Leanexe's targets: composed theorems in stated semantics, with exact trusted bases

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. The C and machine-code translations start from that module as an intermediate representation, and WebAssembly engines run it as a final representation. GPU kernels in WGSL and PTX come from a separate kernel language. For the C, AArch64, and Cortex-M7 targets, leanexe composes the module's theorem with a simulation theorem for that target. Each composed theorem relates the Lean function to the target's semantics and does not mention the WebAssembly interpreter's run. This report states each target's theorem, the semantics in which it holds, and the items that it takes without proof. The step from leanexe's Lean copy of CompCert's Clight to CompCert's own Clight is machine-checked for eleven AArch64 programs. For every other program, review and tests support that step. Lean proofs compare each instruction form of the AArch64 and Cortex-M7 machine models with a trace of Arm's A-profile specification from the Isla tool. For floating-point forms, the proofs take as hypotheses equations that state the results of Arm's floating-point functions. The GPU kernels' theorems hold under partial statements of the Vulkan and PTX memory models. No theorem bounds running time, and no machine model is proved equal to Arm's specification.

1 Introduction

The correctness proof of a compiler states that the code the compiler writes computes what the source program computes. What the statement says about a running machine depends on two things. The first is the semantics in which the proof describes the target: a definition, inside the logic, of what the target's code does. The second is the trusted base: the components that a user of the theorem accepts without proof, such as a model of the processor or the program that loads a file. A reader who applies the theorem to a running machine relies on both, so a report on a compiler's targets has to state both exactly.

Leanexe [20] compiles programs written in a first-order subset of Lean, a language in which one writes both programs and proofs. For each compiled program, Lean's kernel checks a theorem that the compiled code computes the Lean definition. The compiled code runs on AArch64 and x86-64 processors, on an Arm Cortex-M7 microcontroller, on GPUs, and in WebAssembly engines (`README.md:3`).

Leanexe writes AArch64 and Thumb-2 machine code, C for the CompCert compiler, WGSL and PTX kernels, and WebAssembly modules, including commands for the WebAssembly System Interface (WASI). CompCert compiles the C for x86-64 and AArch64 under an operating system

and for bare-metal AArch64 and Cortex-M7 (`README.md:24`). The companion report on leanexe's verified compiler [13] describes that compiler, which produces one WebAssembly module for each program, and the theorem that each module receives. The GPU kernels come from a separate kernel language, whose translations to WGSL and PTX have theorems of their own (`docs/design.md:236`).

The WebAssembly module that the verified compiler produces serves two roles. WebAssembly engines run it as a final representation, and the C, AArch64, and Cortex-M7 back ends start from it as an intermediate representation (`docs/native-translator.md:25`, `docs/a64-bare-metal.md:23`, `docs/m7-bare-metal.md:32`). The module's theorem holds in Talos, a WebAssembly interpreter written in Lean [4]. For each of the three back ends, leanexe proves a simulation theorem, which relates runs in two semantics. The theorem's premise is Talos's run of the module, and its conclusion is a run in the target's semantics.

For its targets, leanexe adds the following results to the verified compiler of the companion report [13].

1. Simulation theorems from the module to a Lean copy of CompCert 3.18's Clight and to Lean models of an AArch64 machine, a Linux process, and the Cortex-M7 microcontroller, each composed with the module's theorem (Sections 2, 4, 6, and 8).
2. Kernel checks of the written machine-code files (Section 2.3).
3. Whole-program theorems for bare-metal C programs, with a Rocq proof that transfers them to CompCert's assembly and a machine-checked step between the two provers for eleven AArch64 programs (Section 5).
4. Lean proofs that each instruction form of the AArch64 and Cortex-M7 models agrees with Isla's trace of Arm's A-profile specification (Sections 7 and 8).
5. Theorems for GPU kernels about every consistent execution under statements of the Vulkan and PTX memory models (Section 9).

This report lists the trusted base of each target item by item (Table 1) and states separately what is proved, what is tested, and what is assumed (Section 11).

This report describes leanexe at commit `4556fa84` of 26 November 2026, with Lean 4.34.1, Mathlib at `85e3a25` [27], CompCert 3.18 [2], and Talos at `87e3aa5e`. Leanexe's build files, `lakefile.lean` and `lake-manifest.json`, pin the repository `jsmorph/talos` [4] at that commit. That repository is a fork of `cajal-technologies/talos` [5], which leanexe's README links (`README.md:22`). Every link to a leanexe file points to commit `4556fa84`. Where leanexe's documents at that commit contradict leanexe's sources, this report states what the sources show.

2 Composition of the module's theorem with each target's theorem

For the C, AArch64, and Cortex-M7 targets, leanexe composes the module's theorem with a simulation theorem for that target. Each composed theorem relates the Lean function to the target's semantics and does not mention Talos's run. Each composed theorem trusts Lean's kernel, the definitions that its statement names, and the items that Table 1 lists for its target.

2.1 The module as intermediate and final representation

Talos is a WebAssembly interpreter written by hand in Lean. Its function `run` gives a module's execution under a fuel argument, a number that bounds the execution's steps (`LeanExe/Pipeline/Aborts.lean:40`). For a compiled function `f` of a program `p`, the theorem `p.f.implements` concerns Talos's run of the module `p.module` from every store that meets the allocator's invariant, with arguments that represent `x`. The theorem states that the run returns words that represent `f x`

