Leanexe's guarantees: kernel-checked theorems and their written assumptions

exaspec@morphism.com

27 November 2026

Abstract

Leanexe compiles programs written in a first-order subset of Lean and proves that the compiled code computes the Lean definition. Lean's kernel checks the proof for each program. The compiled code runs on AArch64 and x86-64 processors, on an Arm Cortex-M7 microcontroller, on GPUs, and in WebAssembly engines. This report gives a general account of what leanexe's theorems cover and what they trust. It compares leanexe's account of trust with those of CompCert, CakeML, seL4, and HACL*.

A leanexe guarantee is a theorem that Lean's kernel checks, together with a written list of what the theorem takes without proof, and the theorem says only what the definitions in its statement say. A statement that allows a trap holds of a module that always traps. A claim that a call returns therefore follows from the trap-free theorem, whose premises the caller proves for its inputs. Leanexe's proofs use the axioms `propext`, `Classical.choice`, and `Quot.sound`, and leanexe's checks fail on any other axiom.

No independent kernel has checked a leanexe proof. Leanexe's records show one replay of every module by `leanchecker`, a checker that uses Lean's own kernel, on 30 October 2026 under Lean 4.34.0-rc2. The C and bare-metal targets also depend on Rocq's kernel and on CompCert's proofs and axioms. Tests support the items that no proof covers. Leanexe's verification knowledge base counts a test as evidence only after a recorded run in which a seeded fault makes the test fail.

1 Introduction

For a verified compiler, a theorem states that the code it writes computes what the source program means [26]. A user of the theorem also relies on three things that the proof does not establish. They are the meaning of the statement, the items that the theorem takes without proof, and the program that checked the proof. Each of the three can make a correct proof say less than its user expects. A statement can hold of a program that never returns, an item taken without proof can fail on a real machine, and a defect of the checker can admit a false theorem.

Leanexe compiles programs written in a first-order subset of Lean and proves that the compiled code computes the Lean definition [28]. Leanexe writes AArch64 and Thumb-2 machine code, C for the CompCert compiler, WGSL and PTX kernels, and WebAssembly modules, including WASI commands (`README.md:3`). The verified compiler produces a WebAssembly module, which serves as the intermediate representation from which the C, AArch64, and Cortex-M7 translations start. WebAssembly engines also run the module as a final representation. Companion reports describe core leanexe [7], leanexe's targets [9], and its numerical library [8], and this report does not repeat them.

This report gives a general account of what leanexe's theorems cover and what they trust. A leanexe guarantee is a theorem that Lean's kernel checks, together with a written list of what the

theorem takes without proof, and the theorem says only what the definitions in its statement say. Leanexe's trusted-base document gives that list for each kind of artifact, with the check of each item and its source (`docs/trusted-base.md:3`). Table 1 of the companion report on leanexe's targets condenses the list for each target [9], and this report refers to that table.

The report describes the following parts of leanexe's account of trust.

1. The statement of a guarantee: the definitions that it names, the premises that a caller proves, and the statements that hold of every module (Section 2).
2. The three permitted axioms and the checks that reject any other (Section 3).
3. Lean's kernel, the one recorded replay of leanexe's modules, the absence of an independent kernel, and the Rocq proofs of the C and bare-metal targets (Section 4).
4. File theorems and file readers, which remove the program that writes a file from the trusted base (Section 5).
5. The tests that support trusted items, the seeded faults that each test must detect, and the theorems outside the full check, the script that runs leanexe's checks in order (Section 6).
6. A procedure for reading a leanexe guarantee, shown on leanexe's quicksort (Section 7).

This report describes leanexe at commit `4556fa84` of 26 November 2026, with Lean 4.34.1, Mathlib at `85e3a25` [32], the WebAssembly interpreter Talos at `87e3aa5e` [31], and CompCert 3.18 [3]. Every link to a leanexe file points to that commit. Where leanexe's documents at that commit contradict its sources or its history, this report states what the sources and the history show. The author read the statements, sources, and documents at that commit and did not rebuild the proofs or run the checks.

2 Guarantees as kernel-checked statements with written assumptions

A leanexe guarantee is a theorem that Lean's kernel checks, together with a written list of what the theorem takes without proof, and the theorem says only what the definitions in its statement say. The kernel's check establishes that the statement follows in Lean's logic from the axioms that the proof uses. The written list states what must hold for the statement to describe a running artifact. The definitions in the statement fix its meaning, so a reader learns what a guarantee says only by reading them. Figure 1 shows the parts of a guarantee and the support of each.

2.1 Generators outside the trusted base and items inside it

Leanexe's generators are the programs that write its programs, proofs, encodings, and translations. They run as meta code or as compiled Lean code, and the kernel checks what they produce. The trusted-base document lists as untrusted the reflector, the verified compiler `compileWith` with its choice of parameter modes, the native evaluation that proposes encodings and translations, the command `file_theorems`, and leanexe's verification knowledge base (`docs/trusted-base.md:15`). The reflector reads a Lean definition as a program of the compiler's source language, and `file_theorems` writes the theorems about an emitted file. A wrong output of a generator makes a kernel check fail and cannot produce a false theorem (`docs/verified-manual.md:9`).

The trusted-base document states the condition under which a theorem describes a running artifact. A theorem's conclusion holds of a running artifact only if every item of the artifact's section, and every item of the shared components, does what its table says (`docs/trusted-base.md:3`). Each section of the document covers one kind of artifact, which leanexe's README names by a code,

