Core leanexe: a verified compiler for a subset of Lean

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. The verified compiler produces a WebAssembly module, which serves as the intermediate representation: the C and machine-code translations start from that module, and WebAssembly engines run the module itself. GPU kernels in WGSL and PTX come from a separate kernel language.

This report describes the verified compiler from the Lean subset to the WebAssembly module and the theorem that each module receives. With current proof technology, restricting programs to a small, total, first-order subset of Lean makes it practical to give each compiled program a kernel-checked correctness theorem. Specifications and proofs about the programs still use all of Lean, its mathematics library Mathlib, and other Lean developments. The report compares the approach with CompCert, CakeML, Œuf, CertiCoq, Axon, and other systems.

The module's theorem holds in Talos, a WebAssembly semantics written in Lean. Only tests relate Talos to the WebAssembly standard. That relation enters the trusted base only when the module runs in a WebAssembly engine.

1 Introduction

Leanexe compiles programs written in a restricted part of Lean, a language in which one writes both programs and mathematical proofs. For each compiled program, it produces a proof that the code computes the same function as the Lean program, and Lean checks that proof as it checks any other. The compiled code runs on AArch64 and x86-64 processors, on an Arm Cortex-M7 microcontroller, on GPUs, and in WebAssembly engines. The restriction applies only to the programs. Statements and proofs about a program may use all of Lean and its mathematics library, so a proof that Lean checks can relate the compiled code to a mathematical result.

A proof assistant checks proofs about definitions in its logic, and running a definition requires a compiler or an extraction tool. In Coq, for example, extraction is "a large complicated process that is in the trusted computing base" of a development that runs extracted code [19]. Lean 4 compiles definitions to C through an intermediate representation, erases proofs, and manages memory by reference counting [8, §3]. No theorem covers that compiler.

A proof that relies on Lean's compiled code depends on more than Lean's kernel, the small program that checks every proof term. Lean marks each such proof, such as one by the tactic `native_decide`, with a dedicated axiom. These axioms "track proofs that depend on the correctness of the entire compiler, and not just on the much smaller kernel" [4]. A user who proves a property of a Lean function and runs its compiled code therefore trusts the compiler. A verified compiler removes that trust with a theorem that its output computes the meaning of its input [16].

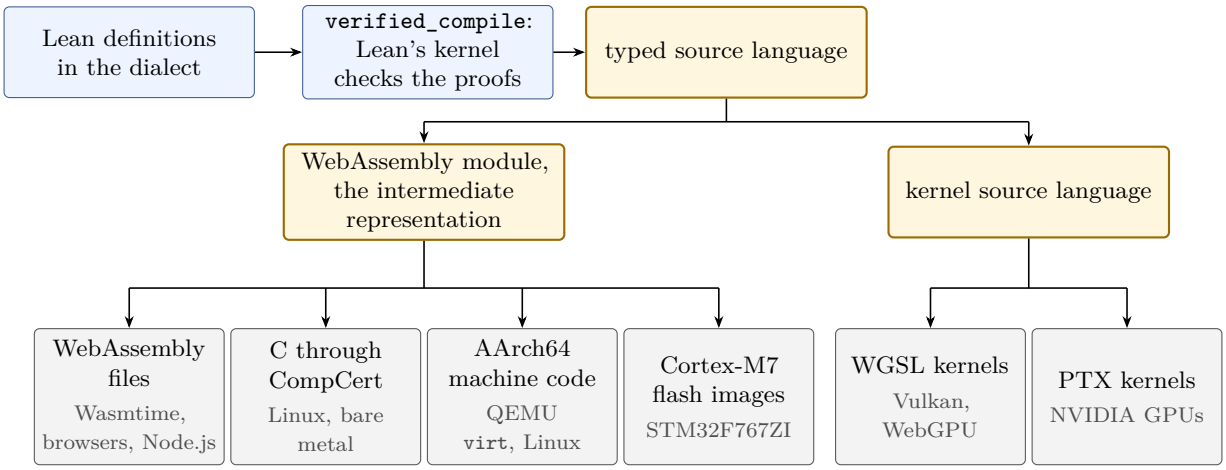

Figure 1: The structure of leanexe's toolchain, after the diagram in leanexe's README (`docs/toolchain-summary.mmd`). Blue boxes are the source and the command that compiles it, orange boxes are intermediate representations, and gray boxes are compiled forms, each with the platforms where it runs. C runs under Linux on AArch64 and x86-64 and as a bare-metal program on AArch64 and the Cortex-M7. The AArch64 back end writes bare-metal arm64 Images and Linux executables. The C, AArch64, and Cortex-M7 back ends start from the WebAssembly module, and WebAssembly engines run the module itself. GPU kernels come from the kernel source language. The companion report on leanexe's targets [10] describes the back ends and their theorems.

Leanexe's verified compiler, `compileWith`, is a Lean function from a typed source language to a WebAssembly module [18]. The source language is a Lean datatype of programs with a meaning function. One theorem, `Prog.correctStaticWith`, proved once, states that every program's module computes the program's meaning.

Leanexe's command `verified_compile` writes Lean definitions as programs of the source language, a step called reflection. The definitions must be in the dialect, the total, first-order subset of Lean that the command accepts. The command proves, with equations that the kernel checks, that each source program means the original definitions. It also states the theorems `p.f.implements` and `p.f.trapFree` of Section 6 for each definition `f`, where `p` is the name that the command gives the program. Œuf [19] divides the work in the same way: it checks its translation of each program into its source language and compiles the result with a compiler verified once.

The WebAssembly module that `compileWith` produces serves as the intermediate representation. The translation to C and the AArch64 and Cortex-M7 code generators start from it, and each composes the compiler's theorem with a simulation theorem of its own (`docs/native-translator.md:25`, `docs/a64-bare-metal.md:23`, `docs/m7-bare-metal.md:9`). WebAssembly engines also run the module as a final representation. Figure 1 shows the structure of the toolchain.

Leanexe makes the following contributions.

1. A compiler written in Lean from a typed first-order language to the WebAssembly module that serves as the intermediate representation, with one correctness theorem that covers every typed program (Section 5).
2. A front end, outside the trusted base, that states each program's theorem with Lean's own types (Section 5).
3. Allocation bounds computed from the code, and theorems of total correctness under them (Section 6).

4. For the WebAssembly target, a kernel-checked theorem that each module's bytes decode to the module (Section 5).
5. Specifications that use Mathlib and other Lean developments, among them theorems that OpenAI proved in Lean in 2026 (Section 7).

This report adds the rationale for restricting programs to the dialect while specifications keep all of Lean (Section 3).