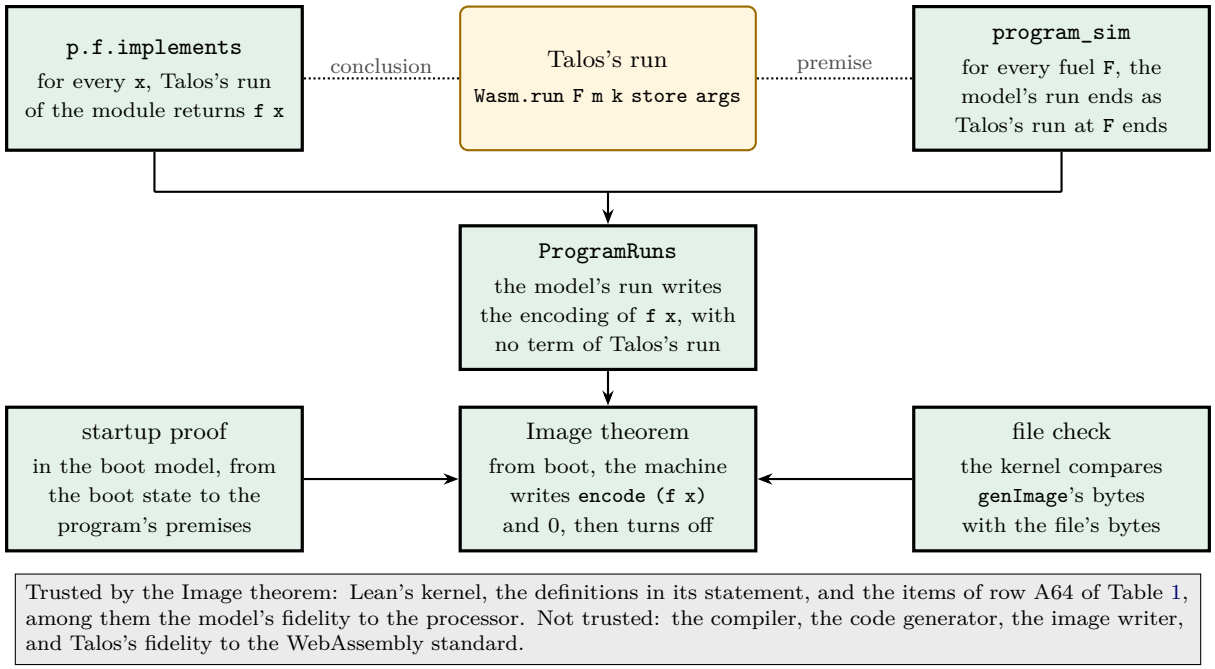

Figure 1: The composition of theorems for an AArch64 Image (code A64). Boxes with heavy borders are theorems that Lean's kernel checks. The orange box is the term that the two upper theorems share: the conclusion of `p.f.implements` and the premise of `program_sim`. Solid arrows are compositions. `ProgramRuns` follows from the two upper theorems at a fuel of at least the module theorem's bound, so its statement contains no term of Talos's run. Source: `docs/trusted-base.md:125`.

or traps at the instruction `unreachable` [13]. The theorem `p.f.trapFree` excludes the trap under premises on free memory and call depth [13].

When a WebAssembly engine runs the module, the conclusion of `p.f.implements` holds of the engine's run only if Talos describes the engine (`docs/trusted-base.md:19`). Only tests relate Talos to the WebAssembly specification [6] (`docs/trusted-base.md:23`). The C translator and the AArch64 and Cortex-M7 code generators read the same module. The compiler's theorem therefore covers every function that each of them translates, and none of them needs a proof about the source language (`docs/a64-bare-metal.md:29`). Each accepts only part of WebAssembly, and Sections 4, 6, and 8 state the parts.

2.2 Elimination of Talos's run

Figure 1 shows the composition for an AArch64 Image, a bare-metal program that Section 6 describes. The module's theorem gives, for every argument `x`, a fuel `N`. At every fuel `F` of at least `N`, Talos's run `Wasm.run F m k store args` returns words that represent `f x` [13].

The simulation theorem `program_sim` takes Talos's run as a parameter (`LeanExe/A64/Proof/Program.lean:1470`). It states that, for every fuel `F`, the machine model's run of the generated code ends with the outcome of Talos's run at `F`. That outcome is a return with the result's bytes written to the UART, or a trap with its status. The composition chooses a fuel at or above `N`, so the machine's run writes the encoding of `f x`. The composed statement, `ProgramRuns`, names the module, the entry, the arguments, the bytes written, and the machine model, and no term of Talos's run occurs in it (`LeanExe/A64/Proof/ImageTheorem.lean:227`).

```
theorem gcd (h : Hw) (B : UInt64) (hB : B.toNat % 4096 = 0) (hlo : 0x40000000 ≤ B.toNat)
    (hhi : B.toNat + gcdImageSize ≤ 0x80000000) (b : Boot)
    (hm : ∀ j < gcdSize, b.core.mem (B + UInt64.ofNat j) = byteAt gcdBytes j)
    (hpc : b.core.pc = B) (hoff : b.mmuOff) :
    ∃ b', BootReach h (protectedBytes B gcdLength) (fetchFrom b.core.mem B (codeWords gcdLength))
        b b' ∧
      fetchFrom b.core.mem B (codeWords gcdLength) b'.core.pc = some (encode .hvc0) ∧
      b'.core.x 0 = 0x84000008 ∧ b'.uartDone = true ∧
      b'.core.out = b.core.out ++
        LeanExe.Native.encodeWord (TestSupport.Native.Gcd.gcd 1071 462) ++ [0]
```

Listing 1: The theorem of the GCD Image (tests/TestSupport/A64/ImageFiles.lean:94), without its proof. `gcdBytes` is `file_nat% "gcd.img"`, the file's bytes as one number.

The module's theorem and `program_sim` both hold of `Wasm.run` as Lean defines it. If Talos departed from the WebAssembly standard on some instruction, both theorems would still hold of Talos's definition, and so would their composition (docs/a64-bare-metal.md:23). Talos's fidelity to the standard is therefore outside the trusted base of every processor target. For these targets, Talos serves as an intermediate language whose semantics only the proofs use (docs/native-translator.md:25, docs/m7-bare-metal.md:32).

Some of Talos's definitions remain in the statements of the composed theorems. `ProgramRuns` requires that the area of the module's globals and memory hold the bytes of Talos's initial store. The Image's theorem discharges that premise from the file's bytes and the proof of the startup routine (docs/a64-bare-metal.md:23). The C theorem relates the C memory to a Talos store, so Talos's store and memory definitions remain in what the C target trusts (docs/trusted-base.md:70).

Listing 1 gives the theorem of the Image that computes `gcd 1071 462`. The theorem starts from a boot state whose memory holds the bytes of `gcd.img` at a base `B`. From that state, the machine writes the eight bytes of the Lean value `gcd 1071 462` and the status byte 0, and it reaches the PSCI SYSTEM_OFF call, which turns the machine off. A reader of the statement needs the boot model, the elaborator that reads the file's bytes, and the Lean definition of `gcd`, and no definition of Talos.

2.3 Generators outside the trusted base

The reflector, which produces the deeply embedded program of each Lean definition, runs as compiled Lean code, as do the compiler, the translator, the code generators, and the image writers. The kernel checks what these programs produce (docs/trusted-base.md:15). For the AArch64 Images, the Linux executables, and the Cortex-M7 flash images, the kernel also checks the written file.

To check a written file, an elaborator reads the file's bytes as one number, and the kernel evaluates the generator on the program's module, entry, and arguments and compares the result with that number (docs/a64-bare-metal.md:77). The theorem's premise states that memory holds the file's bytes, so the trusted base includes neither the compiled generator nor the code that wrote the file. These targets therefore trust no compiler, assembler, or linker (docs/trusted-base.md:125, docs/trusted-base.md:164).