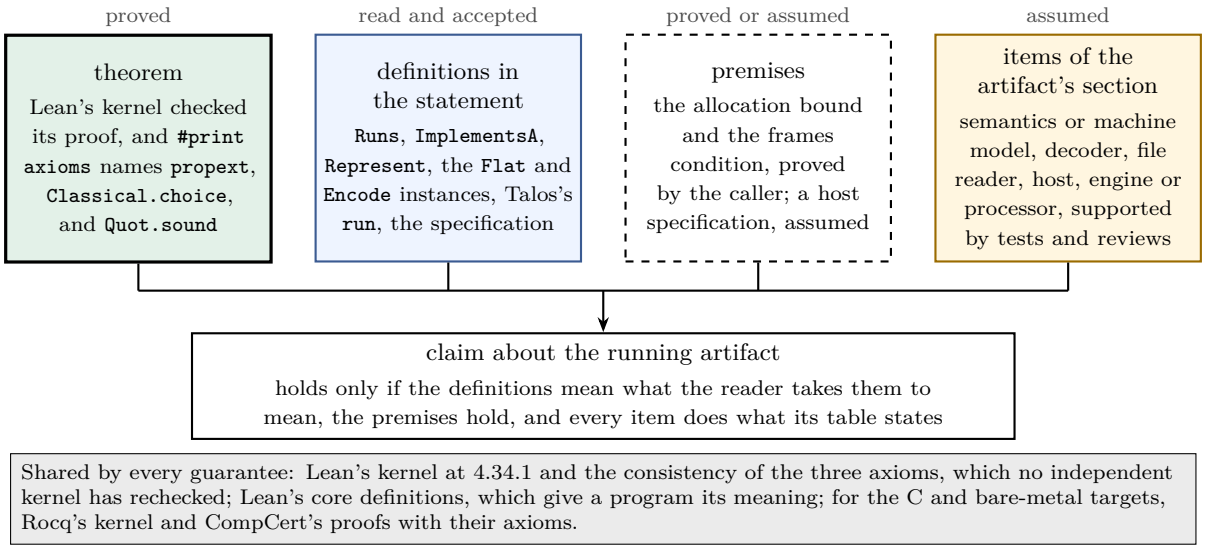

Figure 1: The parts of a leanexe guarantee and the support of each. The box with a heavy border is a theorem that Lean's kernel checks. The blue box holds the definitions that fix what the statement says, which a reader reads and accepts. The dashed box holds the premises, which a caller proves for its inputs or takes as assumptions. The orange box holds the items that tests and reviews support and that no proof covers, which Table 1 of the companion report on leanexe's targets lists for each target [9]. The gray box lists what every guarantee takes without proof (`docs/trusted-base.md:3`, `docs/trusted-base.md:7`).

such as WASM for a WebAssembly module of the verified compiler or A64 for an AArch64 Image (`README.md:86–98`). The shared components include Lean's kernel, the proof policy of Section 3, and Lean's core definitions (`docs/trusted-base.md:9–15`).

Every guarantee shares Lean's core definitions, because they give each source program its meaning. A program of the dialect, leanexe's first-order subset of Lean, computes with Lean's `UInt64`, `Int64`, `Bool`, and `Array` and with the floating-point models `Float.Model` and `Float32.Model` of Lean 4.34. Each theorem states that the compiled code computes the function that these definitions give (`docs/trusted-base.md:14`). Tests compare compiled code with native Lean on concrete inputs. Lemmas relate Lean's floats to the IEEE 754 functions of Talos, the WebAssembly interpreter written in Lean in whose semantics the module's theorem holds (`docs/trusted-base.md:14`, [7]).

2.2 Definitions that a statement names

The definitions that a statement unfolds to fix its meaning, so they belong to the trusted base, although they are ordinary Lean code (`docs/trusted-base.md:233`). A lemma in any file can make a proof fail and cannot change what a statement says. A reader of a statement therefore needs its definitions and none of the lemmas (`docs/trusted-base.md:233`).

For a function `f` of a program `p` that the command `verified_compile` compiles, the theorem `p.f.implements` states that the module computes `f`. That theorem unfolds to `ImplementsA`, the class `Represent` with its instances, and the `Flat` and `Encode` instances of the types that `f` takes and returns. It also unfolds to a model of the heap and to Talos's `run`, `Store`, and `Value` (`docs/trusted-base.md:237–239`). These definitions fix how a Lean value appears as WebAssembly words and heap data, and a reader must accept them as the intended representation [7].

A namespace does not mark which definitions a statement names. The files that hold statement

```
def Trapped : Bool → Result α → Prop
  | true, r => ∃ st, r = .Trap st "unreachable"
  | false, _ => False

def Runs (aborts : Bool) (env : HostEnv α) (m : Module) (id : Nat) (initial : Store α)
    (args : List Value) (P : Store α → List Value → Prop) : Prop :=
  ∃ N, ∀ fuel ≥ N, (∃ vs st, run fuel m id initial args env = .Success vs st ∧ P st vs) ∨
    Trapped aborts (run fuel m id initial args env)
```

Listing 1: The definitions of `Trapped` and `Runs` (LeanExe/Pipeline/Aborts.lean:16–43), without their docstrings. The term `run fuel m id initial args env` is Talos's run of function `id` of the module `m` with at most `fuel` steps.

definitions also hold lemmas, and modules in other directories declare names in the namespace `LeanExe.Pipeline` (docs/verified-manual.md:32). The last section of the trusted-base document lists, for each kind of statement, the definitions that it reaches and the files that hold them (docs/trusted-base.md:231).

A specification theorem adds the definitions of its specification. A hashing theorem equates a program with a transcription of a standard, such as FIPS 180-4 for SHA-256. An accuracy theorem compares a result with a real function of Mathlib through a rounding reference (docs/trusted-base.md:214). The trusted-base document lists each such definition as trusted to say what its name says. Published test vectors check the transcriptions of the hashing standards, and independent reviews check the specifications of programs written from a request in English (docs/trusted-base.md:218–229).

2.3 Traps, premises, and statements that hold of every module

A statement that allows a trap holds of a module that always traps. For a compiled function, `p.f.implements` states `ImplementsA aborts`. When the flag `aborts` is true, a run may end in a trap at the WebAssembly instruction `unreachable` instead of a return (docs/verified-manual.md:13). Listing 1 gives the definitions that take the flag: `Runs true` holds whenever every run with enough fuel traps at `unreachable`, whatever the postcondition `P`. Leanexe's documents state the consequence: a module that always traps satisfies `Implements` (docs/trusted-base.md:35, docs/design.md:251).

The trap-free theorem `p.f.trapFree` states `ImplementsA false`, which excludes the trap, under two premises (docs/verified-manual.md:15). The first premise states that `top`, the allocator's bump pointer, lies at least the function's allocation bound below the memory cap. The theorem `p.f.bound_eq` states that bound as a term that rewriting turns into a closed form [7]. The second premise, the frames condition, applies to code that takes the call depth and states that every call of such a function in the run happens below depth 1,000.

The premises of `p.f.trapFree` are Lean terms in the function's parameters, so a caller proves them for its inputs (docs/trusted-base.md:35). A claim that a call returns therefore names `p.f.trapFree` and the proofs of its premises. A claim that names only `p.f.implements` says nothing about a call that traps.

A statement holds vacuously when no case of interest meets its premises, and its conclusion says little when a trivial run meets it. The kernel accepts such a statement as it accepts any other, so no failed proof reveals one (VKB/entries/non-vacuous-statements:3). Leanexe proves further theorems that exclude such cases, and Table 1 lists those that its documents name.

Table 1: Statements that would hold of every module or every run in a stated case, and the theorems that exclude the case.

