabelle documentation (tutorial `codegen`).