This report describes leanexe at commit `4556fa84` of 26 November 2026, with Lean 4.34.1, Mathlib at `85e3a25` [34], and Talos at `87e3aa5e` [5]. Leanexe's `lakefile.lean` and `lake-manifest.json` pin the repository `jsmorph/talos` at that Talos commit, and leanexe's README links the upstream repository `cajal-technologies/talos`. Every link to a leanexe file points to commit `4556fa84`.

This report describes the verified compiler, the module's theorem, and the module's use as a WebAssembly target. The companion report on leanexe's targets [10] describes the translations from the module to C and machine code and their simulation theorems. It also describes the WASI commands and the WGSL and PTX kernels. A separate compiler for Lean actions with effects produces the WASI commands (`README.md:72`). Leanexe lowers the kernels from the typed source program through a separate kernel language (`docs/design.md:236`).

2 Background

2.1 Lean, its kernel, and its axioms

Lean 4 is a functional programming language and a proof assistant based on dependent type theory [8]. A definition is a term of the logic, and a theorem is a term whose type is its statement. The kernel accepts a theorem only after it checks that term. The kernel also evaluates definitions by reduction: the tactic `decide +kernel` proves a decidable proposition by having the kernel evaluate its decision procedure.

Lean's standard axioms are `propext`, `Classical.choice`, and `Quot.sound`. Leanexe's proofs use no other axiom. Leanexe's design forbids the tactics `native_decide` and `bv_decide`, because both add an axiom that asserts the correctness of compiled code (`docs/design.md:145`).

2.2 Verified compilers, translation validation, and embeddings

Leroy distinguishes a *verified compiler*, whose correctness theorem holds for all source programs, from *translation validation*, in which a validator checks each compilation after it runs [16, 22]. A *deep embedding* represents the programs of a language as values of a datatype in the logic, with a function in the logic that gives each value its meaning. A *shallow embedding* uses the logic's own terms as programs. *Reflection* produces, from a definition in the logic, the deeply embedded program that corresponds to it.

2.3 WebAssembly, Talos, and the two uses of the module

WebAssembly is a low-level code format with a stack machine, typed functions, and a linear memory that grows in pages of 64 KiB [6]. Engines such as Wasmtime compile modules to machine code. A *trap* ends an execution with an error, as the instruction `unreachable` does. Leanexe states its compiler's theorem in Talos [5], a WebAssembly interpreter written by hand in Lean. Talos's function `run` gives a module's execution under a fuel argument that bounds its steps (`LeanExe/Pipeline/Aborts.lean:40`).

For the WebAssembly target, the conclusion of the module's theorem holds of an engine's run only if Talos describes the engine. Talos implements the specification's deterministic profile for floating point, in which every NaN that an instruction produces is the positive canonical NaN (`docs/design.md:198`). Lean's float model makes the same choice. Only tests relate Talos to the WebAssembly specification. On the official test suite at revision `9233a0a8`, the recorded run passes every test that it runs and skips the tests and text forms outside Talos's subset (`docs/trusted-base.md:23`).

On the processor paths, which translate the module to C or machine code, Talos serves as an intermediate language. Each back end composes the compiler's theorem with a simulation theorem whose premise is Talos's result (`docs/a64-bare-metal.md:23`). The composed statement relates the Lean function's value to the result of the C program or the machine, and it does not mention Talos's run. A departure of Talos from the standard therefore cannot make the composed statement false. The C path's trusted base still includes Talos's store and IEEE 754 operations (`docs/trusted-base.md:66`).

3 Programs in a first-order subset of Lean, proofs in all of Lean

With current proof technology, restricting programs to a small, total, first-order subset of Lean makes it practical to give each compiled program a kernel-checked correctness theorem. Specifications and proofs about the programs still use all of Lean, its mathematics library Mathlib, and other Lean developments. The subset makes the compiler's proof possible, because one proof by induction covers the meaning of a small language. The programs remain Lean definitions, so the generated theorems name those definitions and combine with any Lean theorem about them.

3.1 Programs as values of a datatype

A compiler correctness theorem has the form $\forall p,\ \mathit{run}(\mathit{compile}(p)) = \mathit{meaning}(p)$. Its quantifier ranges over the values of a type. In leanexe that type is `Prog S`, an inductive Lean datatype of typed programs over the function signatures `S` (`Verified/Source.lean:1169`). The meaning is `Expr.denote`, a Lean function from expressions to Lean values (`Verified/Source.lean:560`).

A Lean definition is not a value of `Prog S`. It is a term of the logic, and only meta code, which manipulates the syntax tree `Lean.Expr` outside the logic, can inspect it. Meta code could translate definitions to a WebAssembly module or to any other target, but no Lean theorem could quantify over its inputs. The logic has no function that gives a `Lean.Expr` its meaning. Leanexe's design document gives this absence as the reason for leanexe's architecture (`docs/design.md:171`).

Two designs give each compiled definition a theorem without quantifying over definitions. One produces, for each program, a proof that covers the whole translation, as Myreen and Owens's translator from HOL4 to CakeML does [20]. The other embeds a source language deeply, proves its compiler once, and checks for each program only the step from the definition to its embedded form.

Œuf, Hupel and Nipkow's compiler, and leanexe each embed a source language deeply and prove a compiler once. Œuf validates each reflection of a Coq definition by checking that the denotation of the reflected program is definitionally equal to the original term [19, §3]. Hupel and Nipkow translate Isabelle/HOL definitions into a deep embedding with a proof for each definition [11].

3.2 The cost of embedding all of Lean

The compiler's proof, made once for all programs, must treat every construct of the embedded language, and the meaning function must assign each construct a Lean value. For all of Lean, that

function would need a meaning inside Lean for dependent types and universes, closures and partial application, unbounded numbers, and values shared under reference counts. The compiler would also need run-time support and a proof for each of these features.

Mullen et al. write of Coq's language Gallina that "it remains an open research question whether it is possible to denote a general Gallina AST into a Gallina term, all within Gallina" [19, §3]. Lean's logic is also a dependent type theory, so this report infers that the question is open for Lean as well. The author knows no source that states the question for Lean.

Unbounded numbers and sharing have a cost at run time. Unbounded `Nat` arithmetic in compiled code needs arbitrary-precision numbers, or a trap where Lean returns the exact value (`docs/design.md:136`). Sharing needs reference counts or a garbage collector, and the compiler's proof would have to cover either one. CertiCoq-Wasm is a verified compiler from Coq's intermediate language to WebAssembly. The code that it generates never frees allocated memory [17, §3.5].