Statement	Case in which it holds trivially	Theorem that excludes the case	Source
`p.f.implements`	The module traps at `unreachable` on every run	`p.f.trapFree`, under the allocation bound and the frames condition	`docs/verified-manual.md`:15
`Implements` over a recursive type	The type's layout breaks a condition of the record header, so no value has a representation	`EncodeWellFormed`, which `verified_recursive_type` proves	`docs/verified-manual.md`:40
WASI theorems over every host that meets a host specification	No host meets the specification	`Host.Total.satisfiable`, which builds a host that meets it	`docs/trusted-base.md`:54
`delim_prefix`: the bytes written begin the answers to the segments read	The run writes nothing	`delim_answered`: when a window reads a byte, the bytes written are all the answers	`VKB/entries/non-vacuous-statements`:11
Floating-point theorems under the hypothesis `F7.Hyp`	No state of the processor meets the hypothesis	`F7.hyp_tcg`, `F7.hyp_kvm`, and `F7.hyp_sme`, for the states of the Isla profiles	`VKB/entries/non-vacuous-statements`:5

Table 2: The axioms of Lean 4.34 [18, §8.4], in plain words, and leanexe's treatment of each.

Axiom	Statement	Leanexe's treatment
`propext`	Two propositions that imply each other are equal	Permitted
`Classical.choice`	A proof that a type is nonempty gives an element of the type	Permitted
`Quot.sound`	Two elements that a relation relates have equal classes in the quotient by that relation	Permitted
`sorryAx`	Any statement: Lean adds it for an unfinished proof	Rejected by every check; `verified_compile` refuses a recursive definition that uses it
`Lean.ofReduceBool`, `Lean.ofReduceNat`, `Lean.trustCompiler`, and the axiom that `native_decide` or `bv_decide` adds for each computation	A computation by Lean's compiled code gives the result that it reports	Rejected; leanexe's design forbids `native_decide` and `bv_decide`
An axiom that a file declares	Any statement that the file names	Rejected

3 Three permitted axioms and the checks that reject any other

Leanexe's proofs use three axioms, and leanexe's checks fail on any other. An axiom is a statement that Lean accepts without proof, so a guarantee depends on the axioms that its proof uses. The three are `propext`, `Classical.choice`, and `Quot.sound`, which the Lean Language Reference describes as known to be consistent [18, ch. 8]. Table 2 states each in plain words, with the axioms that leanexe's checks reject (`docs/verified-manual.md`:830).

Leanexe proves closed facts, such as the bytes of a module's encoding or a bound at a concrete input, by kernel evaluation. The tactic `decide +kernel` has the kernel evaluate a decision procedure, so the proof adds no axiom (`docs/trusted-base.md`:13). The tactic `native_decide` evaluates the procedure with compiled code instead. Since Lean 4.29.0, Lean records each such computation as an axiom of its own, as it does for `bv_decide` [13]. Leanexe's design forbids both tactics, since each adds an axiom that asserts the correctness of compiled code (`docs/design.md`:145).

The command `#print axioms T` displays every axiom on which the declaration `T` transitively relies [18, §8.5]. Leanexe's theorem files end with such commands. The full check is the script `tools/full-check`, which runs leanexe's checks in order (Section 6.4). Its build step fails when a report from a file of the repository names another axiom (`tools/full-check`:39–41).

The axiom walk, `tests/verified/AllAxioms.lean`, covers the declarations that no `#print axioms` names. It imports the libraries `LeanExe`, `Examples`, `Demos`, `Verified`, `Library`, and `TestSupport`, follows the constants of every declaration, and fails on any axiom beyond the three (`tests/verified/AllAxioms.lean`:3–10). Step 1 of the full check runs the axiom walk (`tools/full-check`:816–820). The command `verified_compile` also refuses a recursive definition that uses `sorry` (`Verified/Reflect/Definitions.lean`:5566–5567).

Modules outside the six libraries of the axiom walk have audits of their own. The Isla and image theorems under `tests/TestSupport/A64` and `M7`, the reference semantics in `LeanExe/Native/ClightN/`, and the file modules, which hold the theorems about emitted files (Section 5), have their own `#print axioms` reports (`docs/trusted-base.md`:13). Their drivers require an exact number of reports, none of which may name an axiom beyond the three, so a module that printed no report would fail its check (`VKB/entries/discriminating-checks`:11). An audit inside the build, `Demos/OrderBook/ProcessAxioms.lean`, throws an error on a forbidden axiom in any theorem of its namespaces (`VKB/entries/out-of-build-checks`:9).

The axiom reports, the axiom walk, and the audits are Lean programs that run outside the kernel, and this report counts them as tests. Each reads the declarations of the environment and follows their constants, so it depends on the kernel's check of each declaration when Lean added it. A metaprogram can add a declaration to the environment without that check [2, §7]. Section 4.2 describes the replay that detects such a declaration.

4 Lean's kernel, the absence of an independent check, and Rocq's kernel

4.1 Lean's kernel and a recorded defect

Lean's kernel is a type checker for Lean's core type theory. It checks every inductive type and definition before the elaborators add it to the environment [19, §2.3]. It has no termination checker and performs no unification: the elaborator turns recursive functions into uses of primitive recursors and carries out unification before the kernel's check [19, §2.3]. De Moura et al. describe the kernel of the 2015 system as small and trusted, and they write that a small kernel makes an independent type checker simpler to implement [5, §2, §5]. Every leanexe theorem trusts the kernel of leanexe's toolchain, Lean 4.34.1 (`lean-toolchain`:1, `docs/trusted-base.md`:11).

Lean 4's kernel has had defects that admitted a proof of `False`. Carneiro writes that the kernel, which Lean 4 kept in C++ and extended for performance, acquired soundness bugs, and that the work on his checker Lean4Lean found one [2, Abstract, §1]. The Lean issue "Kernel accepts wrong-structure projections, allowing an axiom-free proof of `False`" reports a defect that a public repository, CollatzLean at `a793574`, used in a disproof of the Collatz conjecture [11, 17].

De Moura's postmortem states that CollatzLean's disproof is not valid, because it exploits a defect in the kernel's handling of nested inductive types. Lean 4.32.2 fixes the defect [4, 14]. Leanexe's trusted-base document records that the kernels of Lean 4.27.0, 4.29.1, 4.31.0, and 4.32.0 accept CollatzLean's construction. It records that the kernels of 4.34.0-rc2 and 4.34.1 reject the construction's projection, so the construction does not derive `False` under them (`docs/trusted-base.md`:12).