2.4 Semantics, theorem forms, and trusted bases of the targets

Table 1 gives, for each target, the semantics of its theorem, the form of the theorem, and what the theorem trusts beyond the items that every target shares. The codes are those of leanexe's table of artifacts, in which WASMF is an emitted file with a file theorem, CALIGN a C file whose memory

Table 1: Targets by the codes of leanexe's table of artifacts, the semantics of their theorems, the form of each theorem, and what each theorem trusts beyond Lean's kernel and Lean's core definitions (README.md:86–98, docs/trusted-base.md:17–210).

Code	Semantics	Form of the theorem	Further trusted items
WASM, WASMF	Talos and the binary decoder	The call returns `f x` or traps at `unreachable`, and `trapFree` excludes the trap under its premises	Talos's semantics; the decoder; the engine; the host's placement of arguments
WASI	Talos with a model of the Preview 1 host	For every host that meets the host specification, the exports perform the action's ordered effects	The items of WASM; the Preview 1 model; Wasmtime's agreement with it
C, CALIGN	The Lean copy of CompCert 3.18's Clight	The C function terminates with the words of `f x` or with a trap code	Talos's store and IEEE 754 definitions; the Lean copy; one front end in `clightgen` and `ccomp`; CompCert's proofs and unverified parts; host, libc, and operating system
METAL	The Lean copy, transferred by Rocq to CompCert's Asm	The whole program terminates with one UART store per result byte, or runs a step forever with each window's trace	The items of C except host, libc, and operating system; the Lean copy outside N's eleven programs; CompCert's Asm; startup code; the stack-bound tool; the machine
A64	Model of the AArch64 boot machine	From boot, the machine writes the encoding of `f x` and status 0, or a status alone, and turns off	The model's fidelity; the floating-point equations; the boot model's system registers; the file reader; QEMU's loader and UART
A64L	Model of a process at EL0 with a rule for `SVC #0`	For every reply sequence of the kernel, the process makes the stated calls and output and exits	A64's model at EL0; Linux's agreement with the call layer; Linux's loader and initial stack; the shim's specification
M7	Model of the Cortex-M7 microcontroller from reset	For every device input and number of windows, the model performs the initialization and each window's request	The model's fidelity; the M-profile reading; reset; run conditions; USART3 and its channel; timing
WGSL	Invocation semantics and part of the Vulkan memory model	Every consistent execution of the dispatch is race-free and leaves the Lean result	Invocation semantics; the model's statement; no invocation past a stop; WebGPU's ordering of queue operations; shader compilers; binary32 arithmetic
PTX	Thread semantics and part of the PTX memory model	Every consistent execution of the launch is race-free and copies back the Lean result	Thread semantics; the model's statement; no thread past a stop; readings of the ISA; `ptxas`

accesses omit the run-time test of address alignment, METAL a whole bare-metal program, and A64L an AArch64 Linux executable (README.md:86–98). Every target trusts Lean's kernel at version 4.34.1 and the axioms `propext`, `Classical.choice`, and `Quot.sound`. Every target also trusts Lean's core definitions, which give a source program its meaning (docs/trusted-base.md:7–14).

3 WebAssembly modules and WASI commands

For the WebAssembly target, the module's theorem is the final theorem [13]. Leanexe's documents state three differences between Talos and an engine: Talos bounds no call depth, it grows memory whenever the new page count lies within Talos's memory cap, and it returns the positive canonical NaN for every NaN result (docs/trusted-base.md:25). The binary decoder gives the module's

bytes their meaning. Where a file theorem exists, it states that the emitted file holds the module's encoding (docs/trusted-base.md:27, docs/trusted-base.md:39).

WASI Preview 1 commands come from a separate compiler for Lean actions that read and write the standard streams, call other functions of Preview 1, and exit (README.md:22, README.md:72). Their theorems hold in Talos with a Lean model of the Preview 1 host, which follows the WASI interface definitions at commit `fae981ba` [1] (docs/verified-wasi-design.md:7). The theorems quantify over every host that satisfies leanexe's host specification (docs/trusted-base.md:54).

Under a host that satisfies leanexe's host specification, every run with enough fuel that returns normally gives the source's value. A run with enough fuel that ends at an accepted exit or a host trap gives the source's terminal outcome and ordered trace of effects (docs/wasi.md:202–208). A theorem about every satisfying host says nothing about a host that leaves a request unanswered. Leanexe proves separately that a satisfying host exists (docs/trusted-base.md:50).

4 C functions in the Lean copy of CompCert's Clight

CompCert is a C compiler programmed and proved in Rocq, the prover formerly named Coq [17, 18]. Clight is CompCert's subset of C whose expressions have no side effects, and Asm is CompCert's semantics of each target's assembly language [18, §1.3]. Leanexe's translator writes one C function for each WebAssembly function of a module that passes a decidable check of the translator's subset of WebAssembly (README.md:183).

The translator's simulation theorem, `translate_correct`, relates every Talos run that ends in a return or a trap to a terminating execution in a Lean copy of CompCert 3.18's Clight, by induction on Talos's fuel (LeanExe/Native/Proof/Correct.lean:216). The theorem `ImplementsA.native` composes `translate_correct` with the module's theorem (LeanExe/Native/Proof/Native.lean:115). The result, `p.f.native`, takes a C state related to a Talos store that meets the module theorem's premises. It states that from that state the C function terminates with words that represent `f x`, or with a trap code when the function may abort (LeanExe/Native/Proof/Native.lean:89). The Lean copy takes the pointer width and the NaN choices of the target as parameters, so the theorem holds under the parameters of AArch64, x86-64, and 32-bit ARM (docs/trusted-base.md:71).

The C theorems trust the Lean copy's agreement with CompCert's Clight, which review and tests support. A side-by-side review compares each definition of the Lean copy with the CompCert 3.18 definition that it cites (docs/native-translator.md:237). A differential test compares the Lean copy's evaluator with CompCert's reference interpreter, `ccomp -interp`, on random programs under three architectures (docs/trusted-base.md:71). The test would not detect a rule of the Lean copy that gave an outcome to a state that CompCert's Clight leaves stuck, and only review checks for such a rule (docs/trusted-base.md:74). For eleven whole programs compiled for AArch64, a proof replaces this trust, as Section 5 describes.

Each build prints the translation as a C file, and `clightgen`, CompCert's tool that converts a C file to Clight, reads the file back. The build's `clightgen` and its compiler, `ccomp`, come from the same CompCert tree. An unverified Lean reader compares the result of `clightgen` with the translation (docs/trusted-base.md:78). The build runs `ccomp` on the file only after the comparison passes. The comparison leaves two items trusted: the premise that the two tools run the same front end on the same text, and the Lean reader (docs/trusted-base.md:78).

