ULPBENCH: AUTONOMOUS FORMAL VERIFICATION OF NUMERIC C

Anonymous authors
Paper under double-blind review

ABSTRACT

Machine-checked verification of critical C code, from cryptographic routines in OpenSSL to the seL4 microkernel, has historically demanded months to years of expert effort per component. Large language models (LLMs) are now capable of reasoning about C code and producing proofs in proof assistants such as Lean and Rocq, however these capabilities have largely been demonstrated for isolated pieces of the verification pipeline, such as synthesizing assertions or proving verification conditions, rather than for true end-to-end verification. In this paper, we explore the degree to which autonomous agents can perform end-to-end formal verification for numeric C code. To facilitate this, we introduce ULPBench, our custom verification benchmark consisting of 560 tasks drawn from production numeric C code. For each task, an agent is given a formal C semantics inside a proof assistant and must (i) write an executable functional model, (ii) prove that the C source refines it under that semantics, and (iii) prove a property of the model, such as a bound on floating-point error in terms of unit in last place (ulp). Completing all three stages of a task results in a full end-to-end proof, and each task is provided in both Lean and Rocq. To support the Lean side, we release a port of the CompCert Clight semantics and IEEE-754 binary32/binary64 floating-point semantics to Lean, providing the foundation needed to verify each real-world C program. We evaluate frontier agents, measuring success in end-to-end formal verification, and show that agents are able to autonomously formally verify challenging properties of C code in 34% of our test cases. Finally, we present a case study showing the use of this approach in practice which led to uncovering an unreported defect in CPython's atanh function as well as a defect in a major geospatial algorithm library.

1 INTRODUCTION

The C programming language continues to be one of the most widely used programming languages, as well as supporting many safety critical systems where trustworthiness is imperative (TIOBE Software BV, 2026; Holzmann, 2014). Providing formal verification that C code does not exhibit undefined behavior and satisfies certain properties is a natural and long-sought next step.

Complete end-to-end verification (inside of a proof assistant) is the strongest guarantee and refers to the ability to formally prove properties about a piece of code. Generally, this is a very expensive and often unscalable process, such as with verifying malloc/free, which took 2 person-months to verify 108 lines of C Appel & Naumann (2020), and seL4, an operating-system microkernel, which required about 20 person-years to verify 8,700 lines of C (Klein et al., 2010). Fortunately for numeric C, it is easier to specify and there is already a known architecture for end-to-end verification based on a deep embedding of the source code, identifying a functional model, and proving that (a) the source code is a refinement of the functional model, and that (b) the functional model is within some margin of error of the analytic function it approximates. Even with this architecture laid out, it still took many person-months to bound the accuracy of a simple C program based on Newton's method that approximated the square-root function Appel & Bertot (2020).

Large language models (LLMs) are capable of reasoning about C and, increasingly, can generate complex proofs in proofs assistants such as Lean and Rocq. Due to this, many research efforts have been emerging to construct components of the full end-to-end verification flow, such as bench-

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

marks for verified code generation, agents for verifying C, floating-point verification, and translating programs into proof assistants which we will discuss in depth in Section 2. However, no existing system or benchmark asks an agent to perform end-to-end verification over production numeric C code without a human in the loop. ULPBench addresses this gap in both Lean and Rocq, working towards the goal of fully verifying or identifying bugs across widely used numeric libraries in scientific computing, engineering and mathematics. We note that while our initial focus is on numeric C code, this method is not limited to it, with almost any C code being in scope. Our contributions are:

- **ULPBench.** A benchmark of 560 tasks built from numerically sensitive functions in production C code, each with a concrete specification and each provided in both Lean and Rocq. Each task comes in four variants, ranging from completion of individual stages to full end-to-end verification. The functions are drawn from open-source repositories spanning popular libraries implementing special mathematical functions, signal-processing codecs, geospatial algorithms, and more.
- **Lean tooling for C.** To support the Lean side, we release a port of the CompCert Clight semantics and IEEE-754 binary32/binary64 floating point, together with `forward` and `interval`-style tactics.
- **Evaluation.** We establish what current frontier agents can do on these tasks with hours-long budgets.
- **Case study.** We deploy the method on actively used open-source math libraries and uncover large numerical errors in CPython's math library, a popular library of geodesic algorithms, and more.

2 Related Work

Benchmarks for verified code generation. Recent benchmarks increasingly evaluate language models on machine-checked programming tasks. DafnyBench asks models to reconstruct proof annotations for existing Dafny programs (Loughridge et al., 2025). Other recent benchmarks target verified code generation more directly: VERINA and CLEVER decompose the generation of programs, specifications, and correctness proofs (Ye et al., 2026; Thakur et al., 2025), while VeriCoding scales synthesis from formal specifications across Lean, Dafny, and Verus (Bursuc et al., 2025). NTP4VC moves closer to production software by extracting verification conditions from real-world C programs and translating them into Lean, Rocq, and Isabelle via Frama-C and Why3 (Xu et al., 2026). In each of these settings, however, the object presented to the prover is already a program in the verification language or an extracted verification condition. ULPBench instead exposes the *refinement gap* between the pre-existing C source and a functional model: the agent must reason about the C program under a mechanized C semantics before proving the higher-level numerical property.

Agents for verifying C. Recent work on liblzma is closest to our setting in scale and method: agents construct VST proof scripts for pre-existing production C, establishing memory safety and partial functional correctness for components of a decoder (Shlyakhtun et al., 2026). Unlike the manually executed pipeline of Appel & Bertot (2020) (§3), liblzma reaches production scale with 338 lines of C yielding 183,000 lines of proof, but retains humans in the loop throughout the verification. Humans write and review models and specifications and approve every semantic change, while agents discharge proof obligations, and no numerical property is in scope. ULPBench instead turns this style of source-level verification into a benchmark across many independent production functions and explicitly evaluates model construction, C-to-model refinement, property proofs, and explores whether the human-in-the-loop can be removed entirely rather than replaced by agents.