Lean 4.34.1 is a release of runtime fixes to Lean 4.34.0 [15]. Its release notes describe a defect that could trigger a use-after-free in the official kernel on a machine with at least 18 GB of free memory. The fix "fix: preserve sticky counts in deletion cascades" closes that defect. The notes add that such a use-after-free could be extended into a proof of `False` [15]. Leanexe changed its toolchain to 4.34.1 in the commit "Move to Lean v4.34.1 for its kernel fixes" [23].

4.2 One recorded replay and no independent kernel

No independent kernel has checked a leanexe proof. An independent kernel is a separate implementation of Lean's type checker, which can reject a proof that a defect of Lean's kernel admits. The checker `leanchecker` replays the declarations of each compiled module through the kernel, and the replay detects a declaration that a metaprogram added without the kernel's check [20, 21]. The checker uses Lean's kernel, so it cannot detect a defect of that kernel. Its README states that it is not an external verifier [21].

Leanexe's records show one replay of every module. The commit "Record the module replay and the byte comparison" of 30 October 2026 records a replay by `leanchecker` under the release candidate Lean 4.34.0-rc2. The replay covered every module that declares something, and no module failed [25]. The full check does not run `leanchecker`, and leanexe's history records no replay under Lean 4.34.1 (`tools/full-check`).

At commit `4556fa84`, the trusted-base document states in the present tense that `leanchecker` replays every module with no failure, and it gives no date or toolchain (`docs/trusted-base.md:11`). The commit "Replace metaphors, provenance, and counts in documents" removed the date from that row of the document [24]. The commit "Move to Lean v4.34.1 for its kernel fixes" changed the row's toolchain without recording a new replay [23].

Independent kernels for Lean 4 exist, and leanexe has run none of them. Lean4Lean is a type checker for Lean, written in Lean, that can check all of Mathlib. Carneiro reports it 20 to 50 percent slower than the C++ kernel, with parts of its correctness proved [2, Abstract, §1, §4.1]. Nanoda is an external type checker written in Rust that reads exported declarations and permits only the axioms that its configuration lists [1]. Lean 4.35.0, a later release than leanexe's toolchain, bundles Lean4Lean, nanoda, con-leche, and con-ron, together with an option `--paranoid` of `lake check` that runs them [16].

An independent kernel detects only the defects that it does not share with Lean's kernel. Carneiro writes that Lean4Lean mirrors the C++ kernel closely, that most defects he found while formalizing Lean4Lean are replicated in the C++ kernel, and that the two implementations share the type of expressions [2, §3.1, §3.2]. De Moura's postmortem of the CollatzLean defect reports that Lean4Lean shares the defect, since its handling of inductive types ports the reference implementation [4]. The postmortem also reports that the construction passed an earlier version of nanoda through an unrelated defect of nanoda. Figure 2 shows the checks of leanexe's Lean proofs and the one that has not run.

4.3 Rocq's kernel and CompCert's proofs for the C and bare-metal targets

The C and bare-metal targets depend on Rocq's kernel and on CompCert's proofs, in addition to Lean's kernel. CompCert compiles leanexe's C to an executable. The C target trusts CompCert's theorem `transf_c_program_correct`, together with Rocq's kernel, CompCert's axioms, and CompCert's parameters of the target (`docs/trusted-base.md:79`).

A bare-metal program's Lean theorem states a judgment of the Lean copy of CompCert's Clight. The Rocq file `rocq/Clight2Asm.v` composes CompCert's forward simulations. The composition

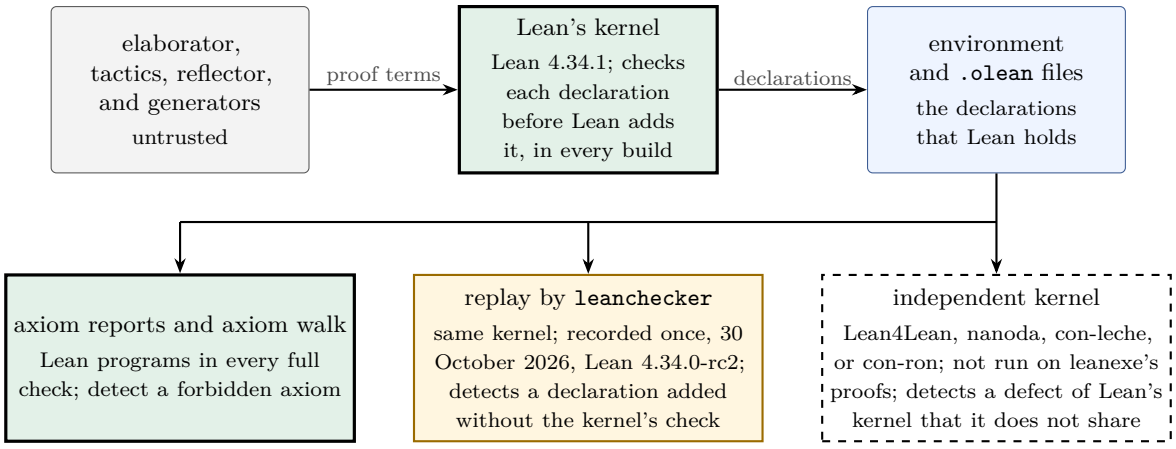

Figure 2: The checks of leanexe's Lean proofs. The gray box holds the programs that write proof terms. The box to its right, with a heavy border, is Lean's kernel, and the blue box holds the kernel's output. Of the checks that read that output, the green box with a heavy border runs in every full check. The orange box is a check that leanexe's history records once, and the dashed box is a check that has not run on leanexe's proofs. Section 4.3 describes the checks of the Rocq proofs of the C and bare-metal targets.

derives a judgment of CompCert's assembly language, Asm, from the corresponding judgment of CompCert's own Clight (`docs/trusted-base.md`:102). The companion report on leanexe's targets describes that composition and the step between the two Clight judgments [9, §5]. Proofs cover that step for eleven AArch64 programs, and review and tests support it for every other program.

Leanexe's script `rocq/check.sh` compiles the Rocq files against a CompCert 3.18 tree whose proofs are built (`rocq/check.sh`:2–8). It compares the assumptions that the Rocq command `Print Assumptions` reports for each theorem with those of CompCert's `transf_c_program_correct`. The script fails when the composition uses an assumption outside that set, with one exception: `ClassicalEpsilon.constructive_indefinite_description`, which CompCert's argument about behaviors uses (`rocq/check.sh`:57). The script compares the names of the assumptions, so it accepts any subset of CompCert's. The full check runs it only on a host with `coqc` and such a tree (`docs/verified-manual.md`:962).