Compiled Lean code evaluates the translation that the build compares, and the kernel checks each translated function against `translate` only for the eleven programs of Section 5 (docs/trusted-base.md:76, LeanExe/Native/ClightN/Instance.lean:509). Leanexe's AArch64 `ccomp`

comes from a CompCert 3.18 tree with a changed function prologue, because the released prologue misaligns the stack pointer SP for some frames [3]. Leanexe's repository holds no copy of the change (`docs/trusted-base.md:80`). A hosted executable links the translation with a C host, libc, and the operating system, and no theorem covers the executable as a whole (`docs/trusted-base.md:66`).

5 Whole bare-metal programs and the Rocq step to CompCert's Asm

A bare-metal program replaces the C host, libc, and the operating system with startup code and a `main` function that the translator writes and proves (`docs/native-bare-metal.md:5`). In the simplest form, `main` calls one entry on fixed arguments and writes the result's bytes to the UART (`docs/native-bare-metal.md:7`). The theorem `p.f.program`, which follows from the general theorem `bare_program`, states CompCert's whole-program judgment in the Lean copy (`LeanExe/Native/Proof/Main.lean:663`). By that judgment, the program terminates with result 0 and one volatile store per byte of the result's encoding, or, when the entry may abort, with result 1 and no output (`docs/native-bare-metal.md:104–112`).

A generated `main` can also read a message that a loader places in memory, or run a step function forever (`docs/native-bare-metal.md:7`). For a `main` that runs forever, the theorems `ctl_program` and `req_program` establish the judgment `LoopReacts` for every sequence of values that the volatile loads return (`LeanExe/Native/Proof/Reacts.lean:34`). `LoopReacts` states that `main` runs its first statements and then a loop whose iteration n has the trace of window n, where a window is one call of the step function (`README.md:220`).

5.1 The Rocq composition from Clight to Asm

CompCert's main theorem, `transf_c_program_correct`, is a backward simulation from the semantics of CompCert C to Asm (`driver/Compiler.v` at v3.18). Applying it requires a fact about every behavior of the CompCert C program, and a big-step judgment about Clight gives no such fact (`docs/native-bare-metal.md:149`). Leanexe's Rocq file `rocq/Clight2Asm.v` instead composes CompCert's forward simulations from Clight to Asm. Its theorem `clight2_bigstep_to_asm` concludes that the Asm program terminates with the trace and result of a Clight termination (`rocq/Clight2Asm.v:108`).

From the determinacy of Asm, the theorem `clight2_bigstep_asm_unique` adds that the termination of `clight2_bigstep_to_asm` is the Asm program's only behavior when the trace reads no input (`rocq/Clight2Asm.v:335`). The theorem `clight2_bigstep_asm_reads` covers programs that read inputs, and `clight2_loop_to_asm` transfers a loop that runs forever to a reactive behavior of the Asm program (`rocq/Clight2Asm.v:552`, `rocq/Clight2Asm.v:1050`). The proof of `clight2_loop_to_asm` builds the infinite Clight execution by a cofixpoint, as the Coq development of Vélus [12] does (`docs/native-bare-metal.md:300`). On ARMv7-M, the uniqueness theorems take the premise that every integer volatile load reads an integer, which a device that returns bits meets (`docs/trusted-base.md:103`). CompCert's Asm semantics has no stack size, so nothing follows about a run whose stack overflows (`docs/native-bare-metal.md:167`).

5.2 The step from the Lean copy to CompCert's Clight

The Rocq theorems of Section 5.1 take as their premise a judgment of CompCert's own Clight in Rocq, while the Lean theorems state a judgment of the Lean copy. The step between the two judgments is machine-checked for the eleven AArch64 programs of Table 2 through a reference

Table 2: The programs whose judgments the reference semantics N transfers from the Lean copy to CompCert's Clight on AArch64 (clightn/README.md:51–63).

Instances	Programs	Rocq theorem
gcd, crc32, sha256, euler, depth	Programs that terminate: GCD, CRC-32, SHA-256 of a constant, an Euler solver, a recursion of depth 1,000	clight2_bigstep_to_asm
sha	SHA-256 of a message that the loader places in memory	clight2_bigstep_asm_reads_ptr64
ctl, echo, deadline, records, gateway	Programs whose main runs forever: a filter, an echo, an echo with a deadline, a program that answers records, and an energy controller's gateway	clight2_loop_to_asm_ptr64

semantics N (clightn/README.md:3, clightn/check.sh:65). Rocq proves that every derivation of N's judgments is a derivation of CompCert 3.18's Clight semantics. Lean proves that, for a program in the forms that the translator writes, every execution in the Lean copy is a derivation of N (LeanExe/Native/ClightN/Exec.lean:2183, LeanExe/Native/ClightN/Exec.lean:2227). Figure 2 shows the chain of theorems from the module's theorem to CompCert's Asm.

Leanexe states N once in a prover-neutral language and renders that statement mechanically into Rocq and Lean. The script clightn/check.sh confirms, from what each prover elaborated, that both renderings say what the neutral statement says (clightn/README.md:3). The script also plants errors of each kind in copies of the renderings and the proofs, and it requires the comparison to report exactly those errors (clightn/README.md:103). The step through N trusts this check and the premise that clightgen's exporter printed the program that clightgen holds (docs/trusted-base.md:72–73). It also trusts equations between CompCert's floating-point operations and the Lean copy's, which no proof in either prover supports (clightn/README.md:77).

The statements that pass from Lean to Rocq differ among the three kinds of program in Table 2. For the terminating programs, that statement names the trace of output bytes and the result. The chain therefore reaches the Asm program's termination with those bytes (clightn/README.md:65, clightn/rocq/NInst.v:19). For the program with input and the programs that run forever, the Rocq statements require only that the trace's loads read the input values in order. For these programs, only the Lean copy's theorems state the output and the states (clightn/README.md:73, clightn/rocq/NInst.v:77).

Outside the eleven AArch64 builds of Table 2, the review of the Lean copy, the differential tests, and the per-build comparison support the step from the Lean copy's judgment to CompCert's Clight judgment (docs/trusted-base.md:105). The builds outside Table 2 include every ARMv7-M build of the eleven programs, because N's proofs need 64-bit pointers. They also include every x86-64 build, because that path has no CompCert tree with built proofs (clightn/README.md:113). At the pinned commit, leanexe's README states that the step is machine-checked only for the terminating GCD and SHA-256 programs and that no proof covers a program that runs forever (README.md:24, README.md:220). The sources that this section cites show the eleven programs of Table 2.