3.3 Dialect forms in place of unbounded numbers, sharing, closures, and dependent types

The dialect replaces unbounded numbers, sharing, closures, and dependent types with forms whose meaning is a small Lean function, as Table 1 lists. The resulting language has a meaning small enough that one proof by induction covers every program. `Prog.calls` proves the specification of each function by induction on the program, and for a recursive function by strong induction on the remaining call depth (`Verified/Correct.lean:20`). `Expr.code_spec` proves that the code of every expression computes its value, with a case for each construct (`Verified/AlignedExprCorrect.lean:284`).

The code of the dialect's forms needs no run-time support beyond an allocator. Every module contains the same allocator, Knuth's first-fit allocation and liberation with a free list sorted by address [12, §2.5], proved once (`docs/design.md:113`). The compiled code releases each owned object after its last use, so a module needs neither a garbage collector nor reference counts.

3.4 Programs as ordinary Lean definitions, specifications in all of Lean

A dialect program is an ordinary Lean definition, and the reflector, the part of `verified_compile` that performs reflection, reads it as written (`docs/design.md:172`). Programs of other libraries that stay in the dialect therefore compile unchanged. The generated theorem names that definition: `p.f.implements` states that the module computes `f`, with the instances that Lean synthesizes for its argument and result types.

A specification of a dialect program may use any Lean type and term, among them natural numbers, real numbers, integrals, and probability measures. It may also use any Lean library, among them Mathlib [34] and other developments such as OpenAI's published proofs [30]. Any Lean theorem about `f` combines with `p.f.implements` by rewriting, whatever the theorem imports (`docs/verified-manual.md:739`).

Leanexe's back ends state their theorems about the same Lean definition, so one specification theorem applies to every compiled form. For example, `sha256_call`, `sha256_input`, and `sha_board` compose one specification of SHA-256 with the theorems of a WebAssembly module, a bare-metal C program, and a Cortex-M7 flash image (`README.md:104`). The definition also runs as native Lean, and tests compare each compiled module with native Lean on shared cases (`docs/design.md:119`).

Table 1: Features of Lean that the dialect replaces, and the effect of each replacement on the compiler and its proof.

Lean feature	Dialect form	Effect
Unbounded `Nat` and `Int`	`UInt64` and `Int64`	The program and its module compute the same values, wraparound included, so no proof concerns overflow (`docs/verified-manual.md`:80, `docs/design.md`:153).
Values shared under reference counts	One owner per object, moves at last use, updates in place of an owned array	The allocator's `release` needs no reference count (`docs/design.md`:135).
Closures and partial application	First-order calls of listed definitions applied to all their arguments	No function values in memory and no closure conversion (`docs/verified-manual.md`:311).
`for`, `while`, `do`, and general recursion	`LeanExe.loop`, `repeatWhile`, `build`, and folds; tail recursion compiled as a loop; other recursion guarded at depth 1,000	One proof rule for each loop form. The guard turns exhaustion of the engine's stack into a defined trap (`docs/design.md`:178).
Dependent types, polymorphism, and universes	Monomorphic types from a fixed list, in a syntax typed by construction	The meaning function maps into ordinary Lean types, and no separate type checker needs trust (`Verified/Source.lean`:428).
`String`, `Option`, and arrays of arrays	Arrays of flat elements, packed bytes, `List UInt64`, and recursive records	Each type has one fixed representation in memory (`LeanExe/Pipeline/Implements.lean`:13).
Operations that other languages leave undefined	Lean's total meaning: `x / 0 = 0`, `x % 0 = x`, shifts modulo 64, the default element past the end of an array	The theorem has no premise that the source avoids undefined behavior (`docs/verified-manual.md`:142).

3.5 Kernel checks for each program, with their time and memory

The design moves work from the compiler's proof, done once, to checks that Lean's kernel performs for each program. The reflector builds each equation as a term from one lemma per Lean form (`Verified/README.md`:964). The kernel then checks the equation without unfolding a definition. Unfolding a loop over literal data would make the kernel run the loop pass by pass.

Leanexe's public knowledge base records kernel checks of 5 to 10 minutes for three demonstration modules, on a machine that it does not state (`VKB/entries/kernel-evaluation-costs/README.md`). A complete build of the library with its numerical proofs can take several hours (`README.md`:362). One demonstration's allocation proofs need more than 10 GiB of memory on one thread (`README.md`:369). These costs are minutes to hours of computation and more than 10 GiB of memory for a build, and this report calls the approach practical in that sense.

3.6 Rewrites, overflow hypotheses, float definitions, and a recursion limit

The reflector accepts only definitions in the dialect. It rejects any other definition with a message, and leanexe's compiler manual pairs each message with a rewrite (`docs/verified-manual.md`:289). For example, a `for` loop becomes a combinator, a `Nat` becomes a `UInt64`, and an array of arrays becomes one array read at computed positions.

With fixed-width arithmetic, the absence of overflow becomes a hypothesis of the specification. For example, the theorem `steps_exact` states that cached GPT-2 generation equals full inference bit for bit. It assumes dimensions below 2^{16} and 2^{32} to exclude wraparound in the `UInt64` index arithmetic (`docs/design.md`:109).

Lean's opaque float functions, such as `Float.exp`, need definitions in the dialect, which leanexe's

library provides with proofs of accuracy (docs/verified-manual.md:148). The dialect excludes conversions between `Float` and `Float32`. It limits recursion that does not compile as a loop to depth 1,000 with at most 32 values per frame.

4 The dialect

4.1 The source language and its types

An expression `Expr S Γ t` of the source language carries three indices: the signatures `S` of the functions that it may call, the context Γ of variable types, and its type `t` (Verified/Source.lean:428). Every value of the datatype is therefore well typed, reads only variables in scope, and calls only functions that exist. A program `Prog S` is a list of functions in which each function may call the functions after it in the list, which come before it in the module, and, when it is recursive, itself (Verified/Source.lean:1169). `Prog.Meaning` states that a family of Lean functions is the program's meaning (Verified/Source.lean:1213).

A Lean structure, enumeration, or sum type can appear in a dialect program through a `Flat` instance, which maps it to a tuple of words (docs/verified-manual.md:60). A recursive type can appear through a fixed record layout. Signed division of `Int64` values compiles to unsigned division of the magnitudes, so the module executes no instruction that traps on its operands (docs/verified-manual.md:144).

4.2 Control, recursion, and ownership

Control consists of `let`, `if`, `match`, folds over arrays and lists, calls of definitions listed earlier in the command, and three combinators (docs/verified-manual.md:160). `LeanExe.loop n init f` applies `f` to the indices 0 to $n-1$ from the state `init`. `repeatWhile` applies a step while a condition holds, at most a given number of times. `build n f` is the array whose element i is `f i`.

