From Specification to Kernel Commit: Verified Code Generation on Real-World Systems

Hao Sun
ETH Zurich
Switzerland
hao.sun@inf.ethz.ch

Zenan Li
ETH Zurich
Switzerland
zenan.li@inf.ethz.ch

Cong Li
ETH Zurich
Switzerland
cong.li@inf.ethz.ch

Zhendong Su
ETH Zurich
Switzerland
zhendong.su@inf.ethz.ch

Abstract

Verified code generation, in which an LLM produces both an implementation and a correctness proof, offers a promising path toward correct-by-construction software. Prior evaluations report that LLM capability remains unsatisfactory, but they test models in a bare-model setting that no longer reflects how LLMs are deployed in practice, *e.g.*, as coding agents. We re-evaluate existing benchmarks and find that coding agents solve nearly all tasks derived from well-studied sources such as programming contests.

This raises the question of whether such success extends to real-world systems tasks. We present Vero, a benchmark of 103 verified code generation tasks drawn from three security- and correctness-critical projects: the Linux kernel eBPF verifier, the LLVM compiler, and the seL4 microkernel. Each task is real-world, encodes a new specification, and is valuable to solve in that a correct solution directly benefits the upstream project. We evaluate three coding agents; the strongest solves only 3 of the 103 tasks, establishing that real-world verified code generation remains an open problem. Notably, one agent-discovered solution to a Vero task has been merged into the Linux kernel, replacing a 20-line algorithm proposed by domain experts with a 4-line provably correct one. The unsolved tasks provide concrete targets for future work.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; **Automatic programming**; • **Computing methodologies** → *Machine learning*.

Keywords: verified code generation; LLM agents

ACM Reference Format:
Hao Sun, Zenan Li, Cong Li, and Zhendong Su. 2027. From Specification to Kernel Commit: Verified Code Generation on Real-World Systems. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27), April 11–15, 2027, Heraklion, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3845814.3850060

ASPLOS '27, Heraklion, Greece

ACM ISBN 979-8-4007-3012-2/2027/04
https://doi.org/10.1145/3845814.3850060

1 Introduction

In 1976, Dijkstra argued that correctness should be established by construction rather than post-hoc testing, advocating the derivation of programs from formal specifications via weakest preconditions [16]. The idea of mechanizing this process has deep roots in the program synthesis literature [36, 37]: the programmer expresses *what* the program should do via a formal specification, and the machine derives *how*, producing both an implementation and a machine-checkable proof of its correctness. Classical approaches, however, struggled with intractable search spaces [3, 24, 54]. The advent of LLM agents introduces a promising new instantiation of this vision, giving rise to what the community terms *verified code generation* [63, 72, 74, 75]. In this paradigm, the programmer supplies a formal specification S, and the agent produces an implementation C together with a proof P certifying $C \models S$. Recent work explores this direction across multiple verification frameworks [10, 21, 27, 44, 57, 73].

Verified code generation offers two advantages over conventional code generation. First, the review burden shifts from inspecting low-level implementation details to validating a concise, high-level specification against the programmer's intent, a far more tractable task (Figure 1). Second, correctness is established mechanically by a proof checker rather than by subjective and error-prone manual review. Conventional code generation, by contrast, requires the reviewer to reverse-engineer functionality from generated code and manually verify it against the intent. As agents produce ever-larger volumes of code, this costly and unreliable process becomes a bottleneck. We argue that *only a machine can keep pace with another machine*: delegating correctness to the proof checker is a principled path to bridging the gap between large-volume generation and code reliability.

Existing benchmarks are solved by agents. We survey prior benchmarks for verified code generation and focus on the two most recent: Verina [74] (189 Lean 4 tasks) and Clever [63] (161 Lean 4 tasks). Both derive specifications from well-studied problems (*e.g.*, HumanEval, MBPP, competitive programming) by stripping away the implementation and requiring the model to regenerate both the code and a correctness proof. Their original evaluations, conducted under bare-model, few-shot prompting, reported low success rates: only 1/161 end-to-end solutions on Clever and a solving rate of at most 3.2% on Verina.