6 AArch64 Images and Linux executables

Leanexe's AArch64 back end compiles a module without imports to AArch64 instructions (LeanExe/A64/Gen.lean:947). It accepts the forms that its test programs and the WASI commands need, and it rejects every other form (docs/a64-bare-metal.md:29). The theorem program_sim

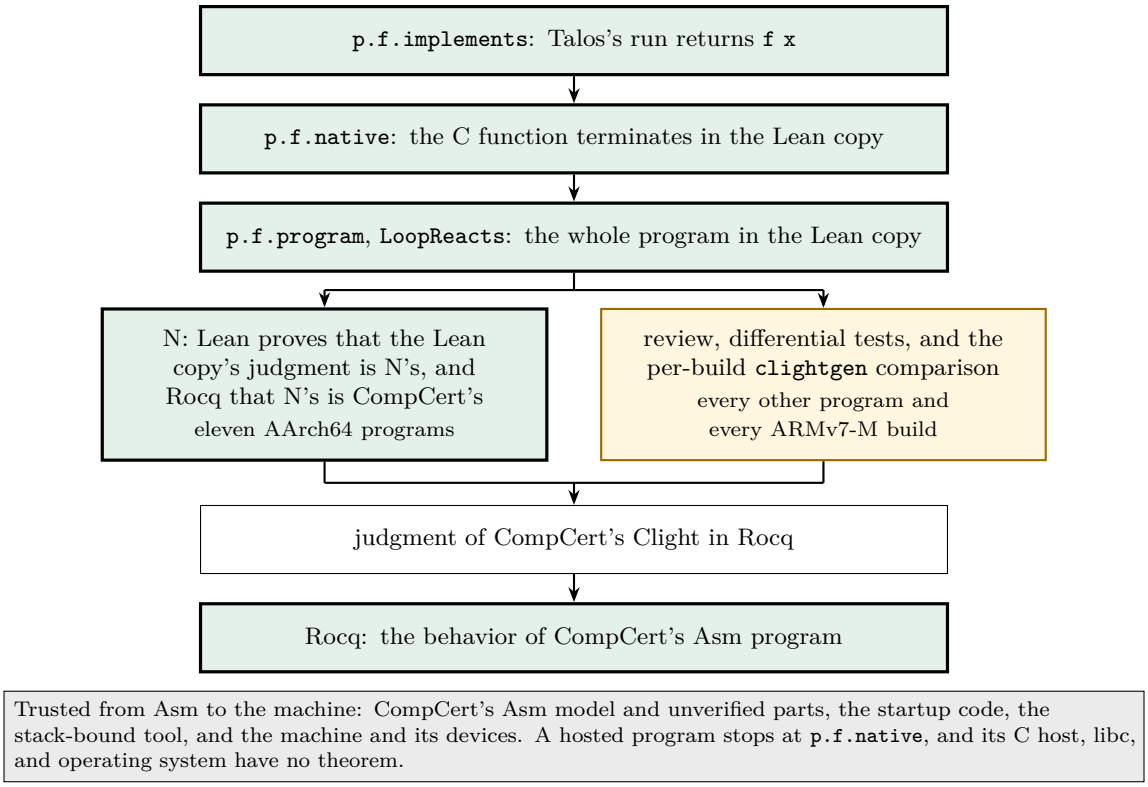

Figure 2: The chain from the module's theorem to CompCert's Asm for a bare-metal C program. Boxes with heavy borders are theorems that Lean's kernel or Rocq checks, the orange box is a step that review and tests support, and the white box is the judgment that both branches reach. The left branch holds for the programs of Table 2. The gray box lists what the chain takes without proof after Asm (`docs/trusted-base.md`:98–119). The stack-bound tool in it is a script that fails the build when the static stack bound of `main` exceeds the reserved stack (`docs/trusted-base.md`:110).

relates Talos's run to the run of the generated code in a Lean model of one AArch64 core at the kernel privilege level EL1, by strong induction on Talos's fuel (`docs/a64-bare-metal.md`:49). The model's state holds the general and floating-point registers, SP, the program counter, the flags, memory, and the UART's output (`docs/a64-bare-metal.md`:61). The model stops wherever the architecture permits several behaviors or depends on state that the model does not track, so a theorem about a run that reaches its outcome excludes those states (`docs/a64-bare-metal.md`:71).

The function `genImage` places the generated code after a startup routine in an arm64 Linux Image, the format of the arm64 Linux kernel's boot file (`LeanExe/A64/BootImage.lean`:34). The startup writes every system register that it relies on, invalidates the TLB, enables the MMU with RAM as Normal memory, sets the floating-point control register FPCR to `0x02000000`, and calls the program (`docs/a64-bare-metal.md`:87). A proof in a model of the boot machine composes the startup with the program's theorem. That proof starts from every boot state at EL1 with the MMU off, whatever the values of the other system registers and the contents of the TLBs (`docs/a64-bare-metal.md`:97).

The theorem `image_run_at` states the Image's run at every base address in the GiB of RAM that is a multiple of 4096 (`LeanExe/A64/Proof/ImageTheorem.lean`:319). The Image's final theorem allows the outcomes that the compiler's theorems allow. Where `p.f.trapFree` and the stack's bound cover the program, the only outcome is the result, and otherwise a trap's status is

also an outcome (docs/a64-bare-metal.md:481).

Leanexe's AArch64 back end also writes static Linux executables, whose theorems cover the run from the entry at the user privilege level EL0 to the call of `exit_group` (README.md:266). The theorems hold in a model of the process that adds to the machine model a rule for the system-call instruction `SVC #0`. That rule is a Linux call layer whose table a script generates from the system-call descriptions of the kernel fuzzer syzkaller, or, for one executable, a model of `read` and `write` (README.md:266). The theorems hold for every reply sequence of the kernel, except two theorems that assume a class of replies (README.md:266).

The back end also turns a WASI command of Section 3 into a static Linux executable, with stubs that implement a shim from WASI to Linux in place of the command's imports. The executable's theorem states its system calls as the images of the WASI events of a run of the command's action under the shim's host (docs/trusted-base.md:145). Talos and the WASI host model therefore occur in the theorem's proof and not in its statement.

7 Validation of the AArch64 model against Arm's specification

No proof about the generated code can establish that the processor executes each instruction as the model does. Leanexe gathers three kinds of evidence that the processor agrees with the model: co-simulation, tests of the floating-point forms, and Lean proofs against Isla's traces (docs/a64-bare-metal.md:13). In co-simulation, a bare-metal harness runs single instructions with random field values under QEMU's emulation (TCG) of a Cortex-A57 and under KVM on the host's Cortex-X925 and Cortex-A725 cores. A checker compares each final state with the model's (docs/a64-bare-metal.md:121).

Two tests cover the floating-point forms of the model. A vector test runs the floating-point arithmetic forms on operands at the corner cases of IEEE 754 under QEMU and on both host cores, and every result word equaled the model's (docs/a64-bare-metal.md:139). An exhaustive test runs the single-precision code of each unary operation and conversion on every 32-bit word (docs/a64-bare-metal.md:156). These tests cover sampled states on particular processors (docs/a64-bare-metal.md:150).