The reflector compiles tail recursion as a loop, with a decreasing rank derived from Lean's generated induction theorem (docs/verified-manual.md:281). Other recursion passes a call depth and traps at depth 1,000, with at most 32 values in a frame. Wasmtime keeps 8 bytes for each value live across a call and 16 bytes for each frame. A thousand frames of at most 32 values therefore take at most 272,000 bytes of Wasmtime's default stack of 512 KiB (docs/verified-manual.md:283).

Each parameter that holds an array, a buffer, or a tree is *borrowed* or *owned*. The reflector makes a parameter owned when the body consumes it at its last use, for example in an update or as the result, and borrowed otherwise (docs/verified-manual.md:168). An update at the last use of an owned array changes the array in place, and any other update copies it first. The theorem holds for every assignment of the two modes, so the assignment affects only the code's cost and needs no trust (docs/design.md:176).

5 The three parts of a program's theorem

A program's theorem combines three parts: a proof made once for all programs, a proof that the reflector builds for the program, and kernel evaluation of closed facts. Figure 2 shows the three parts and what each one trusts. It also shows the two uses of the module: the WebAssembly target on the left and the processor back ends on the right.

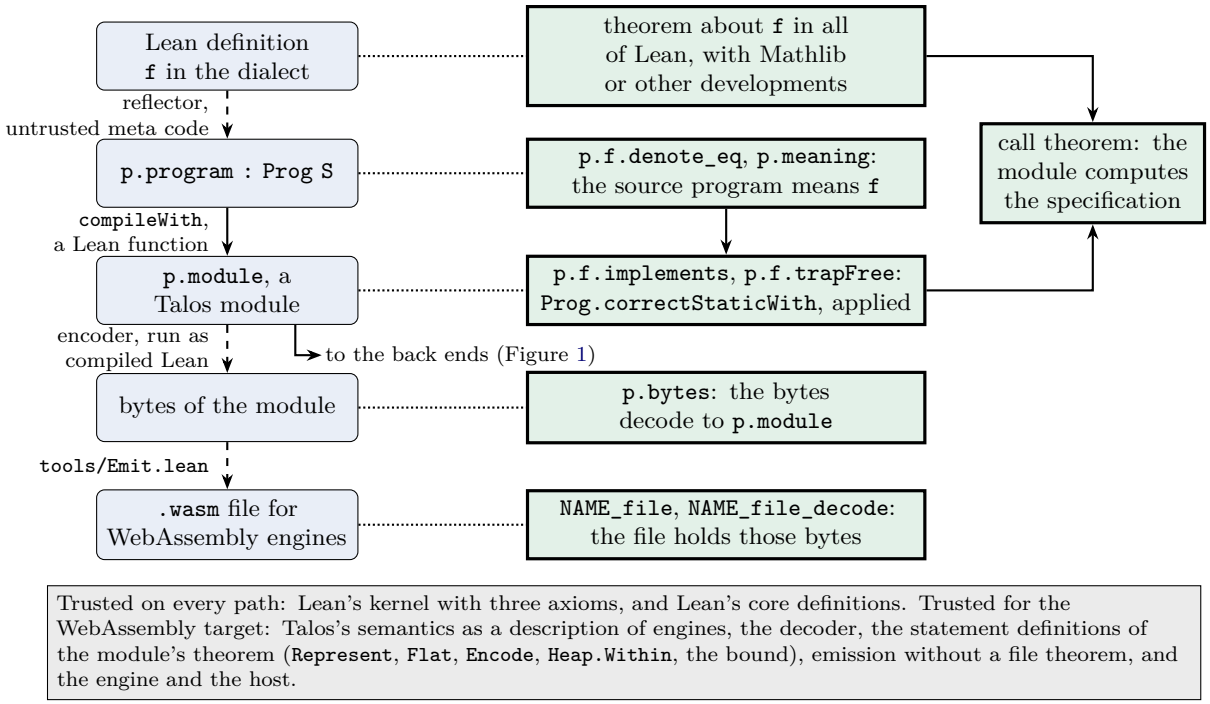

Figure 2: The pipeline for one program, with the module as the start of two paths. Rounded boxes are artifacts, boxes with heavy borders are theorems that the kernel checks, and dotted lines join each artifact to the theorem about it. Dashed arrows are steps of untrusted generators. The kernel checks the output of each generator except emission, whose output a file theorem covers where one exists. Solid arrows between artifacts are steps whose correctness a theorem states, and solid arrows between theorems are compositions. The gray box lists what the WebAssembly target takes without proof (`docs/trusted-base.md:19`), and the companion report on leanexe's targets [10] lists what the processor paths take.

5.1 The proof made once

`compileWith` is a total Lean function from a typed program, its constant tables, and the exported entries that apply its functions to those tables, to a Talos module (`Verified/Compile.lean:1628`). Because `compileWith` returns a module for every typed program, only the reflector, which is untrusted, rejects programs. `Prog.correctStaticWith` quantifies over every program `prog`, every family `funs` of Lean functions with `prog.Meaning funs`, and every literal pool that covers the program's byte literals (`Verified/Correct.lean:111`). It states that every function of the compiled module meets its specification, with flags for traps and call depth, allocation bounds, and frame conditions that Lean functions of the program compute. The proof of `Prog.correctStaticWith` is `Prog.calls`, with `Expr.code_spec` for each expression.

5.2 The proof for each program

The command `verified_compile p := [f, g, ...]` is meta code that writes each listed definition as a source function `p.f.func` (`Verified/README.md:906`). For each definition, it builds the equation `p.f.denote_eq`, which states that the source function means `f`. It builds the equation from the lemmas of `Verified/Reflect/Lemmas.lean`, one for each Lean form. It also builds the proof `p.meaning` of `Prog.Meaning p.program p.funs`. The kernel checks every equation, so a wrong

reflection fails a check and cannot produce a false theorem.

The command obtains each program's theorem from `Prog.correctStaticWith`, which it applies to `p.program`, `p.funs`, and `p.meaning` (Verified/Reflect/Command.lean:578). `ImplementsA.transferAgree` restates the result with the instances that Lean synthesizes for `f`'s argument and result types (Verified/Correct.lean:601). Leanexe thus checks each program's reflection, a form of translation validation, and verifies the compiler once.