Floating-point verification and libm accuracy. Floating-point code, and mathematical libraries in particular, have a long history of specialized verification and accuracy analysis. A substantial toolchain exists for proving tight rounding-error bounds automatically or semi-automatically, from Harrison's machine-checked trigonometric proofs (Harrison, 2000) to interval and rounding-error reasoning in Gappa (de Dinechin et al., 2011), Taylor expansions and global optimization in FPTaylor (Solovyev et al., 2018), optimization-based math.h error proofs (Lee et al., 2018), and certified approximation tools like Sollya (Chevillard et al., 2010). Complementary work has exposed and

measured errors in deployed implementations: Kong et al. identified floating-point bugs in an embedded GNU C library (Kong et al., 2013), and large-scale studies have measured worst-known ulp errors across widely used libm implementations (Innocente & Zimmermann, 2021). Finally, projects such as RLIBM and CORE-MATH construct implementations designed to achieve correct rounding (Lim & Nagarakatte, 2021; Sibidanov et al., 2022). ULPBench is complementary to these efforts: rather than introducing a new numerical analyzer or libm implementation, it evaluates whether general-purpose agents can construct machine-checked proofs about existing production C, separating the semantic refinement problem from the subsequent numerical argument.

Further related work on mechanizing programming-language semantics in proof assistants is discussed in Appendix A.

3 Background and Task

3.1 Background

Proof-assistant code verification gives a machine-checked guarantee over all inputs. Rocq has the infrastructure to reason about and formally verify a large subset of C programs due to decades of foundational work Leroy (2009); Appel (2011); Blazy & Leroy (2009), which has been demonstrated explicitly in numerous projects Schwabe et al. (2021); Appel & Bertot (2020). While Lean has received less attention in this area and its infrastructure is comparatively less mature, much of the groundwork laid out for Rocq is reusable for Lean, and we take advantage of that in our work as discussed later on in this section and in Section 5.

To claim end-to-end verification, the source program must be elaborated into the proof assistant via a deep embedding, meaning it is semantically faithful to the source language (Boulton et al., 1992). This method of embedding a target language in a higher-order logic is commonly used to verify real-world programs (Lee et al., 2007), and several such embeddings (Blazy & Leroy, 2009) (Memarian et al., 2016) (Krebbers, 2015) have been created for C in other theorem provers. However, to our knowledge, no such embedding previously existed for Lean. This is not due to any fundamental limitations of Lean. To enable rigorous verification of C in Lean, we present a C semantics embedding for Lean using as a reference the Clight semantics Blazy & Leroy (2009), defined in Rocq for the CompCert (Leroy, 2009) formally verified compiler and used by the Verified Software Toolchain (Appel, 2011). This has the advantage of enabling a 1-1 syntactic representation of elaborated programs directly as abstract syntax trees, so that the correspondence between the original source code and the program in Lean is clear. We will refer to the original source program and its elaboration via the deep embedding in Lean interchangeably throughout this paper.

3.2 Task

Proof assistant-based formal verification of C consists of three stages: (i) construct a functional model of the source program; (ii) prove that the source program refines this model (the refinement proof); (iii) prove that the model satisfies the target property (the properties proof). These are illustrated in Figure 1. In stage (i), the agent is asked to generate a functional model that is functionally equivalent to the source program over all inputs. The functional model will be written in the proof assistant language and can be thought of as an abstraction of the source code. Stage (ii) is proving that this is indeed true: the source code and the functional model are functionally equivalent; the source code refines the functional model. We note that functional models can more or less closely model the full complexity of the source program, meaning that depending on where in that spectrum the LLM-generated models fall, this may bias a performance tradeoff between tasks (ii) and (iii), as voiced by CLEVER's concern Thakur et al. (2025). However, we believe this will not impact end-to-end performance.

The property proof in stage (iii) is based on specifying error bounds. Floating-point error is conventionally measured in *units in the last place* (ulps): one ulp is the gap between adjacent floating-point numbers near the exact result (Muller et al., 2018). A correctly rounded result is off by at most 0.5 ulp, which IEEE-754 requires of basic arithmetic and square root (IEEE, 2019). For elementary functions such as `exp` and `sin`, the standard only recommends correct rounding, so math libraries instead aim to be "within k ulp" for a small k. For a typical task in ULPBench, the specification

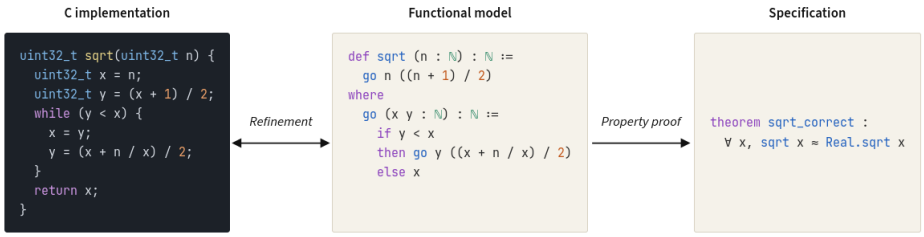


Figure 1: Given a C implementation on the left and a specification for its behavior on the right, the agent must identify a functional model and then generate a refinement proof and a property proof. By transitivity the C implementation satisfies the specification

is a bound with error given in this form. We give the formal definition and our conventions in Appendix D.

4 ULPBENCH

ULPBench combines the three stages from Section 3 into a benchmark. A *task* fixes a single C function, the Clight abstract syntax tree the CompCert front end produces for it, and a specification; an agent must produce some combination of the model, the refinement proof, and the property proof. Every task is issued in both Lean and Rocq against the same pinned source, so the two provers are compared on identical work. The benchmark comprises 560 tasks, each built from one of the 582 functions in our corpus of production numeric C, and each task comes in four variants that range from a single stage to full verification. Our primary focus is the full end-to-end verification and the intermediate variants of stages (i)-(iii) exist primarily to localize failures in particular components of the pipeline or witness different model behaviors. Note that this method generalizes beyond numeric code since any specification can be used; ulp bounds are simply the specifications that production math code already advertises.

4.1 SOURCE CORPUS

We draw functions from 35 open-source repositories under permissive licenses, pinned to specific commits and linked to their respective source lines. A function is eligible when three conditions hold. First, it is *numerically sensitive*: it branches on the bit pattern of its input, performs argument reduction or compensated summation, or has thin regions where error spikes—the places where a bug can hide from sampling but not from proof. Second, it carries a *concrete accuracy claim*, either stated in the source or its documentation (for example an "error $<$ 1 ulp" comment, a correctly-rounded guarantee, or a nanometre bound) or derivable from the algorithm. Third, it lies within the fragment of C that CompCert's front end accepts, which excludes `_Complex`, x87 extended precision, and compiler intrinsics that Clight does not model.