7.1 Agreement of each form with its Isla trace

Arm writes its architecture specification in ASL, its specification language [22], and a translator generates from it a model in Sail, a language for instruction-set semantics [9]. Isla executes a Sail model symbolically [10]. For one instruction with symbolic operands, Isla produces a trace of the instruction's register and memory accesses, with their values as SMT expressions and the conditions of each path.

For each instruction form that a program's run in the model uses, a Lean theorem states that the model's execution agrees with the form's trace from Arm's Armv9.4 specification (LeanExe/A64/Isla/Check.lean:64, docs/a64-bare-metal.md:164). Whenever the model's step succeeds from a state, every path of the trace from that state ends in the model's resulting state, and some path reaches it. The theorem holds for every value of the opcode's fields and of the registers that Isla's configuration leaves unknown, in a profile for QEMU's processor and one for the host's cores (docs/a64-bare-metal.md:166).

Isla's Arm model represents IMPLEMENTATION DEFINED choices as constants and CONSTRAINED UNPREDICTABLE choices as symbolic values. A form's dependence on either therefore appears in its trace, while a test on one processor observes only that processor's choices (docs/a64-bare-metal.md:162). The theorems of branches, loads, and stores have

premises on targets, addresses, and the translation table, which every run of a program meets (docs/a64-bare-metal.md:168–172).

The check replaces trust in the model's agreement with the processor by trust in five items. The items are Arm's ASL; its translation into Sail; a patched Isla with the SMT solver Z3; three edits of the specification's intermediate representation; and the Lean reader and semantics of traces, which follow Islaris [23] (docs/trusted-base.md:133). The startup's system instructions have no Isla theorem, and the boot model gives them meanings from a specification that cites the Arm Architecture Reference Manual [8] rule by rule (docs/trusted-base.md:134).

7.2 Floating-point equations as hypotheses

Arm's pseudocode computes each floating-point result by rounding a real number, which Isla does not implement, so an edit leaves each floating-point function abstract in the traces (docs/a64-bare-metal.md:176). The theorem of each floating-point form takes as a hypothesis an equation between Arm's function under FPCR `0x02000000` and Talos's IEEE 754 function, such as `FPAdd(a, b, FPCR) = Wasm.f64Add a b` (docs/a64-bare-metal.md:178). The floating-point tests and proofs for every operand, over Sail 0.20.3's Lean translation of the functions, support the equations (docs/a64-bare-metal.md:200).

A script checks the proofs of the equations over Sail's Lean translation on a host with Sail, outside the repository's build (docs/trusted-base.md:131). A change of Talos, Mathlib, or the toolchain therefore leaves those proofs unchecked until the script runs again. The proofs trust Sail's Lean back end, which Sail's authors call experimental, and linking statements that Isla's calls return what the translation returns (tools/sail/README.md:658, tools/sail/README.md:670). The check as a whole compares individual forms in fixed configurations with traces of the specification, and it proves no equality of the model with Arm's specification (docs/a64-bare-metal.md:503).

8 Cortex-M7 flash images and step functions that run forever

A program with requests runs a step function `tick : G → UInt64 → G × UInt64` forever. A window is one call of the step: it takes the state and the value that the last request read, and it returns the next state and a request for one device access (docs/m7-bare-metal.md:30). Leanexe's Cortex-M7 back end compiles the step's module to Thumb-2 code for the STM32F767ZI microcontroller on ST's NUCLEO-F767ZI board (docs/m7-bare-metal.md:3). The back end rejects floating point and memories of more than one page (docs/m7-bare-metal.md:571).

The theorem `step_call` relates the generated step's run in a Lean model of the microcontroller to Talos's run (LeanExe/M7/Proof/Step.lean:198). The theorem `board_run` takes the step's module theorem as a premise and states the run from reset (LeanExe/M7/Proof/Board.lean:249). For every sequence of values that device reads return and every number of windows, the model performs the initialization's accesses and then the accesses that each window requests (README.md:292). Further theorems relate the bytes sent to a specification: `sha_board` states that the bytes sent begin the SHA-256 digests, as FIPS 180-4 defines them, of the prefixes of the blocks read (README.md:292).

8.1 The Isla check against the A-profile and the reading of the M-profile

Leanexe's documentation states that no machine-readable specification of the M-profile, Arm's architecture profile for microcontrollers, is public (docs/m7-bare-metal.md:431). The check therefore compares each form with Isla's trace of the same T32 encoding in Arm's A-profile specification, in AArch32 Thumb state (docs/m7-bare-metal.md:431). Each theorem establishes

the form's behavior as the A-profile defines it, and nothing about the M-profile (docs/m7-bare-metal.md:431). A reading of each form's M-profile pseudocode in the ARMv7-M manual, DDI 0403E.e [7], against its A-profile pseudocode, with an independent review, belongs to the trusted base (docs/trusted-base.md:170).

Reid, then at Arm, wrote that in the M-class architecture many encodings that access the stack pointer or the program counter are UNPREDICTABLE, while the same encodings are well defined in the A-class architecture [22, §II-A]. He concluded that "it would be unsound to use the A-class specification to reason about Thumb machine code intended for an M-class processor" [22, §II-A]. Leanexe's model excludes those encodings, because each register field of the model names only r0 to r7, and each form with SP or LR in a register field is a constructor of its own (docs/m7-bare-metal.md:46). Leanexe's Cortex-M7 document lists the remaining differences, among them the masking of SP's low bits and division by zero, and the trusted base includes the reading of those differences (docs/m7-bare-metal.md:459–472). Co-simulation on QEMU and on the board tests the model, and a model with a planted error in `ADC`'s carry disagreed with the board on exactly the `ADC` cases with the carry set (docs/m7-bare-metal.md:485).

8.2 Run conditions and time

The theorems hold under run conditions: a power-on reset, or an NRST reset with the debug registers at their power-on values, and no interrupt, debugger access, or DMA during the run (README.md:294). Theorems bound the number of instructions in every window of each image except the SHA-256 step's, and no theorem bounds time (README.md:296, docs/m7-bare-metal.md:12). The serial port USART3 holds one received byte, so each window must end before the next byte arrives, and cycle measurements on the board support that margin (docs/trusted-base.md:176).

9 GPU kernels from the kernel source language

From the typed source program, a lowering step of the compiler constructs a kernel in a kernel source language that belongs to neither GPU target, with a proof of the kernel's meaning (docs/design.md:236–241). The WGSL and PTX compilers translate that kernel with theorems of their own. Each printer has a parser with a proof that the parser inverts the printer, and the emitters check the printed text with that parser in compiled Lean code (docs/trusted-base.md:181). The GPU's compiler reads the text with its own parser, which no theorem relates to leanexe's parser.