CompCert's `lib/Axioms.v`:49 declares proof irrelevance, and the same file proves CompCert's lemmas of functional extensionality from the axiom `functional_extensionality_dep` of Rocq's standard library [3]. CompCert's semantics of C imports excluded middle, `Classical_Prop.classic` (`cfrontend/Cstrategy.v`:19), which leanexe's Rocq file also applies (`rocq/Clight2Asm.v`:244). CompCert also states as axioms that the abstract semantics of external functions and of inline assembly have the properties that its proofs use (`common/Events.v`:1485). On AArch64, a further axiom states that the two halves of a symbol's address add up to the address (`aarch64/Asm.v`:389). CompCert's other assumptions are parameters: compiler options, the target's configuration, and untrusted OCaml routines such as the register allocator, whose results CompCert's checks validate [26, §4.2].

This report names CompCert's axioms from CompCert's sources. No record of a run of the script keeps the output of `Print Assumptions`, and the author had no Rocq installation. A run of `rocq/check.sh` would decide whether `Print Assumptions` also reports axioms of Rocq's real numbers, which CompCert's floating-point proofs use through the Flocq library. Rocq's standalone checker of compiled libraries, `rocqchk`, type-checks the logical content of compiled files in a separate

program, which code that tactics load cannot alter [33]. Leanexe does not run `rocqchk`.

5 File theorems and file readers

The step from a module in Lean to the file that an engine or a processor loads needs a theorem of its own. The program `tools/Emit.lean` evaluates the module and its encoding with Lean's compiled code and writes the file. A file without a file theorem therefore trusts that compiled code (`docs/trusted-base.md`:28).

For a WebAssembly file, the command `file_theorems` states that the bytes that the elaborator `binary_file%` reads from the file equal `encode p.module` and decode to the module (`docs/trusted-base.md`:39, `LeanExe/Pipeline/FileBytes.lean`:89–122). The kernel compares the bytes. The elaborator `binary_file%` has six lines and reads the file, relative to the source file, when Lean checks the theorem's module (`LeanExe/Pipeline/FileBytes.lean`:15–22). The theorem describes the file as it was at that check, so the verified driver, `tests/verified/run.sh`, writes each file and then checks its file module (`docs/trusted-base.md`:45).

The AArch64 and Cortex-M7 back ends read their image files with the elaborator `file_nat%`, which gives the kernel the file's bytes as one natural number. The kernel evaluates the generator on the program and compares the generator's bytes with that number (`LeanExe/FileNat.lean`:3–11, `LeanExe/A64/File.lean`:16–20). The theorem's premise states that memory holds the file's bytes, so the trusted base holds neither the compiled generator nor the code that wrote the file [9, §2.3]. The file reader `file_nat%` remains in the trusted base, with inspection and a seeded byte change as its checks. The trusted base also holds the assumption that the checked file is the file that QEMU loads or that the flash receives (`docs/trusted-base.md`:135, `docs/trusted-base.md`:174).

Compiled Lean code remains in the trusted base for two textual outputs: the C translation and the WGSL and PTX kernels. For the C translation, compiled Lean evaluates the translation that each build compares with `clightgen`'s reading of the printed file (`docs/trusted-base.md`:76). The kernel checks each translated function against the translator only for the eleven programs of the reference semantics N, which the companion report on leanexe's targets describes [9, §5.2]. For WGSL and PTX kernels, compiled Lean evaluates the printer, and a parser with a proof that it inverts the printer reads the text back in compiled code [9, §9].

6 Tests that support trusted items

6.1 Tests and proofs

A test decides a statement only on the inputs that it runs, while a theorem quantifies over every input. A test can therefore show a statement false and cannot show it true (`VKB/entries/statement-testing`:9). Leanexe uses tests for the trusted items that no proof covers. Examples are the agreement of Talos with the WebAssembly specification, of the Lean copy of CompCert's Clight with CompCert's Clight, of the machine models with processors, and of the transcribed standards with their published vectors (`docs/trusted-base.md`:23, `docs/trusted-base.md`:71, `docs/trusted-base.md`:130, `docs/trusted-base.md`:218–223). An item that a test supports stays in the trusted base, and the test's result enters no proof, as the trusted-base document states of the test tools of leanexe's energy-gateway demonstration (`docs/trusted-base.md`:118).

Table 3: Seeded faults of leanexe's checks and the outcome that each check must give.

Check	Seeded fault	Required outcome	Source
Co-simulation of the Cortex-M7 model with the board	The model with the carry input of `ADC` replaced by 0	A difference on exactly the `ADC` cases with the carry flag set, on every run	`VKB/entries/discriminating-checks:5`
Comparison of the Rocq and Lean renderings of the reference semantics N	Planted errors in copies of the renderings and proofs, among them a changed operator, a weaker theorem under a statement's name, and an assumed proof	The comparison reports exactly those errors	`docs/trusted-base.md:72`
Frame test of WASI calls under Wasmtime	A region of a call's description declared short, late, or moved by half its alignment	Each fault makes the test fail	`VKB/entries/discriminating-checks:31`
Frame test of the Linux call layer	A fault seeded into the descriptions of the calls	Each fault makes the test fail, natively and under `qemu-aarch64`	`docs/trusted-base.md:150`
Kernel check of a Cortex-M7 image	One changed byte of the image	The kernel rejects the image's theorem, or the part that holds the byte, in a single recorded run	`VKB/entries/discriminating-checks:9`
Bare-metal startup code	A build without the floating-point enable, a store to the guard page, a short stack, or a wrong UART address	Each build fails visibly	`docs/trusted-base.md:108`

6.2 Checks and the seeded faults that they must detect

A check can pass vacuously when it compares nothing, compares a result with itself, or accepts every outcome (`VKB/entries/discriminating-checks:3`). Leanexe's public verification knowledge base states a rule against such checks. Under the rule, a check counts as evidence only after a control, which is a recorded run in which a fault of the kind that the check should catch makes the check fail (`VKB/entries/discriminating-checks:3`). The strongest form of the rule requires the predicted failure, so that a check that fails for an unrelated reason, such as an unreadable result file, does not pass the control (`VKB/entries/discriminating-checks:5`).

The seeded-fault rule applies the idea of mutation analysis to leanexe's checks. In mutation analysis, a tester judges test data by whether they distinguish a program from its mutants, which are copies of the program with simple seeded errors [6, p. 36]. Table 3 lists seeded faults of leanexe's checks and the outcome that each must produce.

Some of leanexe's controls are single recorded runs. The controls with altered logs for the replay checker of the battery-management demonstration ran once, and so did the controls with changed bytes of the Cortex-M7 images. A change to those checks therefore requires new runs of their controls (`VKB/entries/discriminating-checks:39`).