The corpus spans several domains so that success is not an artifact of one coding style: special-function libraries (openlibm, musl, glibc's libm, SLEEF, CORE-MATH, crlibm, and CPython's `math`), fixed- and floating-point signal-processing codecs (LC3, RLIBM, libvpx), geospatial and cartographic routines (GeographicLib, PROJ, WCSLIB), and further scientific code in oceanography (GSW) and astronomy (ERFA). Table 1 summarizes the breakdown by repository, function count, and floating-point precision.

4.2 EVALUATION AND GRADING

Evaluating frontier models on long-horizon verification tasks is expensive, so our main experiments use a 100-task subset of ULPBench, titled ULPBench-Small, rather than the full corpus. We construct this subset to approximate the broader distribution of the benchmark rather than selecting for apparent difficulty or expected solvability.

Category	Repos	Functions	Examples
`libm` variants	7	260	musl, openlibm
Audio/video codecs	7	117	libvpx, dav1d
DSP & transforms	6	77	liquid-dsp, pffft
Graphics & 3D math	4	53	raylib, cglm
Numeric utilities	11	75	ryu, pcg
Total	35	582	

Table 1: ULPBench source corpus, grouped by domain (35 permissive repositories, 582 functions).

Selection is stratified using several observable properties of the source tasks, including C function length, source repository, and structural complexity such as control flow, loops, and memory access. We additionally consider the form of the numerical specification and use human judgment to avoid over-representing families of near-duplicate routines. The resulting subset is intended to preserve the diversity of the full benchmark while making repeated long-horizon agent evaluation tractable.

The success of stage (i) is graded by differential testing of the submitted model against the compiled C implementation on special and boundary values and a large number of randomly sampled inputs.

The other stages are graded by machine checking in the target prover. Lean submissions are validated with `comparator`, which checks that the submitted proofs establish the frozen theorem statements and are accepted by the Lean kernel. Rocq submissions are checked by the Rocq kernel against the corresponding frozen statements Lean FRO.

5 Infrastructure

5.1 Lean: Clight and IEEE-754 semantics

Our C semantics definition for Lean is largely a translation of the Clight semantics defined for CompCert, with some notable modifications. We make minor simplifications to the CompCert memory model, since preservation across compiler passes is not needed for our use case. We validate the port by differential fuzzing against CompCert and state its trusted computing base in Appendix B.

To perform the elaboration from a C source program to an AST in Lean, we reuse the CompCert C frontend, which consists of a parser which is formally verified to accept the C grammar. We modify the CompCert parser backend to output ASTs in Lean syntax compatible with our C embedding instead of Rocq.

5.2 Rocq: CompCert, VST, and Flocq

On the Rocq side, we build on the existing CompCert and Verified Software Toolchain ecosystem: C refinement proofs use VST-Floyd over CompCert's Clight semantics (Cao et al., 2018), while floating-point reasoning relies on Flocq and related verified libraries (Boldo & Melquiond, 2011; Boldo et al., 2015). This provides a mature baseline against which we compare the corresponding Lean infrastructure.

Agents in both provers are given tactics for symbolic execution of C and for numerical bounds; we describe them in Appendix C.

6 Experiments

We evaluate whether frontier coding agents can autonomously verify production C end to end using the 100-task ULPBench-Small described in Section 4.2, in both Lean and Rocq.

6.1 Setup

We evaluate Claude Opus 5.5 in Claude Code and GPT-6-Sol as well as GPT-6-Luna in Codex. Agents run autonomously with no Internet access and may freely edit files, invoke the prover, run

tests, and iterate on their solutions. Each agent has a 5 hour limit and runs in an isolated virtual machine with 8 vCPUs and 16 GB of memory. We run the end-to-end variant (T4), in which the agent must construct the model and complete both proof stages, and grade each stage of the submitted files separately to obtain per-stage outcomes (T1–T3). A task counts as solved for an agent and prover if a run passed all three stages within the budget.

6.2 Main results

Table 2 reports end-to-end (T4) success by the origin of the C function. Claude Opus 5.5 solves the most tasks in both provers: 34 of 100 in Lean and 32 in Rocq, compared with 24 and 23 for GPT-6-Sol and 12 and 9 for GPT-6-Luna. The two provers are close for every agent, with Lean ahead by one to three tasks in each case; as discussed below, we interpret this as a property of the complete verification environments on this numerical workload rather than as a general comparison between the proof assistants.

Success depends strongly on where the code comes from. Floating-point math libraries are the hardest category: the best agent solves 10 of 32 such tasks in Lean and 9 in Rocq, and GPT-6-Sol solves 4 in each. Integer arithmetic, random-number and hashing routines are the most tractable, and there GPT-6-Sol matches Claude Opus 5.5 (7 of 16 in Lean, 7 and 6 in Rocq).

Grading the stages of each end-to-end run separately shows where the agents fall short (Table 3). Constructing a correct functional model is the easiest stage for every agent. Claude Opus 5.5 loses few tasks after that point: it completes 34 of the 36 Lean tasks and 32 of the 35 Rocq tasks it models correctly. GPT-6-Sol builds a correct model for 35 Lean and 41 Rocq tasks but completes the refinement proof for only 29 in each prover, and the property proof for 26 and 24. GPT-6-Luna loses most of its modelled tasks at refinement (20 to 12 in Lean, 22 to 9 in Rocq). The refinement proof is also where the time goes: GPT-6-Sol takes a median of over 40 minutes to finish it, against about 20 minutes for its property proofs, whereas Claude Opus 5.5 finishes both in roughly 15–25 minutes.

		Claude Opus 5.5		GPT-6-Sol		GPT-6-Luna	
Origin	n	Lean	Rocq	Lean	Rocq	Lean	Rocq
Floating-point math libraries	32	10	9	4	4	1	1
Scientific / numerical	9	4	4	3	2	3	3
Integer arithmetic, RNG, hashing	16	7	6	7	7	4	3
Signal processing and codecs	20	6	6	4	4	2	1
Systems, parsing, compression	23	7	7	6	6	2	1
Total	100	34	32	24	23	12	9

Table 2: End-to-end (T4) tasks solved out of n on the 100-task evaluated subset, by origin of the C function, five hour budget per task.