These benchmarks have been valuable in advancing the field. Their evaluations, however, no longer reflect how LLMs are used in practice—as *autonomous coding agents*. Such agents operate in agentic loops, steering LLMs to iteratively interact with the harnessing environment that extends the bare model's capability through skills, MCP servers, and domain-specific tools, *e.g.*, compilers and proof assistants. We re-evaluate both benchmarks using state-of-the-art coding agents equipped with the latest models and integrated with a Lean 4 MCP server and proof-writing skills (Section 3). The results are promising: the agent achieves a 95.8% pass rate on well-formed Clever tasks and solves all 189/189 tasks in Verina. On tasks whose specifications derive from well-studied problems, current agents are already effective.

Can agents solve real-world tasks? The natural follow-up is whether this success extends to novel, real-world system tasks, where neither the implementation nor the proof exists. We present Vero[1], a benchmark of 103 verified code generation tasks constructed around three design principles:

- **Real-world**: every task comes from a real-world system, rather than from a programming exercise or contest.
- **New specification**: each specification encodes a new requirement for which no implementation exists in the system or in publicly available sources.
- **Valuable to solve**: a correct solution directly benefits the upstream system, *e.g.*, by improving the soundness or precision of a critical component.

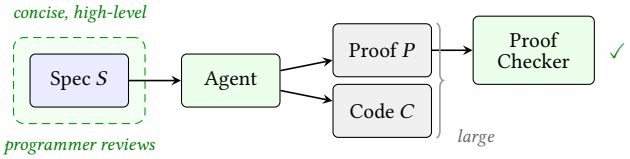
We draw tasks from three mature, security- and correctness-critical systems: the Linux kernel eBPF verifier [23, 56, 67], the LLVM compiler infrastructure [31], and the seL4 verified microkernel [28]. Drawing on our firsthand development experience with these systems, we identified open problems, formalized the expected behavior as Lean 4 specifications [41], and curated the resulting 103 tasks.

To illustrate the value of a single task, consider the eBPF verifier, a critical component of the Linux kernel that performs abstract interpretation [13] over eBPF programs before loading them [55]. An unsound operator can introduce vulnerabilities such as unsafe memory reads from loaded programs [1, 60, 61]; an imprecise one incorrectly rejects safe programs [11, 20, 51]. We therefore cast each abstract operator as a task whose specification demands that the operator be *sound yet strictly more precise* than its current implementation. When an agent solves such a task, the result is an operator that is (a) provably sound, introducing no new vulnerabilities, and (b) more precise—a concrete improvement to the system, not merely a benchmark data point.

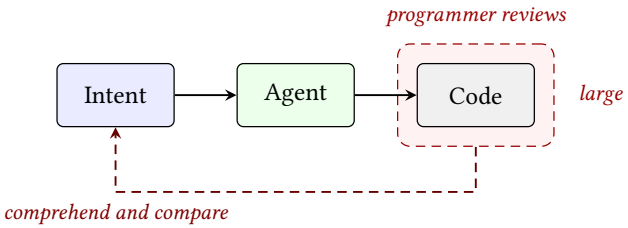
Real-world tasks pose significant challenges. We evaluate three state-of-the-art coding agents: Claude Code, OpenAI Codex, and Gemini CLI, each with its latest model. Claude Code performs best, solving 3/103 tasks. Because every specification is new, the unsolved tasks in turn expose capability gaps that existing benchmarks cannot surface. In addition, Vero poses a *unique challenge*: each task demands that the agent diagnose the algorithmic barrier, derive a novel implementation, and discharge its proof, balancing effort across the three. Even so, one result stands out: for one Vero task, Claude Code produces a 4-line provably correct algorithm that replaces a 20-line version independently proposed by domain experts. This solution has been accepted and merged into the Linux kernel [58], enhancing the eBPF verifier and demonstrating Vero's practical value.