5.3 Kernel evaluation of side conditions and bytes

The command proves closed side conditions, such as the absence of byte literals and the fit of the tables below the page cap, by `decide +kernel` (Verified/Reflect/Command.lean:556). It also proves the theorem `p.bytes`, which extends the module's theorem to the bytes that a WebAssembly engine reads. To prove `p.bytes`, the command evaluates the encoder with compiled Lean to obtain each section and function body as a literal list of bytes, and the kernel checks each part in its own theorem (Verified/README.md:1073–1078). The theorem `decode_encode` states that the encoded bytes decode to the module (LeanExe/Encoding/RoundTrip.lean:10). A file theorem, where one exists, states that the bytes that the elaborator `binary_file%` reads from the emitted file equal `encode p.module` (docs/trusted-base.md:39).

6 The theorem for a compiled function

6.1 The statement ImplementsA

For a program without byte literals, `p.f.implements` states `ImplementsA aborts p.module k f`, where `k` is the index of `f`'s function in the module. Listing 1 gives the definition of `ImplementsA`. The definition quantifies over every host environment, store, heap, and input `x`. It covers every input `x`, because the dialect has no undefined behavior.

```
def ImplementsA [Represent α] [Represent β] (aborts : Bool) (m : Module) (entry : Nat)
    (f : α → β) (Pre : α → Heap → Store Unit → Prop)
    (Post : α → Heap → Store Unit → Heap → Store Unit → Prop) : Prop :=
  ∀ (env : HostEnv Unit) (store : Store Unit) (heap : Heap) (params : List Value) (x : α),
    heap.At store → Pre x heap store → Represent.borrowed heap store params x →
    Separate store (Represent.moves store params x) (Represent.reads store params x) →
    store.memoryCap m 0 ≤ 65535 →
    Runs aborts env m entry store params.reverse fun final values =>
      ∃ heap' : Heap, heap'.At final ∧ Represent.owned heap' final values.reverse (f x) ∧
        final.memoryCaps = store.memoryCaps ∧
        (∀ r, heap.Region r → 0 < r.2 → Apart store (Represent.moves store params x) r →
          (∀ a, r.1 ≤ a → a < r.1 + r.2 → final.mem.bytes a = store.mem.bytes a) ∧
            heap'.Region r ∧ Represent.outside final values.reverse (f x) r) ∧
        Post x heap store heap' final
```

Listing 1: The definition of `ImplementsA` (LeanExe/Pipeline/Implements.lean:648).

The premises of `ImplementsA` are that the heap meets the allocator's invariant (`heap.At store`), that the precondition `Pre` holds, that the arguments represent `x`, that the blocks that the call consumes are separate from the regions that it reads, and that the memory's cap is at most 65,535 pages. Under these premises, every run with enough fuel returns words that represent `f x` in an owned result. The run keeps the allocator invariant and the memory cap and meets the postcondition `Post`, and every heap region disjoint from the consumed blocks keeps its bytes and lies outside the result. `Runs` states termination in Talos: some fuel N exists such that every run with at least N steps of fuel returns or, when `aborts` is true, traps at `unreachable` (LeanExe/Pipeline/Aborts.lean:40).

6.2 Traps and the trap-free theorem

When `aborts` is true, a run may trap at `unreachable` (docs/verified-manual.md:13). The code traps only when an allocation finds no memory, an array would hold 2^{29} words or more, a byte builder requests 2^{32} bytes or more, or a recursion reaches depth 1,000. A statement that allows a trap holds of a module that always traps, so a theorem of that form excludes no failure. The design keeps the trap in `p.f.implements` in place of a premise about free memory, because such a premise does not compose across calls (docs/design.md:142).

`p.f.trapFree` states `ImplementsA false` under two premises (docs/verified-manual.md:15). The first, `Heap.Within`, states that `top`, the allocator's bump pointer, is at least `p.f.bound x` bytes below the memory cap. The second, for a function whose code takes the call depth, is a frames condition: every call of the run to a function of this kind happens below depth 1,000. Under these premises, the call returns `f x` without a trap and raises `top` by at most the bound. `p.f.bound_eq` states the bound as a Lean term with the structure of the definition, so a user proves a closed form by rewriting (docs/verified-manual.md:755).

6.3 Representations and generated declarations

The class `Represent` defines how a value appears as WebAssembly values and heap data (LeanExe/Pipeline/Implements.lean:13). It covers borrowed and owned values, the blocks that a value occupies, and the regions that a call reads. A structure, enumeration, or sum type appears in a statement through its `Flat` instance, and a recursive type through its `Encode` instance (LeanExe/Pipeline/Implements.lean:106, LeanExe/Pipeline/Implements.lean:435).

The `Flat` and `Encode` instances are part of what the theorem states, and a reader must accept them as the intended representation (docs/verified-manual.md:32). For a recursive type, the command `verified_recursive_type`, which derives the declarations that `verified_compile` needs for the type, proves `EncodeWellFormed` (docs/verified-manual.md:205). That class excludes the layouts for which a statement would hold of every module (docs/verified-manual.md:40).

Besides the theorems for each function, `verified_compile` adds the declarations `p.initial` and `p.bytes` (docs/verified-manual.md:469). `p.initial` states that the store in which the module starts meets the allocator invariant, caps memory at 65,535 pages, and holds the tables. `p.bytes` states that the module's bytes decode to the module, together with every `p.f.implements`.

7 Specifications in full Lean

7.1 Euclid's algorithm from a dialect definition to a module export

Euclid's algorithm is the shortest example in leanexe's library of a theorem that relates a module export to a mathematical specification. The program is a dialect definition over words (Library/Numerics/Arithmetic/Gcd/Program.lean). The listing omits its proof of termination.

```
def gcd (a b : UInt64) : UInt64 :=
  if b = 0 then a else gcd b (a % b)
termination_by b.toNat
```

The specification uses `Nat.gcd` on natural numbers, a type outside the dialect (Library/Numerics/Arithmetic/Gcd/Spec.lean). An ordinary Lean proof by well-founded induction shows that the program computes it (Library/Numerics/Arithmetic/Gcd/Proof.lean).

```
def expected (x : UInt64 × UInt64) : UInt64 :=
  UInt64.ofNat (Nat.gcd x.1.toNat x.2.toNat)
theorem gcd_eq (a b : UInt64) : gcd a b = expected (a, b)
```

Table 2: Specification theorems at the pinned commit.

Theorem	Statement
`gcd_call` (`Library/Numerics/Arithmetic/Gcd/Compiled.lean:22`)	The export returns `Nat.gcd` of its arguments from every store.
`sha256_call` (`Library/Hashing/Sha256/Bytes.lean:29`)	The export returns the digest of a transcription of FIPS 180-4 [21].
`gaussian_chiSquare` (`Demos/Gate/Gaussian.lean:199`)	With Mathlib's measure theory, the squared Mahalanobis distance of a Gaussian innovation has the chi-square law.
`pc_enclosure` (`Demos/Conjunction/Soundness.lean:750`)	An accepted interval contains the probability of collision, an integral over the reals, for the real values of the inputs.
`propagate_newton` (`Demos/Avoidance/TwoBody.lean:339`)	The two-body propagation solves Newton's equation over the reals at every time.