	Lean				Rocq			
Agent	T1	T2	T3	T4	T1	T2	T3	T4
Claude Opus 5.5	36	36	36	34	35	33	35	32
GPT-6-Sol	35	29	26	24	41	29	24	23
GPT-6-Luna	20	12	15	12	22	9	11	9

Table 3: Tasks (of 100) on which each stage was achieved within the end-to-end runs: a correct functional model (T1), the refinement proof (T2), the property proof (T3, counted whether or not the refinement was also proved), and all three (T4).

6.3 Test-time compute scaling and cost

The final success rate hides substantial variation in how much agent effort individual proofs require. Figures 2 and 3 plot cumulative end-to-end successes against wall-clock time and token use. Claude Opus 5.5 leads at every budget. Its median successful run takes about 25 minutes in Lean and 30

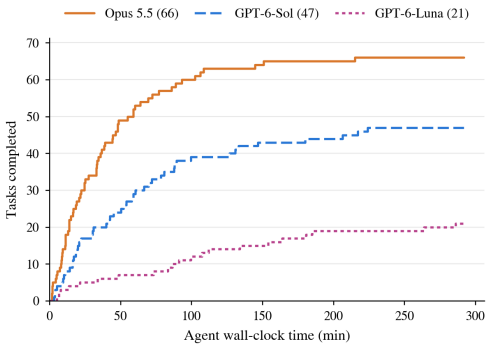


Figure 2: End-to-end tasks completed (both provers, of 200 task–prover pairs) against agent wall-clock time. Each task and prover counts once per agent, at its fastest successful run.

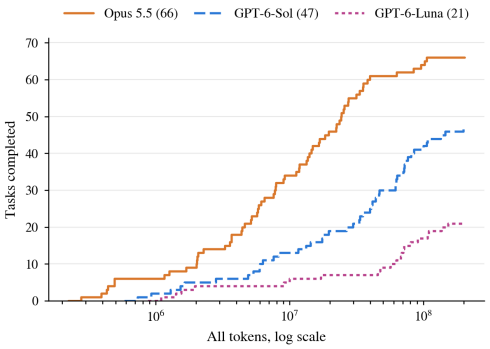


Figure 3: End-to-end tasks completed against total tokens per run (input including cached prompt tokens, plus output), log scale. Each task and prover counts once per agent, at its lowest-token successful run.

Proof mix (refinement share)	n	Lean faster	Rocq faster	Within 10%
Property-heavy ($\leq 30\%$)	23	39%	30%	30%
Mixed (30–70%)	13	46%	38%	15%
Refinement-heavy ($\geq 70\%$)	19	26%	68%	5%
All	55	36%	45%	18%

Table 4: Paired Lean/Rocq comparison on the 55 (agent, task) pairs solved end-to-end in both proof assistants, grouped by the share of agent-written proof lines belonging to the refinement proof. A prover counts as faster when its wall-clock time is at most 90% of the other's; pairs closer than that are counted as within 10%.

in Rocq, against about 50 and 40 for GPT-6-Sol and 110 and 75 for GPT-6-Luna, and it completes nearly all of its solved tasks within two hours. Counting all tokens, including cached prompt tokens, Claude Opus 5.5 reaches the same number of solved tasks with roughly five times fewer tokens than GPT-6-Sol. These aggregate curves, however, conceal important differences in which programs the two environments can prove; we examine these next.

6.4 Analysis

The two verification environments fail for different reasons, and neither dominates. End-to-end success is close between the provers for every agent (Table 2). Among pairs solved in both, Rocq was faster somewhat more often than Lean (45% against 36%), but the split follows the proof mix: Lean was more often faster on property-heavy tasks, and Rocq on refinement-heavy ones (Table 4).

7 Case Study: Defects Found by Proving

LLMs can be powerful in helping to find bugs, however across a large codebase, it is difficult to identify where to focus resources. We used our methodology to identify tasks to focus on for bug identification by ruling out success tasks and ones where the refinement proof did not complete. We collected tasks where no agents were able to close the goals in the property proof during the end-to-end verification. The agents focused on this set were able to identify an error in CPython's `cmath.atanh`, where $>$ should have been $>=$. The issue caused a singular floating point value which should have been ≈ 177.45 to be outputted as `+Inf`. Directly adjacent floating point values were normal (finite), making this bug nearly impossible to find through traditional fuzzing. A CPython maintainer reproduced the defect and confirmed it against two independent implementations: glibc's `catanh` and NumPy both return the correct finite value on the same input. A patch has been accepted and recommended for merge.

7.1 GeographicLib: inverse geodesic accuracy

While the `catanh` finding shows that proof-directed search can reach inputs that fuzzing is unlikely to hit, the function is short enough that an expert tracing its logic by hand could plausibly have found the defect. GeographicLib, from which many ULPBench functions are drawn, is a library of geodesic computations on the WGS-84 ellipsoid that underlies a large share of modern GPS and geospatial software. Its documented accuracy for the inverse geodesic problem is about 15 nm on Earth-scale distances, close to binary64 round-off (Karney, 2013; 2025). This tightness matters because geodesic results feed larger computations, such as intersections and polygon areas, where small errors can propagate. The underlying algorithms are mathematically intricate, with their correctness developed in dedicated papers (Karney, 2011; 2013), and they have been tested extensively, including against 500,000 high-precision reference geodesics that deliberately cover nearly antipodal and nearly equatorial cases (Karney, 2010). While verifying specifications for this library in Lean, an agent instead proved the *negation* of a tolerance specification for the helper `geod_geninverse_int`. On the witness inputs from this proof, `geod_inverse` returns a distance that is off by about 227 μm, more than $15{,}000\times$ the documented bound. The error reproduces in the reference C++ implementation, in its C port, and in PROJ (PROJ contributors, 2026), which bundles that C port.

8 Limitations and Future Work

ULPBench focuses on numerical C, a setting in which programs often have substantial control flow but comparatively simple memory behavior. Our evaluation includes loops, arrays, and pointers, but pointer- and memory-intensive programs are less well represented. In particular, the Lean infrastructure does not yet provide a program logic as expressive or mature as VST's separation-logic-based treatment of memory. Extending the task set to include richer heap-manipulating C programs is therefore an important direction for future work.

Furthermore, both the property proving and refinement stages stand to benefit greatly from a larger lemma library, more tactic tooling which would mirror the Rocq landscape such as Flocq/Gappa, and more agent engineering optimizations such as improved prompts or control flow.