The open leanexe issue "VKB: a seeded byte fault that writes a fixed value can leave the file unchanged" concerns seeded faults that write fixed values [22]. Such a fault leaves the file unchanged where the byte already holds the value, so the check passes on the original file. The issue proposes deriving the new byte from the old one, and it reports no defect in leanexe's code.

6.3 Tests of statements and reviews of specifications

Leanexe's verification knowledge base describes controls for statements and specifications, whose meaning no proof can check. Before the proof of a new statement starts, an executable model tests

the statement on many inputs (VKB/entries/statement-testing:3). The test counts only after it has rejected false variants of the statement, such as a weakened premise or a property that the procedure is known to fail (VKB/entries/statement-testing:13). A review of a specification by an agent that did not write it runs wrong programs on the specification's cases (VKB/entries/independent-review:3). A wrong program that passes every case gets a case that detects it, or the review records that no input can detect it and why (VKB/entries/independent-review:7).

6.4 The full check and the theorems outside it

The script `tools/full-check` runs leanexe's checks in order. Its steps are the build of the six libraries with the axiom walk, the verified driver, the drivers of the targets, the Rocq checks, the Talos conformance driver, and the checks against independent references (tools/full-check:1–34). It sets `LEANRUN_FAST=0`, so that every step runs in the checked configuration. Its build step fails on a `sorry` warning and on a forbidden axiom report (tools/full-check:39–49). The script writes its record of a run to `STATUS.md`, which Git ignores, so leanexe's repository holds no record of a full check at commit `4556fa84` (docs/verified-manual.md:960).

The full check does not cover some theorems. It runs none of the drivers of the Cortex-M7 board and QEMU file modules, the tabulators' Image files, and the proofs over Sail's translation. Only those drivers check these theorems (VKB/entries/out-of-build-checks:5). The scripts of `tests/linux/` run only on an AArch64 Linux host, and the Rocq steps only with `coqc` and a CompCert tree whose proofs are built (docs/verified-manual.md:962). A passing full check says nothing about a theorem that none of its steps elaborates (VKB/entries/out-of-build-checks:3).

7 Steps for reading a leanexe guarantee

A reader who applies a leanexe theorem to a running artifact takes six steps, each with a source at commit `4556fa84`.

1. Find the theorem that the claim names. Leanexe's documents name the theorem behind each sentence that says what an artifact does. A sentence about a property that no theorem states names the property's basis instead (VKB/entries/claims-and-theorems:5).
2. Read the statement and the definitions that it reaches, which the trusted-base document lists for each kind of statement (docs/trusted-base.md:231).
3. List the premises, and decide for each whether the caller proves it for its inputs, as for an allocation bound, or assumes it, as for a host specification.
4. Read the axiom report: `#print axioms` of the theorem names `propext`, `Classical.choice`, and `Quot.sound` and no other.
5. Read the artifact's section of the trusted-base document, with the evidence for each item: a proof, a test with its seeded faults, or a review. Table 1 of the companion report on leanexe's targets condenses these sections [9].
6. Confirm that the file that runs is the file that the kernel checked, through a file theorem or an image check elaborated after the file was written.

Leanexe's quicksort illustrates the reading of a guarantee. The quicksort guide's claim that the module sorts in place names the theorems `quicksort_implements` and `quicksort_bound`, which Listing 2 gives (Library/Sorting/Quicksort/README.md:30–31). `ImplementsA false` states that function 9 of the module, its export `quicksort`, returns without a trap. Its precondition `heap.Within` requires room for the generated bound below the memory cap, and its postcondition

```
theorem quicksort_implements :
    ImplementsA false compiled.module 9 (fun x : Moved (Array UInt64) => expected x.val)
      (fun x heap store => heap.Within store compiled.module (compiled.quicksort.bound x))
      (fun x heap _ heap' _ => heap'.top.toNat ≤ heap.top.toNat + compiled.quicksort.bound x)

theorem quicksort_bound (xs : Array UInt64) :
    compiled.quicksort.bound { val := xs } = Verified.blockCost 128
```

Listing 2: The statements of `quicksort_implements` and `quicksort_bound` (Library/Sorting/Quicksort/Compiled.lean:29–39), without their proofs.

states that `top` rises by at most the bound. The theorem `quicksort_bound` gives that bound as the cost of one block of 128 words, so the premise asks for the same room for every input.

The specification function that `quicksort_implements` names, `expected`, returns the input's elements in the order of Lean's `List.mergeSort` with the unsigned comparison of words (Library/Sorting/Quicksort/Spec.lean:19–20). The specification names the result and not the method, and its docstring states that it does not distinguish Knuth's Algorithm Q from other sorting methods (Library/Sorting/Quicksort/Spec.lean:1–15). The theorem `quicksort_eq` states that the Lean function `quicksort` equals `expected` on every array with fewer than 2^{64} elements. Its proof needs only that steps Q1 to Q8 permute the array, since step Q9, straight insertion, sorts every array. A defect of steps Q1 to Q8 that kept a permutation and finished would therefore change no result and contradict no theorem (VKB/entries/independent-review:7).

The statement of `quicksort_implements` bounds the memory that the call takes and not its running time. The module's theorem states only that some amount of fuel suffices [7]. On an array in increasing order, steps Q2 to Q7 make about $N^2/2$ comparisons, which the guide states and no theorem covers (Library/Sorting/Quicksort/README.md:13).

Quicksort's compilation file prints the axiom report of `quicksort_implements`, which names only the three permitted axioms, as the guide states (Library/Sorting/Quicksort/Compiled.lean:41, Library/Sorting/Quicksort/README.md:34). The file theorems `quicksort_file` and `quicksort_file_decode` state that `build/verified/quicksort.wasm` holds the module's encoding and decodes to the module (Library/Sorting/Quicksort/README.md:32). A run of that file in Wasmtime trusts the items of row WASM of Table 1 of the companion report on leanexe's targets [9]. Among them are Talos's semantics as a description of Wasmtime, the binary decoder, and the host's placement of the array. Like every leanexe theorem, `quicksort_implements` also trusts Lean's kernel at 4.34.1, which no independent kernel has rechecked.

8 Related work

Table 4 compares leanexe's account of trust with those of four verified systems. Each of the four names its prover among what it trusts, as leanexe names Lean's kernel and, for C and bare metal, Rocq's. Leroy lists the Coq proof assistant among the parts of CompCert that remain trusted and calls it probably the least concern [26, §5].

The authors of CakeML and seL4 name ways to check their provers' proofs with separate programs. Kumar et al. write that a prover or checker is part of CakeML's trusted computing base [12, §12.1]. They add that one could apply a separate checker such as OpenTheory, or trust HOL4, whose soundness depends on a kernel of about 1,000 lines. The seL4 Foundation assumes that the axioms of higher-order logic are consistent and that its prover checks the proof correctly. It states that Isabelle supports external proof checking [29].