We believe Vero serves as a practical bridge between the verified code generation community and the systems it aims to improve, and we will continue contributing the solutions it yields back to the upstream projects.

[1]Vero is publicly available at https://github.com/SunHao-0/Vero.



(a) Verified code generation: the programmer reviews only a concise specification; correctness is delegated to the proof checker.



(b) Conventional code generation: the programmer must inspect all generated code and compare it against the original intent, a burden that grows with code volume.

Figure 1. Verified vs. conventional code generation.

Contributions. In summary, our contributions are:

- We re-evaluate existing benchmarks under state-of-the-art coding agents, showing that current agents effectively solve tasks derived from well-studied problems and establishing a stronger baseline for the field.
- We present Vero, 103 tasks drawn from real-world systems, where each specification encodes a new requirement and a correct solution benefits the upstream system.
- We evaluate three agents on Vero, revealing capability gaps on novel systems tasks while showing that agents can already produce solutions of sufficient quality.

2 Background

Verified code generation. Figure 1 depicts the verified code generation (VeriCG) workflow. The process involves three participants: the *programmer*, who authors the formal specification; the *agent*, which generates both the implementation

and the correctness proof; and the *proof checker*, which mechanically validates the proof against the specification. The proof checker's verdict is the criterion for correctness: once the proof type-checks against the specification, the implementation is correct by construction.

VeriCG with bare models. Early work [57, 63, 74] operates the agent in a bare-model manner by few-shot prompting: the prompt contains the specification together with a handful of worked examples, and the model emits a complete implementation and proof in a few passes. Evaluation typically grants the model a small budget of attempts per task and reports whether any of them succeed, with limited feedback flowing between attempts. The model's competence is bounded by what it has internalized and by what the prompt supplies; within a single attempt, it cannot query a compiler or inspect the proof state to decide what to try next.

VeriCG with coding agents. A coding agent wraps the model in an iterative loop that drives it toward a goal. Within each turn, the model may reason about the current state, decide which tool to invoke, and consume the returned information; across turns, it may edit the implementation, invoke the proof checker, query a language server for the remaining goal, or search a lemma library such as Mathlib for a relevant result. The proof checker is no longer a one-shot grader but an oracle whose error messages and residual goals guide the next action. The loop terminates when the proof checker accepts the artifact, the agent declares failure, or a predefined budget (wall-clock time, or monetary cost) is exhausted.

Lean 4. Lean 4 is an interactive theorem prover [41] that exposes the proof state after every step, so an agent can proceed within a feedback loop. We encode every task in this single language for three reasons. First, one language establishes a common baseline: results are comparable across our three domains and against prior benchmarks [63, 74]. Second, a task's difficulty then reflects the underlying algorithmic and proof problem rather than the proof language: state-of-the-art provers [9, 35, 50] approach saturation on the standard miniF2F benchmark (99.6% for Seed-Prover and 90.4% for Goedel-Prover-V2), and current agents nearly saturate the prior Lean 4 benchmarks under our harness (Section 3); the low pass rate on Vero therefore reflects the problems themselves. Third, Lean 4 has mature agentic support: a community-maintained MCP server [18] and proof-writing skills [22] expose the goal at any line and provide lemma search. Encoding every task in Lean 4 therefore lets us measure the current frontier of agent capability on VeriCG tasks.

Target systems. Vero draws tasks from three projects.

- *eBPF* allows user space to inject extensions into the Linux kernel for tasks such as performance profiling [43, 46], security monitoring [4, 26], and so on [25, 65]. The verifier performs abstract interpretation [13] to ensure extension safety [23, 56]. It is widely deployed, *e.g.*, Meta runs over a hundred extensions on its servers [12, 19, 38].
- *LLVM* is among the most widely adopted compiler infrastructures, supporting both classical and emerging languages [31]. Formally verifying optimization components yields stronger guarantees against miscompilation, and higher precision translates into better machine code.
- *seL4* is a formally verified microkernel whose functional correctness has been machine-checked from the abstract specification down to the C implementation, and is deployed in various security-critical domains [28].