9 Conclusion

We introduced ULPBench, a benchmark for autonomous formal verification of production C that evaluates the full path from source code to a machine-checked property. Rather than beginning with a verification condition or a program already expressed in a proof-oriented language, ULPBench requires agents to construct a functional model, prove that the original C refines that model under a formal C semantics, and prove the desired property of the model. By evaluating these stages separately as well as end to end, the benchmark makes it possible to identify where current agents succeed and where verification breaks down.

To make this evaluation possible in Lean, we developed a Clight and IEEE-754 semantics together with automation for symbolic execution and numerical reasoning, and compare against the mature CompCert/VST ecosystem in Rocq. Our results show that while autonomous agents can already carry out nontrivial source-level verification of real numerical C, including refinement, there is still substantial room for improvement. The case study further demonstrates that the same process can be useful beyond benchmark measurement: proof attempts can expose narrow regions of program behavior that ordinary testing misses and lead to concrete defects in widely deployed numerical code. Taken together, these results suggest that verification of existing systems code is becoming a practical target for autonomous reasoning systems, while leaving substantial room for stronger program logics, broader C coverage, and more capable proof automation.

AI use statement

In this work, generative AI systems were used both as systems under evaluation and as tools in the research workflow. In the experiments described in this paper, LLM agents generated functional models and machine-checkable refinement and property proofs for benchmark tasks. Generated

functional models were evaluated using the differential-testing procedures described in the paper, and generated proofs counted as successful only when accepted by the corresponding Lean or Rocq proof-checking procedure under our stated validation policy.

Outside the evaluation itself, we used generative AI tools to identify candidate open-source numeric C libraries and functions, to assist in constructing benchmark artifacts, to assist in proving theorems within the Clight-Lean development, and for evaluation-code development and implementation of methods. We also used generative AI tools to assist in drafting and editing portions of the manuscript. All AI-assisted research artifacts and manuscript content were reviewed by the authors, and generated code was tested as described in the paper. The authors take responsibility for the final content of this work, including all text, claims, code, proofs, and other artifacts produced with the aid of generative AI.

Reproducibility statement

We provide ULPBench, the evaluation and grading code, task prompts, toolchain versions, and experimental configuration to reviewers as anonymized supplementary material. Appendix E gives the prompts; the benchmark and associated tooling will be released publicly after review

References

Andrew W. Appel. Verified software toolchain. In *Programming Languages and Systems (ESOP 2011)*, volume 6602 of *Lecture Notes in Computer Science*, pp. 1–17. Springer, Berlin, Heidelberg, 2011. doi: 10.1007/978-3-642-19718-5_1.

Andrew W. Appel and Yves Bertot. C-language floating-point proofs layered with VST and Flocq. *Journal of Formalized Reasoning*, 13(1):1–16, 2020. doi: 10.6092/issn.1972-5787/11442.

Andrew W. Appel and Ariel E. Kellison. VCFloat2: Floating-point error analysis in Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 14–29. ACM, 2024. doi: 10.1145/3636501.3636953.

Andrew W. Appel and David A. Naumann. Verified sequential malloc/free. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pp. 48–59. ACM, 2020. doi: 10.1145/3381898.3397211.

Clark Barrett, Swarat Chaudhuri, Fabrizio Montesi, Jim Grundy, Pushmeet Kohli, Leonardo de Moura, Alexandre Rademaker, and Sorrachai Yingchareonthawornchai. Cslib: The lean computer science library, 2026. URL `https://arxiv.org/abs/2602.04846`.

Sandrine Blazy and Xavier Leroy. Mechanized semantics for the clight subset of the c language. *Journal of Automated Reasoning*, 43(3):263–288, Jul 2009. doi: 10.1007/s10817-009-9148-3.

Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *2011 IEEE 20th Symposium on Computer Arithmetic*, pp. 243–252. IEEE, 2011. doi: 10.1109/ARITH.2011.40.

Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. Verified compilation of floating-point computations. *Journal of Automated Reasoning*, 54(2):135–163, 2015. doi: 10.1007/s10817-014-9317-x.

Richard J. Boulton, Andrew Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in hol. In *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pp. 129–156, NLD, 1992. North-Holland Publishing Co. ISBN 0444896864.

Sergiu Bursuc, Theodore Ehrenborg, Shaowei Lin, Lacramioara Astefanoaei, Ionel Emilian Chiosa, Jure Kukovec, Alok Singh, Oliver Butterley, Adem Bizid, Quinn Dougherty, Miranda Zhao, Max Tan, and Max Tegmark. A benchmark for vericoding: Formally verified program synthesis, 2025.

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. VST-Floyd: A separation logic tool to verify correctness of C programs. *Journal of Automated Reasoning*, 61(1–4):367–422, 2018. doi: 10.1007/s10817-018-9457-5.

Sylvain Chevillard, Mioara Joldes, and Christoph Lauter. Sollya: An environment for the development of numerical codes. In *Mathematical Software – ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pp. 28–31. Springer, 2010. doi: 10.1007/978-3-642-15582-6_5.

Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Transactions on Mathematical Software*, 37(1):1–20, 2010. doi: 10.1145/1644001.1644003.

Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Transactions on Computers*, 60 (2):242–253, 2011. doi: 10.1109/TC.2010.128.

John Harrison. Formal verification of floating point trigonometric functions. In *Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pp. 217–233. Springer, 2000. doi: 10.1007/3-540-40922-X_14.

Gerard J. Holzmann. Mars code. *Communications of the ACM*, 57(2):64–73, 2014. doi: 10.1145/2560217.2560218.

IEEE. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2019, 2019.

Vincenzo Innocente and Paul Zimmermann. Accuracy of mathematical functions in single, double, double extended, and quadruple precision, 2021. HAL preprint hal-03141101, September 2021 version.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi: 10.1145/3158154. URL https://doi.org/10.1145/3158154.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018. doi: 10.1017/s0956796818000151.

Charles F. F. Karney. Test set for geodesics, 2010. Dataset.

Charles F. F. Karney. Geodesics on an ellipsoid of revolution, 2011.

Charles F. F. Karney. Algorithms for geodesics. *Journal of Geodesy*, 87(1):43–55, 2013. doi: 10.1007/s00190-012-0578-z.