One command compiles the program and generates `compiled.gcd.pure`, the theorem for a function with scalar arguments and results. A short proof composes that theorem with `gcd_eq` (`Library/Numerics/Arithmetic/Gcd/Compiled.lean`).

```
verified_compile compiled := [gcd]
theorem gcd_call (env : HostEnv Unit) (store : Store Unit) (a b : UInt64) :
    Runs false env compiled.module 2 store [.i64 b, .i64 a] fun final values =>
      final = store ∧ values = [.i64 (expected (a, b))]
```

The theorem `gcd_call` concerns function 2 of the module, the export `gcd`. When called from any store with the words `a` and `b`, the function returns the word of their greatest common divisor without a trap and leaves the store unchanged. The compiler turns the tail recursion into a loop. The command `#print axioms gcd_call` reports `propext`, `Classical.choice`, and `Quot.sound` (`docs/verified-manual.md:838`).

7.2 Specification theorems in leanexe's library and demonstrations

Table 2 lists theorems from leanexe's library and demonstrations whose statements use mathematics outside the dialect. For the first two, a named theorem composes the property with the module's theorem. For the others, the demonstration's guide lists the generated theorems of the compiled functions, such as `compiled.pc.implements` for the program that encloses the probability of collision. Each property composes with them by the method of Section 7.4.

7.3 Programs that use OpenAI's theorems

The exaspec examples page [3] describes six programs whose proofs use theorems that OpenAI proved in Lean and published in 2026 in its mathematics repository [30]. According to that page, each program computes or checks the hypotheses of its theorem, and Lean proofs connect the compiled code through that check to the theorem. The programs' sources are in a private repository that this report does not link. Its build checks their proofs against the pinned leanexe commit.

The program for certified residual reduction for linear systems uses Crouzeix's inequality with the constant 2, from OpenAI's paper [25] and its Lean proof. The program for persistence of reaction networks uses the boundedness and persistence of weakly reversible mass-action systems, from OpenAI's paper [27] and its Lean proof. The program for bound-state energies of layered potentials uses the sharp one-dimensional Lieb–Thirring inequality, from OpenAI's paper [26] and its Lean proof. The program for the classical capacity of a decaying qubit uses a theorem on the generalized amplitude-damping channel, from OpenAI's paper [24] and its Lean proof. That theorem states that the channel's classical capacity equals the maximum of a function of one variable.

The program for second neighborhoods in oriented graphs uses a theorem from OpenAI's paper [28] and its Lean proof. That theorem states that every nonempty finite oriented graph has a vertex with at least as many second neighbors as first neighbors. The program for the least energy of point arrangements in the plane uses a theorem from OpenAI's paper [29] and its Lean proof. That theorem states that, for every completely monotone potential, the triangular lattice has the least energy among arrangements of density one.

7.4 Proofs about compiled programs

A proof about a compiled program takes three steps, as the compiler manual describes for a program derived from a request in English (`docs/verified-manual.md:1045`).

1. An ordinary Lean proof shows that the program equals an independent specification. It uses the dialect's lemmas, such as `loop_induction`, which proves a property of a loop from an invariant on the loop's state after k passes (`LeanExe/Dialect/Lemmas.lean:11`).
2. The proof composes that equation with `p.f.trapFree` into `Verified.CallSpec` (`Verified/Call.lean:72`). That theorem concerns a call from a store that meets the allocator invariant, holds the tables, and has room for the generated bound below the memory cap.
3. Rewriting with `p.f.bound_eq` gives the bound as a closed form in the sizes of the inputs. For example, `pushSquare_bound` proves that a push onto an array of n words needs at most $16(n+1)+80$ bytes (`Verified/Examples/Tables.lean:39`).

Kernel evaluation proves concrete facts at a cost that depends on the form of the evaluated terms. Leanexe's public knowledge base records the forms that keep that cost low (`VKB/entries/kernel-evaluation-costs/README.md`).

8 Proofs, tests, and assumptions

Table 3 lists the proved, tested, and assumed items and the paths that each one enters. The companion report on leanexe's targets [10] lists the items that the processor paths add. The author read the proofs at the pinned commit and did not rebuild them.

9 Related work

Table 4 compares leanexe with verified compilers, proof-producing translators, and verifiers whose compilers no theorem covers. Œuf and the compiler of Hupel and Nipkow, described in Section 3, divide the work as leanexe does, and both appeared in 2018. Œuf compiles a simply typed subset of Gallina through verified passes to CompCert's Cminor [19, §4]. Hupel and Nipkow's compiler produces CakeML syntax, with a theorem of partial correctness [11].

Leanexe's source subset includes arrays updated in place, packed bytes, IEEE 754 floats, and recursive records. Leanexe also generates allocation bounds and theorems of total correctness, and Lean's kernel checks the encoded bytes. For the WebAssembly target, leanexe's theorems hold in Talos, which tests relate to the WebAssembly specification. CertiCoq-Wasm's proofs use WasmCert-Coq, a mechanized semantics of the WebAssembly standard, and leave the conversion to the binary format unverified [17, §1.2, §3.8].

Rinard's Axon [33] is a verified compiler written in Lean. It compiles a language inspired by Fortran, with 64-bit integers, floats, Booleans, one-dimensional arrays, `while`, and `if`, to AArch64 assembly [33, §5]. The source is a separate language whose big-step semantics, defined in Lean, serves as the specification of a program. Axon's theorems state that every behavior of

Table 3: Proved, tested, and assumed items, with the paths that each one enters.