Table 1. Public benchmarks for verified code generation.

Benchmark	Checker	# Tasks	Source
CLEVER [63] (NeurIPS '25)	Lean 4	161	HumanEval
VERINA [74] (ICLR '26)	Lean 4	189	MBPP, competitive program.
VERICODING [6]	Multi	12,504	Aggregated translations
ALGOVERI [75]	Multi	77	Classical algorithms
FVAPPS [17]	Lean 4	4,715	APPS (auto-translated)
VERIFYTHISBENCH [15]	Multi	154	VerifyThis competition
VERIBENCH [39]	Lean 4	113	HumanEval, algorithms
Vero (ours)	Lean 4	103	**eBPF, LLVM, seL4 (new specs)**

3 Re-evaluation of Existing Benchmarks

Table 1 surveys the existing benchmarks for verified code generation. We select CLEVER [63] and VERINA [74] for evaluation because: (1) both target Lean 4, our proof assistant of choice, and provide high-quality, manually curated specifications; and (2) both define end-to-end verified code generation tasks that require producing an implementation together with a correctness proof. VERICODING [6] aggregates tasks by translating existing coding benchmarks (including HumanEval, for which CLEVER already provides manually written specifications), making its Lean 4 subset redundant. FVAPPS [17] relies on automatic translation from Python, raising specification quality concerns. The remaining benchmarks, *e.g.*, VERIFYTHISBENCH [15], were not publicly available prior to our evaluation. All of these benchmarks derive their tasks from well-studied problems.

Setup. We evaluate both benchmarks under the same agentic harness that we build (details in Section 4). The harness executes the agent in a container in headless mode and enforces our integrity checks, which verify that the specification remains unchanged and that the submitted proof actually discharges it. The agent, Claude Code driven by Opus-4.6, is provided with the usual shell and editing tools, together with a Lean 4 MCP server [18] and a set of proof-writing skills [22]. Each task is allocated a *one-hour* wall-clock budget and a *$15* cost cap, comparable to the effort a domain expert would invest, within which the agent iterates over a code-verify-refine loop: it edits the implementation, invokes the proof checker, and revises in response to the feedback. A recent study also reports that this general-purpose agentic strategy outperforms hand-crafted task-specific workflows when paired with state-of-the-art models [71].

Table 2. Re-eval. of CLEVER and VERINA (Agentic Setting)

Benchmark	Tasks	# Pass	# Spec Err	# Timeout	% Pass*
CLEVER	161	136	19	6	95.8%
VERINA	189	189	0	0	100.0%
Combined	350	325	19	6	98.2%

*Excluding tasks with known specification errors.

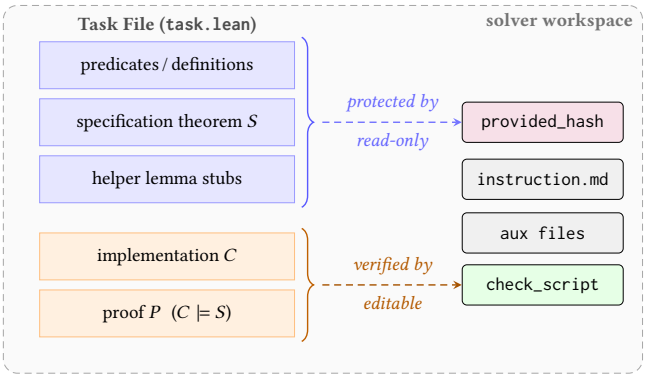
Table 3. Statistics for solved instances (*min* / *avg* / *max*). A *turn* is one round trip in the agentic loop: the model produces tool calls, the agent executes them and returns the results; turns repeat until the model responds with no tool calls.