Charles F. F. Karney. *GeographicLib, version 2.7*, 2025. URL https://geographiclib.sourceforge.io/.

Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, 2010. doi: 10.1145/1743546.1743574.

Soonho Kong, Sicun Gao, and Edmund M. Clarke. Floating-point bugs in the embedded GNU C library. Technical Report CMU-CS-13-130, School of Computer Science, Carnegie Mellon University, 2013.

Robbert Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University Nijmegen, 2015. URL https://robbertkrebbers.nl/research/thesis.pdf.

Lean FRO. Validating a lean proof. https://lean-lang.org/doc/reference/latest/ValidatingProofs/. Lean Language Reference.

Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of standard ml. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 173–184, 2007. doi: 10.1145/1190215.1190245.

Wonyeol Lee, Rahul Sharma, and Alex Aiken. On automatically proving the correctness of math.h implementations. *Proceedings of the ACM on Programming Languages*, 2(POPL), 2018. doi: 10.1145/3158135.

Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538814. URL https://doi.org/10.1145/1538788.1538814.

Jay P. Lim and Santosh Nagarakatte. High performance correctly rounded math libraries for 32-bit floating point representations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 359–374. ACM, 2021. doi: 10.1145/3453483.3454049.

Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. DafnyBench: A benchmark for formal software verification. *Transactions on Machine Learning Research*, 2025.

Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning*, 57(3):187–217, 2016. doi: 10.1007/s10817-015-9350-4.

Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the depths of C: Elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pp. 1–15, New York, NY, USA, 2016. Association for Computing Machinery. doi: 10.1145/2908080.2908081. URL https://doi.org/10.1145/2908080.2908081.

Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2 edition, 2018. doi: 10.1007/978-3-319-76526-6.

PROJ contributors. *PROJ coordinate transformation software library*. Open Source Geospatial Foundation, 2026. URL https://proj.org/.

Tahina Ramananandro, Paul Mountcastle, Benoît Meister, and Richard Lethin. A unified coq framework for verifying c programs with floating-point computations. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pp. 15–26, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341271. doi: 10.1145/2854065.2854066. URL https://doi.org/10.1145/2854065.2854066.

Peter Schwabe, Benoît Viguier, Timmy Weerwag, and Freek Wiedijk. A coq proof of the correctness of x25519 in TweetNaCl. Cryptology ePrint Archive, Paper 2021/428, 2021. URL https://eprint.iacr.org/2021/428.

Prokhor Shlyakhtun, Alexander Gryzlov, Vladimir Kukharenko, Vasilii Nesterov, Nikolai Vasiliev, Kirill Ziborov, Eugene Zolotarev, and Alex Pokras. Agent-driven verification of memory safety for liblzma decoder components with VST, 2026.

Alexei Sibidanov, Paul Zimmermann, and Stéphane Glondu. The CORE-MATH project. In *2022 IEEE 29th Symposium on Computer Arithmetic*, pp. 26–34. IEEE, 2022. doi: 10.1109/ARITH54963.2022.00014.

Alexey Solovyev, Marek Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. *ACM Transactions on Programming Languages and Systems*, 41(1):2:1–2:39, 2018. doi: 10.1145/3230733.

Amitayush Thakur, Jasper C. H. Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetzsche, Greg Durrett, Yisong Yue, and Swarat Chaudhuri. CLEVER: A curated benchmark for formally verified code generation. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-3857.

TIOBE Software BV. TIOBE index for September 2026. https://www.tiobe.com/tiobe-index/, 2026. Accessed 2026-09-21.

Qiyuan Xu, Xiaokun Luan, Renxi Wang, Joshua Ong Jun Leang, Peixin Wang, Haonan Li, Wenda Li, and Conrad Watt. Neural theorem proving for verification conditions: A real-world benchmark. In *The Fourteenth International Conference on Learning Representations*, 2026.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011. ISSN 0362-1340. doi: 10.1145/1993316.1993532. URL https://doi.org/10.1145/1993316.1993532.

Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. VERINA: Benchmarking verifiable code generation. In *The Fourteenth International Conference on Learning Representations*, 2026.

A Further Related Work: Bringing Programs into Proof Assistants

We build on a long body of work in mechanizing programming language semantics in proof assistants. Some recent work in this direction includes the RustBelt (Jung et al., 2017) deep embedding of Rust within Rocq using the Iris concurrent separation logic (Jung et al., 2018); the Cerberus C semantics Memarian et al. (2016), which elaborates C into an extended language that makes ambiguities in the C standard explicit; and the VCFloat library for floating point verification (Ramananandro et al., 2016).

Recently, there has been increased interest in bringing this sort of verification infrastructure to Lean; the Aeneas project translates the safe subset of Rust directly to Lean, essentially eliding the refinement gap. This elision is dangerous for languages such as C, for which there is no natural correspondence to Lean. Finally, the recently-announced Lean CSLib project (Barrett et al., 2026) seeks to foster a comprehensive body of frameworks for language mechanization and program verification with common, interoperable logical foundations. One of its aims is to create a single, unified imperative language to enable more straightforward translation from targets such as C and unsafe Rust into Lean. A general-purpose foundation like CSLib would eventually replace purpose-built embeddings like ours, although until it exists and is validated, our Clight-based embedding gives ULPBench a working C model in Lean in the present.

B Semantics Port: Audit and Deviations from CompCert

B.1 Validating the port

To empirically validate trust in the translation of Clight in Lean, we perform differential fuzzing between CompCert and Clight-Lean. For generation and mutation of a large corpus of different C programs, we use Csmith (Yang et al., 2011),

Over 7000 randomly-generated C programs, for a total of over 2000 CPU hours, we observed zero disagreements between programs compiled by CompCert versus those executed by Clight-Lean. Our Csmith configuration is given in Section B.4.

B.2 Trusted computing base

The CompCert C parser is formally verified to accept the C grammar. However, the backend which exports the internal Clight AST as Lean syntax must be trusted. In addition, the C semantics embedding itself (both the original Rocq definition and our Lean translation) is part of the TCB - there is inherently no formal mathematical object that exists to verify them against.

The TCB includes the proof-checking kernels for both Rocq and Lean. The native_decide tactic requires trust of the whole Lean compiler, not just the kernel, and was thus disallowed for a final validation pass.

B.3 Code Artifact

Clight-Lean can be found at (github link), we are in the process of releasing under an open-source license.