Kind	Item	Paths	Source
Proved	`Prog.correctStaticWith`: every typed program's module computes its meaning	Every path	`Verified/Correct.lean`:111
	For each program: `p.f.denote_eq`, `p.meaning`, `p.f.implements`, `p.f.bound_eq`, and `p.f.trapFree`	Every path	`Verified/Reflect/Command.lean`:500
	Lean's float addition, subtraction, multiplication, division, and square root equal Talos's `IEEE64` and `IEEE32` on bit patterns	Every path	`docs/design.md`:191
	`alloc` and `release` keep the allocator invariant	Every path	`docs/design.md`:113
	`p.initial`, `p.bytes`, `decode_encode`, and `encode_correct`	WebAssembly	`LeanExe/Encoding/RoundTrip.lean`:10
Tested	A check of the axioms of every declaration of the libraries	Every path	`tests/verified/AllAxioms.lean`
	Each module against native Lean in Wasmtime, with counts of the blocks that each call leaves allocated	WebAssembly	`docs/design.md`:119
	Talos against the WebAssembly test suite, with the result of Section 2.3	WebAssembly	`docs/trusted-base.md`:23
	The decoder against the test suite's valid modules in its subset	WebAssembly	`docs/trusted-base.md`:27
	Recursion at depth 999 with frames of 32 values on Wasmtime 44 for AArch64	WebAssembly	`docs/trusted-base.md`:33
Assumed	Lean's kernel at 4.34.1 with three axioms, and Lean's core definitions. No independent kernel has rechecked the proofs.	Every path	`docs/trusted-base.md`:7
	Talos describes what an engine does, within the deterministic floating-point profile	WebAssembly	`docs/trusted-base.md`:23
	The decoder gives bytes their meaning, and compiled Lean writes them when no file theorem covers the file	WebAssembly	`docs/trusted-base.md`:27
	Hosts place arguments as `Represent` states, instantiation matches Talos's initial store, and the stack holds 1,000 frames	WebAssembly	`docs/trusted-base.md`:29
	The statement definitions of the module's theorem mean what a reader takes them to mean	WebAssembly	`docs/trusted-base.md`:30

the generated assembly is a behavior of the source program, that the only behaviors are a halt, an error, or divergence, and that every well-formed program compiles [33, §5.3]. Leanexe compiles Lean definitions, so a program's specification can be any Lean theorem about the definition.

Axon checks its optimizations by credible compilation, in which a verified checker checks a certificate for each run [33, §2.1]. It takes as axioms that the source and machine floating-point operations agree, and it models fused multiply-add as two roundings [33, §5.1]. Leanexe proves Lean's float arithmetic equal to Talos's, and its AArch64 back end takes equations between Arm's floating-point functions and Talos's as trusted hypotheses (`README.md`:302).

CompCert, CakeML, and CertiCoq accept larger languages than leanexe's dialect [1, 7, 13]. Rupicola produces low-level code and its proof together by proof search for each program [31]. Bedrock2's integration proof covers more of a system than leanexe's theorems: one theorem relates network packets to actuator writes across application, drivers, compiler, and processor [9]. Translation validation checks each run of an unverified compiler, as Necula's validator does for passes of GCC [22].

Table 4: Leanexe and related systems, with the scope and trusted base of each correctness theorem. The prover that checks each system's proofs is in parentheses. The trusted base lists what the correctness statement takes without proof, as each primary source describes it.

System	Source and target	Scope of the correctness theorem	Trusted base
leanexe (Lean)	A first-order subset of Lean to the WebAssembly module, the input of the C, AArch64, and Cortex-M7 back ends [10]	Every function of every typed program computes its Lean definition or traps at `unreachable`, with no trap under the generated bound. For the WebAssembly target, the bytes decode to the module	Lean's kernel and three axioms; for the WebAssembly target, Talos (tested), the decoder, statement definitions, emission, and hosts
Axon [33] (Lean)	A language inspired by Fortran, with integers, floats, Booleans, and one-dimensional arrays, to AArch64 assembly	The assembly has only source behaviors (halt, error, divergence), and every well-formed program compiles. Optimizations checked per run	Lean; source and assembly semantics; parser and printer; floating-point axioms; assembler, linker, OS, hardware
CompCert [1, 16] (Coq)	CompCert C (C99 with listed exceptions) to PowerPC, ARM, RISC-V, and x86 assembly	For a source program that does not go wrong, every behavior of the compiled code is a source behavior	Coq, extraction, OCaml; unverified parsing, type checking, assembling, and linking; C and assembly semantics
CakeML [13] (HOL4)	A subset of Standard ML, as a read-eval-print loop in x86-64 machine code	The machine code's output is allowed by the semantics, or it stops with an out-of-memory error	HOL4, the CakeML semantics, and the x86-64 model
Hupel and Nipkow [11] (Isabelle)	Isabelle/HOL recursion equations to CakeML syntax	Proof-producing translation into a deep embedding, then a verified chain; partial correctness against CakeML's big-step semantics	Isabelle and the CakeML semantics
Œuf [19] (Coq)	Simply typed Gallina to Cminor, then CompCert to x86	Reflection validated per program, verified compilation to Cminor, and calls correct if memory suffices	Coq's checker and CompCert's unverified parts
CertiCoq [7] (Coq)	Type-erased Gallina to Clight, then CompCert	Preservation of meaning for each verified phase. No end-to-end statement as of 2025 [17]	Coq, MetaCoq reification, CompCert, and a garbage-collected runtime
CertiCoq-Wasm [17] (Coq)	CertiCoq's λ_{ANF} to WebAssembly	A related value or exhaustion of memory, in WasmCert-Coq's semantics	Coq, WasmCert-Coq, CertiCoq's earlier stages, and the conversion of the syntax tree to binary
Low* [32] (F*, Z3)	A first-order stateful subset of F* to C through KreMLin	A proof by hand that a formal model of the translation preserves execution traces	F* and Z3; the hand proofs; KreMLin; the C compiler
Lean 4 [8]	All of Lean to C through an intermediate representation	No compiler theorem. Proofs by native evaluation carry dedicated axioms [4]	The compiler, the C compiler, and the runtime
Dafny [2, 15] (Boogie, Z3)	Dafny to C#, Java, JavaScript, Go, Python, and C++	The program meets its specification. No theorem covers compilation [23]	The verifier, Z3, and each back end with its toolchain
Verus [14] (Z3)	Rust with specifications and proofs, erased before `rustc`	Verified properties of the Rust source, and no compiler theorem	Verus, Z3, and the Rust compiler

10 Limitations

The dialect excludes `Nat`, `String`, `Option`, closures, `for` loops, arrays of arrays, partial application, and universe polymorphism (`docs/verified-manual.md`:80). It also excludes conversions between `Float` and `Float32` and recursion deeper than 1,000 frames (`docs/design.md`:200). The module's

theorem and the C theorem over a float parameter cover no NaN argument other than Lean's canonical NaN (`docs/design.md:252`).

For the WebAssembly target, the theorems hold in Talos's semantics, which bounds no call depth and grows memory whenever the new page count is within the cap (`docs/design.md:251`). An engine may instead fail to grow memory for reasons of its own. No proof relates Talos to the WebAssembly specification. The theorems do not cover engine instantiation, the host programs that run the modules, or results computed in a browser, which does not implement the deterministic floating-point profile (`docs/verified-manual.md:50`, `docs/design.md:198`).