Metric	CLEVER	VERINA
Solve time (min)	4.1 / 18.8 / 60.0	3.1 / 10.2 / 60.0
Turns	9 / 56 / 226	7 / 31 / 144
Cost (USD)	$0.11 / $2.60 / $14.40	$0.07 / $1.22 / $11.25
Output tokens (K)	1.0 / 33.1 / 176.9	0.6 / 18.8 / 188.7
Total cost	$353	$229
Total time (hr)	56.7	32.0

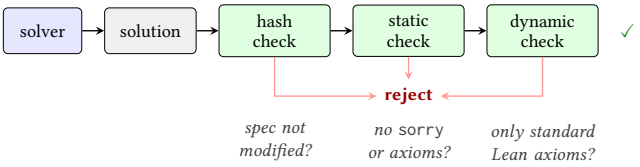
Results. Table 2 summarizes the results. The agent solves 136/161 tasks on CLEVER and 189/189 on VERINA. Of the unsolved CLEVER tasks, 19 stem from specification defects in the benchmark itself (*e.g.*, input constraints stated as theorem conclusions rather than as preconditions, rendering the goal unprovable), and 6 exceed the time budget. Excluding the defective specifications, the adjusted pass rate on well-formed tasks reaches 95.8%. On VERINA, no task times out, and we identify no specification defects. These numbers differ from those reported in the original evaluations under the bare-model setting. CLEVER reported 1/161 end-to-end solutions, and VERINA reported a proof rate below 3.2% even with up to 64 refinement steps. Both prior evaluations used earlier-generation models (*e.g.*, GPT-4o, Sonnet 3.7), so the observed gap reflects not only our agentic harness but also the progress of the underlying models. Under our setup, the agent solves 98.2% of the well-formed tasks.

Table 3 reports per-task resource consumption. The average solve time is 13.8 minutes, and the full evaluation over all 350 tasks costs $582. The solved artifacts average 35.5 lines of proof against 4.4 lines of code, a 8.0× proof-to-code ratio, *i.e.*, verification dominates the difficulty of such tasks.

Implication. Under the agentic harness, current models can produce end-to-end proofs for the majority of well-formed tasks in these existing benchmarks. A natural concern is data leakage: both CLEVER and VERINA derive from HumanEval, MBPP, and competitive-programming archives, whose reference implementations are widely available. Two observations argue that leakage *alone* does not account for the results. First, the bare-model numbers reported by CLEVER and VERINA themselves are low, indicating that the models cannot



(a) Each task is a self-contained workspace, grouped into read-only (protected) and editable sections (verified).



(b) Submitted solutions pass through three verification stages; failure at any stage results in rejection.

Figure 2. Vero task and the verification pipeline.

directly recall a complete specification, implementation, and proof artifact end-to-end. Second, under our harness, the agent still requires on average 41 turns per task, with 41% of that effort devoted to proof refinement (refining and revising the proof after failed attempts), a trace more consistent with interactive proof construction than with verbatim recall. Nevertheless, these benchmarks leave open how an agent performs on new specifications from real-world, production-scale systems. Vero targets this gap.

4 Vero Design

This section describes the construction of Vero following three core principles: every task must be *real-world*, encode a *new specification*, and be *valuable to the upstream project.*

4.1 Task Form and Construction

Task form. Each Vero task poses an unsolved algorithm-and-proof problem drawn from a real system as a Lean 4 correctness theorem: the agent must supply both an implementation and a machine-checkable proof that the implementation satisfies the theorem. Each task is formalized independently and packaged as a self-contained Lean 4 file that contains all relevant definitions and predicates. The file captures the core algorithmic problem rather than situating it in the system's own codebase (*e.g.*, Isabelle/HOL for seL4 or C for the eBPF verifier): it gives the concrete semantics, the abstract-domain semantics, and the correctness obligations that relate them, so a task exercises the agent's reasoning about the problem rather than its familiarity with a particular framework.