B.4 Differential Fuzzing

csmith can generate C programs of varying size, complexity, and features, without undefined behavior. Most programs are several hundred LOC and exercise a large set of C language features exceeding that found in our task corpus. csmith has been previously used for differential fuzzing between several major C compilers, including CompCert. While several bugs were found in LLVM and GCC, and one in CompCert's embedding of the PowerPC instruction set, no bugs were found in CompCert's C frontend or semantics. This, in combination with the generally high confidence in the verification community that CompCert is trustworthy, is strong empirical evidence of CompCert's correctness and suitability to serve as an oracle for differential fuzzing.

The following parameters were used for program generation with csmith:

Data Types

- `--float`: Enable floating-point types
- `--strict-float`: Enforce strict floating-point semantics
- `--no-volatiles`: Disable volatile qualifiers
- `--no-bitfields`: Disable bitfield generation
- `--no-builtins`: Disable compiler builtin functions

Structures

- `--max-struct-fields 6`: Maximum 6 fields per structure
- `--no-packed-struct`: Disable packed structure attributes
- `--no-arg-structs`: Disable structures as function arguments
- `--no-return-structs`: Disable structures as return values

Unions

- `--max-union-fields 3`: Maximum 3 fields per union
- `--no-arg-unions`: Disable unions as function arguments
- `--no-return-unions`: Disable unions as return values
- `--take-no-union-field-addr`: Disable taking addresses of union fields

Pointers and Arrays

- `--max-pointer-depth 3`: Maximum pointer indirection depth of 3
- `--no-volatile-pointers`: Disable volatile pointer qualifiers
- `--no-const-pointers`: Disable const pointer qualifiers
- `--max-array-dim 3`: Maximum 3 array dimensions
- `--max-array-len-per-dim 4`: Maximum length of 4 per dimension

Program Structure

- `--no-argc`: Disable command-line argument processing
- `--max-funcs 3`: Maximum 3 functions per program
- `--max-block-depth 3`: Maximum nesting depth of 3 for code blocks
- `--max-block-size 3`: Maximum 3 statements per block
- `--max-expr-complexity 5`: Maximum expression complexity of 5

OUTPUT OPTIONS

- `--checksum`: Enable checksum generation for validation
- `--no-hash-value-printf`: Disable hash value printing
- `--concise`: Generate concise output format

C PROOF AUTOMATION PROVIDED TO AGENTS

Agents are given proof automation for the two recurring forms of reasoning in our tasks: symbolic execution of C and numerical bound tactics. On the Lean side, our `forward` tactic symbolically executes Clight statements under our executable semantics, propagating symbolic program states and reducing refinement goals to the resulting semantic and arithmetic obligations. Function calls can be handled through explicit callee contracts, while the remaining numerical goals are typically discharged by `interval_bound`.

Our `interval_bound` tactic is a reflective interval tactic for proving bounds on real-valued expressions. It computes interval enclosures using exact dyadic arithmetic, adaptively bisects the input domain when necessary, and checks the result through a small kernel-checkable certificate. Its role is broadly similar to established interval tactics in Rocq such as CoqInterval (Martin-Dorel & Melquiond, 2016), though our implementation is specialized to the needs of this benchmark.

On the Rocq side, agents have access to the corresponding mature ecosystem: VST/Floyd's `forward` tactics for symbolic execution and tools such as CoqInterval, Gappa, and VCFloat for numerical reasoning (Martin-Dorel & Melquiond, 2016; Daumas & Melquiond, 2010; Appel & Kellison, 2024). These tools are not identical across the two provers, and our experiments should therefore be interpreted as comparisons of the complete verification environments rather than of the proof-assistant kernels in isolation.

D UNITS IN THE LAST PLACE

Let p be the precision of a binary floating-point format ($p = 53$ for binary64, $p = 24$ for binary32) and $e_{\min}$ its least normal exponent (-1022 and -126 respectively). For a real number x we define

$$\mathrm{ulp}(x) = \begin{cases} 2^{\,e_{\min}-p+1} & \text{if } |x| < 2^{e_{\min}}, \\ 2^{\,\lfloor \log_2 |x| \rfloor - p+1} & \text{otherwise.} \end{cases}$$

Above the normal threshold this is the spacing of the floating-point numbers in the binade containing x; at zero and throughout the subnormal range it is the fixed subnormal quantum (2^{-1074} for binary64). For binary64 this is the definition `Spec.ulp64` used in the Lean statements.

Conventions. The ulp is always measured at the *exact* real result, not at the computed floating-point value. An accuracy claim of k ulp for a function f implemented by $\hat{f}$ states that for every finite input x in the stated domain, $\hat{f}(x)$ is finite and

$$|\hat{f}(x) - f(x)| \le k \cdot \mathrm{ulp}(f(x)),$$

with $<$ in place of $\le$ where the source claims a strict bound (for example "the error is always less than 1 ulp"). A NaN or infinite result never satisfies the bound. Inputs are quantified explicitly, together with any domain restriction the source states, such as excluding inputs whose exact result overflows. A correctly rounded function satisfies the bound with $k = 1/2$. Specifications that are not error bounds, such as exactness or range claims, are stated directly as theorems about the model.

E AGENT PROMPTS

Every run receives a single Markdown prompt rendered from a per-variant template (`model`, `refine`, `property`, or `all`). Templates use {`name`} placeholders: per-task values (task identifier, function name, frozen theorem names, allowed axioms) come from the dataset row, and per-prover values come from a fixed table (Table 5); the time limit is filled in from the run configuration.

The `Testing kit` bullet is included only for tasks whose model is checked by differential testing, since only those tasks ship the kit. For Rocq tasks whose function calls internal helper functions, the {`helper_rule`} placeholder adds a fourth rule asking the agent to give each helper a specification and to prove its body; otherwise it is empty. All agents are run in goal mode (Claude Code's `/goal` and the Codex goal loop), with the prompt's Goal section plus the final-message requirement as the goal objective and the full prompt as task context.