The module's theorem states that some fuel suffices, and it bounds neither the fuel nor the running time (`LeanExe/Pipeline/Aborts.lean:40`). The allocation bound does not count the reuse of freed blocks, so a failed bound premise says nothing about the program (`docs/verified-manual.md:766`). Every theorem's meaning depends on definitions that a reader must accept, among them `Represent`, the `Flat` and `Encode` instances, and `Heap.Within`.

Checking the whole library takes hours, and some proofs need more than 10 GiB of memory (Section 3.5). No independent kernel has rechecked the proofs. No measurement at the pinned commit shows how proof effort divides among the parts of leanexe, so this report gives none. This report states the properties of the six programs that use OpenAI's theorems as the exaspec examples page states them. The programs' sources are private.

References

[1] The CompCert C language, 2026. URL https://compcert.org/compcert-C.html. Read 27 November 2026.

[2] Dafny reference manual, 2026. URL https://dafny.org/latest/DafnyRef/DafnyRef. Read 27 November 2026.

[3] exaspec examples. Web page, 2026. URL https://exaspec.com/examples. Read 27 November 2026.

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

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

[6] WebAssembly core specification. Living specification, 2026. URL https://webassembly.github.io/spec/core/.

[7] Abhishek Anand, Andrew W. Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau, and Matthew Weaver. CertiCoq: A verified compiler for Coq. In *The Third International Workshop on Coq for Programming Languages (CoqPL)*, 2017. URL https://www.cs.princeton.edu/~appel/papers/certicoq-coqpl.pdf.

[8] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. https://doi.org/10.1007/978-3-030-79876-5_37.

[9] Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. Integration verification across software and hardware for a simple embedded system. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2021. https://doi.org/10.1145/3453483.3454065.

[10] 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.

[11] Lars Hupel and Tobias Nipkow. A verified compiler from Isabelle/HOL to CakeML. In *Programming Languages and Systems (ESOP 2018)*, volume 10801 of *Lecture Notes in Computer Science*, pages 999–1026. Springer, 2018. https://doi.org/10.1007/978-3-319-89884-1_35.

[12] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, third edition, 1997. URL https://www.informit.com/store/art-of-computer-programming-volume-1-fundamental-algorithms-9780201896831.

[13] 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.

[14] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust programs using linear ghost types. *Proceedings of the ACM*

on Programming Languages, 7(OOPSLA1):Article 85, 2023. https://doi.org/10.1145/3586037. Read as arXiv:2303.05491v2.

[15] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16)*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. https://doi.org/10.1007/978-3-642-17511-4_20.

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

[17] Wolfgang Meier, Martin Jensen, Jean Pichon-Pharabod, and Bas Spitters. CertiCoq-Wasm: A verified WebAssembly backend for CertiCoq. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, 2025. https://doi.org/10.1145/3703595.3705879.

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

[19] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. Œuf: Minimizing the Coq extraction TCB. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pages 172–185, 2018. https://doi.org/10.1145/3167089.

[20] Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming*, 24(2–3):284–315, 2014. https://doi.org/10.1017/S0956796813000282.

[21] National Institute of Standards and Technology. Secure hash standard (SHS). Federal Information Processing Standards Publication 180-4, 2015.

[22] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI)*, pages 83–94, 2000. https://doi.org/10.1145/349299.349314.

[23] Daniel Nezamabadi and Magnus O. Myreen. Baking for Dafny: A CakeML backend for Dafny. Dafny Workshop at POPL 2025, arXiv:2501.05111, 2025.

[24] OpenAI. Classical capacity and entropy inequalities for generalized amplitude damping. Preprint dated 24 September, 2026. URL https://github.com/openai/math/blob/3344da59991ed94f9ba711217b29c1d760badfa6/preprints/Classical-capacity-and-entropy-inequalities-for-generalized-amplitude-damping-September-24-2026/paper.pdf.

[25] OpenAI. A direct proof of the complete Crouzeix inequality. Preprint dated 26 September, 2026. URL https://github.com/openai/math/blob/3344da59991ed94f9ba711217b29c1d760badfa6/preprints/A-direct-proof-of-the-complete-Crouzeix-inequality-September-26-2026/paper.pdf.

[26] OpenAI. Sharp one-dimensional Lieb–Thirring constants. Preprint dated 23 September, 2026. URL https://github.com/openai/math/blob/3344da59991ed94f9ba711217b29c1d760badfa6/preprints/Sharp-One-Dimensional-Lieb-Thirring-Constants-September-23-2026/paper.pdf.

[27] OpenAI. Boundedness and persistence of weakly reversible mass-action systems. Preprint dated 25 September, 2026. URL https://github.com/openai/math/blob/3344da59991ed94f9ba711217b29c1d760badfa6/preprints/Boundedness-and-persistence-of-weakly-reversible-mass-action-systems-September-25-2026/paper.pdf.

[28] OpenAI. A proof of Seymour's second-neighborhood conjecture. Preprint dated 23 September, 2026. URL https://github.com/openai/math/blob/3344da59991ed94f9ba711217b29c1d760badfa6/preprints/A-proof-of-Seymours-second-neighborhood-conjecture-September-23-2026/paper.pdf.

[29] OpenAI. An atomic certificate for triangular-lattice universal optimality. Preprint dated 26 September, 2026. URL https://github.com/openai/math/blob/3344da59991ed94f9ba711217b29c1d760badfa6/preprints/An-atomic-certificate-for-triangular-lattice-universal-optimality-September-26-2026/paper.pdf.

[30] OpenAI. Mathematics repository. Git repository, 2026. URL https://github.com/openai/math/tree/3344da59991ed94f9ba711217b29c1d760badfa6.

[31] Clément Pit-Claudel, Jade Philipoom, Dustin Jamner, Andres Erbsen, and Adam Chlipala. Relational compilation for performance-critical applications. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 918–933, 2022. https://doi.org/10.1145/3519939.3523706.

[32] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in F*. *Proceedings of the ACM on Programming Languages*, 1 (ICFP), 2017. https://doi.org/10.1145/3110261. Read as arXiv:1703.00053v6.

[33] Martin C. Rinard. Testing, credible compilation, and verification in the Axon verified compiler in Lean and Claude Code. In *Proceedings of the 2026 ACM SIGPLAN International Workshop on Principles of Agentic Engineering (PAgE)*, 2026. https://doi.org/10.1145/3819802.3820579. Read as arXiv:2605.01660v3.

[34] The Mathlib Community. Mathlib. Git repository, tag v4.34.0-rc2, 2026. URL https://github.com/leanprover-community/mathlib4/tree/85e3a25e006c35636f0e53b0e9296caca2685bc0.