Table 4: Statements, trusted bases, and checks of the prover's proofs for leanexe and four verified systems. The prover is in parentheses, and each trusted base is as the system's primary source states it.

System	Statement	Trusted base	Checks of the prover's proofs
leanexe (Lean; Rocq for C and bare metal)	For each program, the compiled code computes the Lean definition in the target's semantics, under stated premises	Lean's kernel and three axioms; the statement definitions; the items of Table 1 of [9]; for C and bare metal, Rocq's kernel and CompCert's proofs and axioms	Lean's kernel in every build; one replay with the same kernel; no independent kernel
CompCert [26] (Coq)	For a source program that does not go wrong, every behavior of the compiled code is a behavior of the source	The Clight and PowerPC semantics; the parser, assembler, and linker; extraction and the Caml compiler; Coq	Coq's kernel re-checks the proof terms
CakeML [12] (HOL4)	The machine code of the read-eval-print loop behaves as the CakeML semantics allows, or stops with an out-of-memory error	The x86-64 and CakeML semantics; code that sets up input and output; the prover	HOL4's kernel; a separate checker proposed
seL4 [10, 30] (Isabelle/HOL; HOL4 and SMT solvers for the binary)	The C code refines the abstract specification, and the gcc binary refines the C semantics	2009: the compiler, assembly code, boot code, cache management, and hardware; 2013: the ARM model, the validation tool, and the binary's import into HOL4 in place of the compiler, linker, and C parser	Isabelle supports external checking; none reported
HACL* [34] (F*, Z3)	The Low* code is memory safe, functionally correct against its specification, and secret independent	The F* typechecker, Z3, the KreMLin compiler, the C compiler unless CompCert, and hand-written vector C	Not stated

The theorems of verified compilers end at different stages between the source program and the bytes that run. CompCert's theorem ends at the abstract syntax of its assembly language, so its assembler and linker remain unverified [27, §1.2, §1.3]. CakeML evaluates its compiler inside the logic to obtain the bytes of its machine code. Kumar et al. trust a small amount of code that sets up input and output, and no other compiler [12, §2.1, §12.1].

Sewell et al. prove that seL4's gcc binary refines the C semantics. The proof removes the compiler, the linker, and the C parser from the trust chain and adds the ARM model and the validation tools [30, §2.1, §6]. Leanexe's AArch64 and Cortex-M7 back ends compare the written file with the generator's output in the kernel, and its WebAssembly target does so through file theorems (Section 5).

Every verified system's statement depends on definitions that a reader must accept. Leroy calls the agreement of the formal semantics with language standards perhaps the most delicate issue, and he proposes review by experts and tests of executable semantics [26, §5]. Klein et al. write that the proof does not guarantee that the specification describes the behavior that the user expects [10, §4.5].

HACL*'s authors write specifications that a reader can audit against the published standard and test against vectors from the RFCs. They note that the usual vectors miss input sizes that would catch padding mistakes [34, §2, §3]. Leanexe's statement definitions and transcriptions need the same kind of trust, and Section 6 describes the reviews and controls that support them.

Compiler correctness statements differ in the outcomes that they admit. CompCert's theorem covers source programs that do not go wrong [26, §2.1], while leanexe's dialect gives every operation

Table 5: Proved, tested, and assumed items that every leanexe guarantee shares.

Kind	Item	Source
Proved	Each program's generated theorems, from `Prog.correctStaticWith`, and each target's composed theorems	[7, 9]
	The file theorems of WebAssembly files and the kernel's comparison of each image file with its generator's output	`docs/trusted-base.md`:39, `LeanExe/A64/File.lean`
	The theorems that exclude the trivial cases of Table 1	Table 1
	For C and bare metal, Rocq's composition from Clight to Asm	`docs/trusted-base.md`:102
Tested	The axiom reports and the axiom walk, which Lean computes outside the kernel	`tests/verified/AllAxioms.lean`
	One replay of every module by `leanchecker`, under Lean 4.34.0-rc2	[25]
	The comparison of Rocq's assumptions with CompCert's, by name	`rocq/check.sh`:57
	The trusted items of each target, with the seeded faults of Table 3	`docs/trusted-base.md`:3
Assumed	Lean's kernel at 4.34.1 and the consistency of `propext`, `Classical.choice`, and `Quot.sound`. No independent kernel has checked the proofs.	`docs/trusted-base.md`:11
	For C and bare metal, Rocq's kernel and CompCert's proofs and axioms	`docs/trusted-base.md`:79
	Lean's core definitions and the statement definitions mean what a reader takes them to mean	`docs/trusted-base.md`:233
	The specifications' transcriptions and Mathlib's definitions say what their names say	`docs/trusted-base.md`:214
	The file that runs is the file that the kernel checked, and the file readers read it	`docs/trusted-base.md`:45

a total meaning, so leanexe's theorem has no such premise [7]. Leroy notes that a compiler that always fails is verified, although useless [26, §2.2]. In the same way, a module that always traps satisfies leanexe's `Implements`. CakeML's theorem allows a stop with an out-of-memory error [12, §2.1], and leanexe's trap-free theorem excludes its trap under a bound that the caller proves (Section 2.3).

9 Proofs, tests, and assumptions

Table 5 separates what Lean's kernel or Rocq checks from what tests measure and what the theorems take without proof, for the items that every target shares. Tables 1 and 4 of the companion report on leanexe's targets give the items of each target [9]. The author read the statements and sources at commit `4556fa84` and did not rebuild the proofs.

10 Limitations

No independent kernel has checked a leanexe proof. The one recorded replay by `leanchecker` used Lean's own kernel under Lean 4.34.0-rc2, an earlier toolchain than Lean 4.34.1. The axiom checks are Lean programs outside the kernel. The script `rocq/check.sh` compares Rocq's assumptions by name. Rocq's standalone checker, `rocqchk`, has not run on leanexe's Rocq proofs.

Leanexe commits no record of a full check, so this report cannot state when commit `4556fa84` last passed one. The full check does not cover some theorems, and their drivers check them only

when someone runs those drivers (VKB/entries/out-of-build-checks:5). Some seeded-fault controls ran once, and the issue on seeded faults that leave a file unchanged is open [22].

The trusted base includes the statement definitions and the specifications, which reviews and tests support and no proof checks. A statement that allows a trap says nothing about a call that traps. A theorem whose premises do not hold for a call says nothing about that call.