A WGSL dispatch theorem states, in a sequential semantics that runs the invocations one after another, that the dispatch writes the Lean function's result and that distinct invocations store to distinct words (LeanExe/WGSL/Semantics.lean:200, docs/trusted-base.md:181). An axiomatic memory model defines a program's executions as sets of events with relations between them, and an execution is consistent when the relations meet the model's axioms [21, 26]. The theorem `Dispatch.executions` uses a Lean statement of the parts of the Vulkan memory model that the kernels use. It derives that every consistent execution of the dispatch has every invocation end, no data race, and the same output words (LeanExe/WGSL/MemoryModelProof.lean:276). The Lean statement omits reads-from and from-reads from the model's acyclicity axiom, so it admits more executions than the model (docs/trusted-base.md:187).

For PTX, the theorem `Launch.executions` states that every consistent execution of a race-free launch whose sequential run ends has no data race and copies back the output of the sequential run (LeanExe/PTX/Launch.lean:318). Leanexe's statement of the PTX ISA's memory model [21] omits No Thin Air, Fence-SC, atomicity, and sequential consistency per location (docs/trusted-base.

md:203). That statement therefore admits more executions than the model, as the Lean statement of the Vulkan model does.

Both GPU targets trust strict binary32 arithmetic on the device, while a GPU may flush subnormal values and fuse multiply-adds (`docs/trusted-base.md`:190). Both targets also assume that no invocation or thread on the device runs past a point where its semantics stops, such as an access outside a buffer (`docs/trusted-base.md`:186, `docs/trusted-base.md`:205). For WGSL, leanexe states that neither WGSL nor the Vulkan memory model gives a rule from which that property follows (`docs/trusted-base.md`:194). For PTX, leanexe argues that a thread can run past a stop only through a cycle of dependencies on values that such departures produce, which the No Thin Air axiom forbids (`docs/trusted-base.md`:210).

Koska et al. describe a schedule that leanexe's PTX argument has to address. In that schedule, a compiler moves a thread's constant store before its load, two threads exchange the stored value, and one thread reaches undefined behavior [16, §5.3]. Koska et al. call this schedule "not precluded by the No-Thin-Air axiom" and conclude that a sound use of the axiom requires a proof that a semantic dependency exists [16, §5.3, §5.4]. This report therefore states, as an assumption of both GPU targets, that no invocation or thread runs past a stop.

10 Related work

Table 3 compares leanexe's targets with verified compilers and verifications of machine code. CompCert's Asm semantics, written by hand for each of its architectures, is part of its trusted base [18, §1.2, §1.3]. Leanexe's C target adds to CompCert's trusted base the Lean copy's agreement with CompCert's Clight, an item that the step through N removes for eleven AArch64 programs.

Kanabar, Fox, and Myreen related the L3 model of Armv8.0, against which CakeML's Armv8 back end [25] is verified, to Arm's specification. They proved in HOL4 that the L3 model simulates the Sail model of Arm's Armv8.6-A ASL for every encodable instruction that the back end emits [15, §4.4, Theorem 3]. Their result excludes floating point, SIMD, AArch32, and exceptions, and it replaces address translation with an identity mapping [15, §4.1, §4.2, §6]. Leanexe's check also covers floating-point forms, with equations for Arm's floating-point functions as hypotheses, and loads and stores through the startup's translation table. Kanabar et al.'s proof runs in HOL4 over the HOL4 translation of the Sail model, while leanexe's check trusts Isla and its SMT solver, as Islaris does [23, §1].

Islaris's authors validated Isla's output against the Sail-generated Coq semantics for RISC-V. They report that the size of the Armv8-A model made that validation infeasible for Arm [23, §5]. Armstrong et al. validated their 2019 Sail translation of Arm's Armv8.3 ASL by tests, and Arm's validation suite found tests that passed on Arm's model and failed on the Sail model [9, §7].

seL4's binary verification validates each build of gcc's output over Fox and Myreen's ARMv7 model, which its authors tested on hardware boards [24, §1, §2.2], [14, §5]. Leanexe's machine-code back ends instead generate the code with a proof. Vélus's theorem states that the assembly's infinite trace is bisimilar to the volatile loads and stores of the program's streams [12, §1], the shape of leanexe's theorems for a `main` that runs forever.

WasmCert mechanizes WebAssembly 1.0 in Isabelle/HOL and Coq and proves type soundness [28, §1, §3]. Leanexe's WebAssembly target trusts Talos's fidelity to the standard. WasmCert addresses that item with a transcription close to the specification's prose and with the WasmCert-Isabelle interpreter's pass of the official test suite [28, §1, §4]. vWasm proves that its x86-64 output stays within its sandbox, a property that its authors call orthogonal to compiler correctness [11, §1, §4.3].

Lustig et al. formalized the PTX 6.0 memory model with No Thin Air among its axioms [19,

Table 3: Scope of the theorem, target semantics, and trusted base of leanexe and of related systems. The prover is in parentheses, and each trusted base is as the system's primary source states it.

System	Scope of the theorem	Target semantics and its support	Trusted base
leanexe (Lean; Rocq from Clight to Asm)	For each program, the compiled code computes the Lean function in each target's semantics	Talos; a Lean copy of Clight; Lean machine models checked against Isla's traces; parts of GPU memory models	The items of Table 1
CompCert 3.18 [18] (Rocq)	The code's behavior improves on an allowed source behavior	Asm, written by hand for each architecture	Preprocessing, elaboration, Asm, printing, assembling, linking
CakeML backend [25] (HOL4)	The machine code's behaviors are the source's, with out-of-memory added	ISA model of each target architecture	ISA models, FFI as its oracle states, wrapper assembly, linker
Kanabar et al. [15] (HOL4)	L3's Armv8.0 model simulates the ASL-derived Armv8.6 model for CakeML's instructions	Arm's ASL through Sail	The ASL and its translation; no floating point or exceptions
Vélus [12] (Coq)	The assembly's infinite trace is bisimilar to the stream semantics	CompCert's Asm	CompCert's trusted parts
seL4 binary verification [24] (Isabelle, HOL4)	The gcc binary refines the C semantics, for each build	ARMv7 model tested on boards	The validation tool, the binary's import, SMT solvers
Islaris [23] (Coq)	Machine code meets separation-logic specifications	Isla's traces of Sail models	Isla, the SMT solver
WasmCert [28] (Isabelle, Coq)	Type soundness; interpreter and type checker against the semantics	WebAssembly 1.0, close to the prose	Fidelity to the prose
vWasm [11] (F*)	Sandbox safety of x86-64 output, with no functional correctness	Model of an x86-64 subset	F*, Z3, OCaml, the model, printer, assembler
Lustig et al. [19] (Alloy, Coq)	The mapping of race-free scoped C++ to PTX is sound	Model of PTX 6.0 from NVIDIA's prose	The reading of the prose
Koska et al. [16] (Lean)	Hand-written kernels meet their postconditions in every consistent complete execution	PTX subset without No Thin Air and Fence-SC	`ptxas`, hardware, freedom from undefined behavior

§3.5]. Koska et al. verify hand-written PTX kernels in Lean under a model without No Thin Air and Fence-SC [16, §3.3, §4]. Leanexe's PTX kernels come from a compiler with a proof for every compiled kernel.