Placeholder	Lean	Rocq
`arm_stack`	Lean 4.32, CLightLean, Mathlib	Rocq 9.1.1, CompCert 3.17, VST 2.16, Flocq 4.2.2
`project_dir`	`/work/CLightLean`	`/work/vst-eval`
`task_file`	`Task/All.lean`	`Task/All.v`
`hole`	`sorry`	`Admitted.`
`check_cmd`	`lake env lean Task/All.lean`	`coqc -Q . Task All.v`
`axiom_cmd`	`#print axioms`	`Print Assumptions`
`import_word`	imports	`Require` lines
`thm_word`	theorem	lemma
`libraries`	Mathlib; CLightLean; Interval (`interval_bound`)	Flocq, CompCert, VST; Coquelicot; Interval (`interval`), Gappa (`gappa`), `native_compute`

Table 5: Per-prover values substituted into the prompt templates.

Template for the end-to-end (`all`) variant.

```
# Task {task_id}: model, refinement and property ({arm_name})

## Goal
In `{task_file}`:
1. Replace the {hole} of `{model_name}` (section 2) with a functional model of the C function
↪  `{function}`.
2. Prove `{refine_name}` (section 3): the C function, given as its Clight AST in section 1,
↪  returns exactly the value of your model.
3. Prove `{property_name}` (section 4): your model satisfies the stated property.

Translating the C algorithm *faithfully*, using the same general logic, helps make the
↪  refinement theorem easier to prove.

## Setup
- Project: `{project_dir}` ({arm_stack}). Your task file is `{task_file}`; `Task/unit.c` is
↪  the C source.
- Check that the file compiles: {check_cmd}
- Libraries: {libraries}
- Testing kit in `Task/`: `Eval.{ext}` runs your model on a list of inputs, `c_eval.c` runs
↪  the C function on the same inputs, `check.sh` compares the two on `Task/inputs.txt` or on
↪  a file you pass as its argument, and `inputs.txt` holds special values and random inputs.
↪  You may modify these files or write your own tests; they are not graded.
- The project is a local git repository with one baseline commit.
- Do not access the internet.
- Time limit: {time_limit}. When it runs out, or when you finish, `{task_file}` as it is at
↪  that moment is graded.

## Rules
1. Only `{task_file}` is graded. Keep the name and type of `{model_name}`. Do not change
↪  section 1 or the statements of `{refine_name}` and `{property_name}`. You may add
↪  definitions and lemmas anywhere before their first use, and add {import_word}.
2. The file must compile, and the axioms each of `{model_name}`, `{refine_name}` and
↪  `{property_name}` depends on (as listed by {axiom_cmd}) must be among: {allowed_axioms}.
3. If you believe the property is false for the C function, you may instead prove its
↪  negation, as a separate {thm_word} named `{property_name}_false`.{helper_rule}

## Grading
Each of the three is graded separately: the file compiles, the given parts are unchanged, and
↪  the axioms are within the allowed list. The property counts only if the refinement is
↪  also proved.

## Final message
End your final message with this JSON object, in a ```json block:

```json
{"complaint": null, "details": ""}
```

Set `"complaint"` to `"spec_invalid"` if you believe the stated property does not hold for
↪ the C function, and give the reason in `"details"`, with a concrete input if you have
↪ one. Otherwise leave it `null`.
```

Example: `redis/interleave64`, end-to-end, Lean. The prompt exactly as issued (the Rocq prompt for the same task differs only in the per-prover values of Table 5, the Rocq theorem names, and the Rocq axiom allowlist). The one non-ASCII character, $\leq$, is shown as <=.

```
# Task redis/interleave64: model, refinement and property (Lean)

## Goal
In `Task/All.lean`:
1. Replace the `sorry` of `Task.interleave64` (section 2) with a functional model of the C
↪  function `interleave64`.
2. Prove `Task.interleave64_refines` (section 3): the C function, given as its Clight AST in
↪  section 1, returns exactly the value of your model.
3. Prove `Task.interleave64_correct` (section 4): your model satisfies the stated property.

Translating the C algorithm *faithfully*, using the same general logic, helps make the
↪  refinement theorem easier to prove.

## Setup
- Project: `/work/CLightLean` (Lean 4.32, CLightLean, Mathlib). Your task file is
↪  `Task/All.lean`; `Task/unit.c` is the C source.
- Check that the file compiles: `lake env lean Task/All.lean` (from `/work/CLightLean`)
- Libraries: Mathlib; CLightLean; Interval (`import Interval.Tactic`), whose `interval_bound`
↪  tactic proves bounds such as `|e| <= c` on real expressions built from `+ - * / ^`,
↪  `Real.exp`, `Real.log`, `Real.sin`, `Real.cos`, `Real.sqrt` and others, given numeric
↪  bounds on the variables (see `Interval/README.md`).
- Testing kit in `Task/`: `Eval.lean` runs your model on a list of inputs, `c_eval.c` runs
↪  the C function on the same inputs, `check.sh` compares the two on `Task/inputs.txt` or on
↪  a file you pass as its argument, and `inputs.txt` holds special values and random inputs.
↪  You may modify these files or write your own tests; they are not graded.
- The project is a local git repository with one baseline commit.
- Do not access the internet.
- Time limit: 300 minutes. When it runs out, or when you finish, `Task/All.lean` as it is at
↪  that moment is graded.

## Rules
1. Only `Task/All.lean` is graded. Keep the name and type of `Task.interleave64`. Do not
↪  change section 1 or the statements of `Task.interleave64_refines` and
↪  `Task.interleave64_correct`. You may add definitions and lemmas anywhere before their
↪  first use, and add imports.
2. The file must compile, and the axioms each of `Task.interleave64`,
↪  `Task.interleave64_refines` and `Task.interleave64_correct` depends on (as listed by
↪  `#print axioms`) must be among: `propext`, `Classical.choice`, `Quot.sound`,
↪  `Lean.ofReduceBool`, `Events.external_functions_sem`, `Events.inline_assembly_sem`,
↪  `Lean.trustCompiler`, `*._native.*`.
3. If you believe the property is false for the C function, you may instead prove its
↪  negation, as a separate theorem named `Task.interleave64_correct_false`.

## Grading
Each of the three is graded separately: the file compiles, the given parts are unchanged, and
↪  the axioms are within the allowed list. The property counts only if the refinement is
↪  also proved.

## Final message
End your final message with this JSON object, in a ```json block:

```json
{"complaint": null, "details": ""}
```

Set `"complaint"` to `"spec_invalid"` if you believe the stated property does not hold for
↪ the C function, and give the reason in `"details"`, with a concrete input if you have
↪ one. Otherwise leave it `null`.
```

F Full Results

G Successes