This report gives the account of trust that every target shares, and the companion report on leanexe's targets gives the items of each target [9]. The facts about Lean's kernel defect and its rejection under 4.34.1 are leanexe's records, and the author did not reproduce them. This report describes no development whose sources are private.

References

[1] Chris Bailey. nanoda_lib. Git repository `ammkrn/nanoda_lib`, commit 0da9f98f of 7 November, 2026. URL https://github.com/ammkrn/nanoda_lib/tree/0da9f98feadd3a0a2ef4b3dbcf52a298a6fef929.

[2] Mario Carneiro. Lean4Lean: Verifying a typechecker for Lean, in Lean. arXiv:2403.14064v3, 2025. Version 3 of 14 September 2025.

[3] CompCert. CompCert 3.18. Git repository `AbsInt/CompCert`, tag v3.18, 2026. URL https://github.com/AbsInt/CompCert/tree/v3.18.

[4] Leonardo de Moura. Postmortem for kernel soundness bug #14576. Blog post of 1 August, 2026. URL https://leodemoura.github.io/blog/2026-8-1-postmortem-for-kernel-soundness-bug-14576/.

[5] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *Automated Deduction – CADE-25*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015. https://doi.org/10.1007/978-3-319-21401-6_26.

[6] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978. https://doi.org/10.1109/C-M.1978.218136.

[7] exaspec. Core leanexe: a verified compiler for a subset of Lean. Technical report, Morphism, 2026. URL https://exaspec.com/leanexedocs/leanexe-core.pdf.

[8] exaspec. Leanexe's numerical library: proved accuracy for compiled floating-point code. Technical report, Morphism, 2026. URL https://exaspec.com/leanexedocs/leanexe-numerics.pdf.

[9] exaspec. Leanexe's targets: composed theorems in stated semantics, with exact trusted bases. Technical report, Morphism, 2026. URL https://exaspec.com/leanexedocs/leanexe-targets.pdf.

[10] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, 2009. https://doi.org/10.1145/1629575.1629596.

[11] Ramana Kumar. CollatzLean. Git repository `xrchz/CollatzLean`, commit a793574 of 25 July, 2026. URL https://github.com/xrchz/CollatzLean/tree/a79357462a33d2a6babd4cf6c8d8bcd25425d653.

[12] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 179–191, 2014. https://doi.org/10.1145/2535838.2535841.

[13] Lean FRO. Lean 4.29.0 release notes. In *The Lean Language Reference*, 2026. URL https://lean-lang.org/doc/reference/latest/releases/v4.29.0/.

[14] Lean FRO. Lean 4.32.2 release notes. In *The Lean Language Reference*, 2026. URL https://lean-lang.org/doc/reference/latest/releases/v4.32.2/.

[15] Lean FRO. Lean 4.34.1 release notes. In *The Lean Language Reference*, 2026. URL https://lean-lang.org/doc/reference/latest/releases/v4.34.1/.

[16] Lean FRO. Lean 4.35.0 release notes. In *The Lean Language Reference*, 2026. URL https://lean-lang.org/doc/reference/latest/releases/v4.35.0/.

[17] Lean issue. Kernel accepts wrong-structure projections, allowing an axiom-free proof of False. Lean issue, opened and closed 28 July, 2026. URL https://github.com/leanprover/lean4/issues/14576.

[18] Lean reference. Axioms. In *The Lean Language Reference*, version 4.34.0, chapter 8, 2026. URL https://lean-lang.org/doc/reference/4.34.0/Axioms/.

[19] Lean reference. Elaboration and compilation. In *The Lean Language Reference*, version 4.34.0, chapter 2, 2026. URL https://lean-lang.org/doc/reference/4.34.0/Elaboration-and-Compilation/.

[20] Lean reference. Validating a Lean proof. In *The Lean Language Reference*, version 4.34.0, 2026. URL https://lean-lang.org/doc/reference/4.34.0/ValidatingProofs/.

[21] lean4checker. lean4checker. Git repository `leanprover/lean4checker`, archived, commit 91a7f0e8 of 25 March, 2026. URL https://github.com/leanprover/lean4checker/tree/91a7f0e8e9dffe927089f5a6edcfeeb8a0e07709.

[22] Leanexe. VKB: a seeded byte fault that writes a fixed value can leave the file unchanged. Leanexe issue, opened 26 November, open on 27 November, 2026. URL https://github.com/jsmorph/leanexe/issues/15.

[23] Leanexe. Move to Lean v4.34.1 for its kernel fixes. Leanexe commit fef8ce02 of 25 November, 2026. URL https://github.com/jsmorph/leanexe/commit/fef8ce024ae49af4df877155b8d902e05067be77.

[24] Leanexe. Replace metaphors, provenance, and counts in documents. Leanexe commit 92053c82 of 3 November, 2026. URL https://github.com/jsmorph/leanexe/commit/92053c82bb25ba98018655c69c555f92df60d0eb.

[25] Leanexe. Record the module replay and the byte comparison. Leanexe commit c61f6c23 of 30 October, 2026. URL https://github.com/jsmorph/leanexe/commit/c61f6c237e79b10daf2de6bfdaa34ec0d010d176.

[26] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. https://doi.org/10.1145/1538788.1538814.

[27] Xavier Leroy. *The CompCert C verified compiler: Documentation and user's manual, version 3.18*. Collège de France and Inria, 2026. URL https://compcert.org/man/. 30 August 2026.

[28] Morphism LLC. LeanExe. Git repository, commit 4556fa84 of 26 November, 2026. URL https://github.com/jsmorph/leanexe/tree/4556fa84205eee51b989e365a8519a33655833a2.

[29] seL4 Foundation. What the proofs assume. Web page, 2026. URL https://sel4.systems/Verification/assumptions.html. Read 27 November 2026.

[30] Thomas Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 471–482, 2013. https://doi.org/10.1145/2491956.2462183.

[31] Talos. Talos: a WebAssembly interpreter and code library in Lean. Git repository `jsmorph/talos`, commit 87e3aa5e, 2026. URL https://github.com/jsmorph/talos/tree/87e3aa5e8f6e6f3b3eb5e7e4c5aba43071002d47.

[32] The Mathlib Community. Mathlib. Git repository, commit 85e3a25e, 2026. URL https://github.com/leanprover-community/mathlib4/tree/85e3a25e006c35636f0e53b0e9296caca2685bc0.

[33] The Rocq Development Team. The Rocq prover 9.2 documentation: compiled libraries checker (rocqchk). Reference manual, 2026. URL https://rocq-prover.org/doc/V9.2.0/refman/practical-tools/coq-commands.html#compiled-libraries-checker-rocqchk.

[34] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1789–1806, 2017. https://doi.org/10.1145/3133956.3134043.