11 Proofs, tests, and assumptions

Each target assumes the items of its row of Table 1. Table 4 gives, for each target, what is proved and what is tested. Every target also assumes Lean's kernel, which no independent checker has rechecked. Every processor target assumes that the processor implements the architecture as its model, or CompCert's Asm, describes it. The author read the statements and documents at the pinned commit and did not rebuild the proofs.

12 Limitations

No theorem bounds running time. Cycle measurements support the Cortex-M7 timing margins (`README.md:296`). This report quotes no compile time or memory, since leanexe has no committed

Table 4: Proved and tested items by target, from `docs/trusted-base.md`. Lean's kernel checks the proofs unless the table names Rocq.

Target	Proved items	Tested items
WebAssembly, WASI	The module's theorems [13]; file theorems; WASI theorems for every satisfying host, and such a host's existence	Talos against the official test suite; modules against native Lean under Wasmtime
C	`translate_correct` and `p.f.native`; N's leaf forms on AArch64, in Rocq and Lean	The Lean copy against `ccomp -interp` under three architectures; leanexe's test modules natively against Wasmtime
Bare metal	`bare_program` and the `LoopReacts` theorems; in Rocq, Clight to Asm for AArch64 and ARMv7-M; in Lean and Rocq, the step through N for eleven AArch64 programs	Runs on QEMU under TCG and KVM, on the board, and in Renode
AArch64 Images	`program_sim`, the startup, `image_run_at`, the file checks, and each form's agreement with its Isla trace; the floating-point equations over Sail's translation, outside the build	Co-simulation under TCG and KVM on two core types; vector and exhaustive floating-point tests
Linux executables	The process theorems and the shim's stubs	The call layer's frame test; runs natively and under `qemu-aarch64`
Cortex-M7	`step_call`, `board_run`, window instruction bounds, each form's agreement with its A-profile trace	Co-simulation on QEMU and the board; board runs; cycle measurements
GPU	Translations, inverse parsers, dispatch and launch theorems under the statements	Runs on SwiftShader, llvmpipe, browsers, and an NVIDIA GB10

record of them.

No machine model is proved equal to Arm's specification (Section 7). The Cortex-M7 model is checked against the A-profile. Its agreement with the M-profile depends on a reading of Arm's M-profile manual, which no check confirms. The floating-point equations of the AArch64 check are proved outside leanexe's build.

A hosted C executable has no whole-program theorem, and the step from the Lean copy to CompCert's Clight is proved for eleven AArch64 programs only. At the pinned commit, two of leanexe's documents understate the scope of the step through N and of the kernel's check of translations (`README.md:24`, `docs/trusted-base.md:76`). Talos is not proved faithful to the WebAssembly specification. The GPU statements omit axioms of the published memory models, and both GPU targets assume that no invocation or thread runs past a stop.

References

[1] WASI preview 1 interface definitions. `WebAssembly/WASI`, commit fae981ba of 17 December, 2025. URL https://github.com/WebAssembly/WASI/blob/fae981bae14809d91f9bc2d63852d461f331d161/preview1/witx/wasi_snapshot_preview1.witx.

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

[3] CompCert issue 614. GitHub issue on the AArch64 function prologue, 2026. URL https://github.com/AbsInt/CompCert/issues/614.

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

[5] Talos. Git repository `cajal-technologies/talos`, 2026. URL https://github.com/cajal-technologies/talos.

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

[7] Arm Limited. *ARMv7-M Architecture Reference Manual*, 2021. URL https://developer.arm.com/documentation/ddi0403/latest. DDI 0403E.e.

[8] Arm Limited. *Arm Architecture Reference Manual for A-profile architecture*, 2026. URL https://developer.arm.com/documentation/ddi0487/latest. DDI 0487.

[9] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. *Proceedings of the ACM on Programming Languages*, 3(POPL):Article 71, 2019. https://doi.org/10.1145/3290384.

[10] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. In *Computer Aided Verification (CAV 2021)*, volume 12759 of *Lecture Notes in Computer Science*, pages 303–316. Springer, 2021. https://doi.org/10.1007/978-3-030-81685-8_14.

[11] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. Provably-safe multilingual software sandboxing using WebAssembly. In *Proceedings of the 31st USENIX Security Symposium*, pages 1975–1992, 2022. URL https://www.usenix.org/conference/usenixsecurity22/presentation/bosamiya.

[12] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A formally verified compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 586–601, 2017. https://doi.org/10.1145/3062341.3062358.

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

[14] Anthony Fox and Magnus O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Interactive Theorem Proving (ITP 2010)*, volume 6172 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2010. https://doi.org/10.1007/978-3-642-14052-5_18.

[15] Hrutvik Kanabar, Anthony C. J. Fox, and Magnus O. Myreen. Taming an authoritative Armv8 ISA specification: L3 validation and CakeML compiler verification. In *13th International Conference on Interactive Theorem Proving (ITP 2022)*, volume 237 of *LIPIcs*, pages 20:1–20:22, 2022. https://doi.org/10.4230/LIPIcs.ITP.2022.20.

[16] Luk Koska, Tom Koska, and Ben Koska. Formalizing PTX in Lean to prove GPU kernel correctness. Preprint arXiv:2611.04447v1, 3 November; no peer review stated, 2026.

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

[18] 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.

[19] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. A formal analysis of the NVIDIA PTX memory consistency model. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019. https://doi.org/10.1145/3297858.3304043.

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

[21] NVIDIA. Parallel thread execution ISA. Chapter 8, Memory Consistency Model, 2026. URL https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[22] Alastair Reid. Trustworthy specifications of ARM v8-A and v8-M system level architecture. In *Formal Methods in Computer-Aided Design (FMCAD 2016)*, pages 161–168, 2016. https://doi.org/10.1109/FMCAD.2016.7886675.

[23] Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. Islaris: Verification of machine code against authoritative ISA semantics. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 825–840, 2022. https://doi.org/10.1145/3519939.3523434.

[24] 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.

[25] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29:e2, 2019. https://doi.org/10.1017/S0956796818000229.

[26] The Khronos Group. Memory model. Appendix of the Vulkan specification, 2026. URL https://docs.vulkan.org/spec/latest/appendices/memorymodel.html.

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

[28] Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. Two mechanisations of WebAssembly 1.0. In *Formal Methods (FM 2021)*, volume 13047 of *Lecture Notes in Computer Science*, pages 61–79. Springer, 2021. https://doi.org/10.1007/978-3-030-90870-6_4.