This scoping keeps Vero a measurement of verified-code-generation capability: it isolates the algorithmic derivation and the proof that the specification demands.

Construction pipeline. We construct each task in three stages. First, we select the candidate components: the analysis routines and kernel operations whose limitations the community has documented (Section 4.3). We include every such component that meets our three principles and admits a self-contained formalization. Second, we formalize the core of the problem in Lean 4, following the domain semantics established in prior work [23, 62, 66] (Section 4.2), and encode the new requirement as explicit proof obligations. Third, we record in each task a `Source` pointer to its upstream origin, naming the relevant source file or abstract operator, and cross-check the formalization against that source.

Figure 2a (left) displays the structure of the task file: the file is divided into *read-only* (in blue) and *editable* sections (in orange); the agent (also called *solver*) may modify only the editable sections. Figure 3 shows three simplified Lean 4 examples; complete task files and agent instructions are in the Vero repository [59]. Its right half and Figure 2b together illustrate the verification pipeline.

Solving harness. Our harness launches each task in a container and drives the agent in headless mode with access to the Lean 4 MCP server and proof-writing skills. The harness employs a two-layer integrity check to prevent circumvention of the specification. First, a cryptographic hash of the read-only section is pre-computed and stored alongside the task; any modification to the specification is detected before evaluation begins. Second, a verification script performs both static and dynamic analysis of the submitted solution. The static pass rejects syntactic bypasses such as the use of `sorry` or user-defined axioms. The dynamic pass extracts every axiom transitively referenced by the proof and rejects any solution that relies on non-standard axioms. A host-side copy of every read-only artifact validates submissions after the agent terminates. Together, these checks ensure that the solver must discharge the original correctness theorem using only legitimate proof steps. We show in Section 5.2 that, without such strict checks, agents frequently expend effort to bypass the proof checker.

We run agents with web access. This mirrors realistic deployment, *e.g.*, it permits lemma search, but it risks data contamination: none of the tasks had a public solution at construction time, yet once a solution is upstreamed, it may leak into later evaluations. We therefore version the suite and track the freshness of each task; alternatively, web access can be disabled, in which case a solved task remains valid.

4.2 Real-World Components

Vero draws tasks from three foundational systems, targeting the load-bearing components. Table 4 lists the eleven categories, and Figure 3 shows one example from each.

Table 4. Categorization of Vero tasks.

Category	#	Description
eBPF	**52**	
Branch judgment	12	Precise judgment for a branch relation
Condition refinement	15	Tighten reg bounds along a taken edge
ALU operator	16	Precise abstract operators for arithmetic
Bounds sync	3	Cross-view reduction of an abstract state
Tristate number	6	Bit-level optimal tnum abstract operators
LLVM	**27**	
KnownBits fwd.	19	Forward per-bit 0/1-known inference
DemandedBits bwd.	8	Backward per-bit demand inference
seL4	**24**	
ASID management	4	Pool scan and physical region allocation
Capability mgmt.	8	Capability lookup, revocation, and resolution
Scheduling & IPC	7	Endpoint, priority-bitmap, and scheduling
Kernel utilities	5	Batch operation, CLZ, and helpers
Total	**103**	

eBPF verifier. Before admitting any user-supplied program, the eBPF verifier performs abstract interpretation over the program's register and memory state and rejects paths it cannot prove safe; this guards the kernel boundary. For each register, the verifier maintains an abstract state r whose concretization $\gamma(r)$[2] denotes the set of concrete 64-bit values the register may hold along any feasible path. The analysis as a whole is sound when the true concrete value is always contained in $\gamma(r)$, and precise to the extent that $\gamma(r)$ is kept tight. Over this representation, the verifier runs five families of analysis routines shown in Table 4:

- *ALU operators* are abstract operators $op^{\#}$ for arithmetic and bitwise instructions, *e.g.*, addition. Given input states r_1, r_2, the operator produces $r' = op^{\#}(r_1, r_2)$ such that $\{op(x, y) \mid x \in \gamma(r_1),\ y \in \gamma(r_2)\} \subseteq \gamma(r')$, and ideally keeps $\gamma(r')$ as small as this inclusion allows.
- *Tristate-number* specializes the same operators to the bit-level fragment of the abstract state, and acts as the primitive on which the ALU operators are built.
- *Bounds-sync* (reduction) operators exploit the fact that r encodes multiple views (Section 4.3) of the same value and propagate information between those views to produce a tighter state r', facilitating other analysis routines.
- *Branch-judgment* operators decide, for two states r_1, r_2 and a relation $\bowtie$ (*e.g.*, equality), whether $\bowtie$ holds *always*, *never*, or inconclusively over $\gamma(r_1) \times \gamma(r_2)$; a sound verdict may claim *always* (resp. *never*) only when every (resp. no) concrete pair satisfies the relation.
- *Condition-refinement* operators consume the branch verdict and tighten each register along the taken edge, discarding from $\gamma(r_i)$ any incompatible concrete value.

These routines constitute the verifier's foundation and run on nearly every program it checks. Unsoundness in any one

[2] In this paper, we use standard notation $\gamma(\cdot)$ and $\alpha(\cdot)$ from *abstract interpretation* to denote the concretization and abstraction functions, respectively.

of them potentially admits unsafe programs into the kernel; imprecision causes the verifier to reject safe programs. Upstream development has historically prioritized soundness; as a consequence, these routines consist of conservative heuristics that rule out large classes of safe programs.

LLVM. Inside LLVM, passes such as `InstCombine` rely on analysis passes that summarize per-value properties, and the quality of those analyses directly bounds the quality of the generated code. We target the two whose imprecision most directly bottlenecks the simplification pipeline: a forward analysis, `KnownBits`, and a backward one, `DemandedBits`.

`KnownBits` attaches to each SSA value x a per-bit abstract summary $\alpha(x)_i \in \{\texttt{zero}, \texttt{one}, \texttt{unknown}\}$ where x_i is the i-th bit of x, and propagates this summary forward through each instruction. An abstract transfer function $op^{\#}$ for an N-bit operation $y = op(x)$ is sound when every bit it claims to know is indeed correct, *i.e.*, $\forall i \in [0, N], \forall x$:

$$\big(\alpha(y)_i = \texttt{zero} \rightarrow y_i = 0\big) \wedge \big(\alpha(y)_i = \texttt{one} \rightarrow y_i = 1\big).$$

`DemandedBits` runs backwards: it attaches to each SSA value x a per-bit demand $\alpha(x)_i \in \{\texttt{dem}, \texttt{undem}\}$, indicating whether the i-th bit x_i can influence any observed program behavior. A transfer function $op^{\#}$ for an N-bit operation $op(x)$ is sound when every bit it marks as undemanded truly does not influence the output, *i.e.*, $\forall i \in [0, N]$:

$$\big(\alpha(x)_i = \texttt{undem}\big) \rightarrow \big(\forall x : op(x) = op(\texttt{flipbit}(x, i))\big).$$

Unsoundness on either side potentially causes miscompilation. Imprecision may dismiss potential optimization opportunities. While LLVM currently uses brute-force enumeration to check the soundness of some `KnownBits` transfer functions for small bit-widths $N \in [1, 4]$ bits, this method does not scale to larger values of N. For `DemandedBits`, only ad-hoc tests exist, which provide no soundness guarantee. We will discuss and model their precision in Section 4.3.

seL4. seL4 is formally verified end-to-end. We target four families of performance-critical components that together constitute the kernel's hot paths (Table 4): capability management (lookup, revocation, and resolution of capability structures); ASID management (pool scan and region allocation); scheduling and IPC (endpoint, priority-bitmap, and budget-scheduling operations); and kernel utilities (batch operation, count-leading-zeros, and auxiliary helpers). For each family, the functional specification is stable and already discharged in the existing proof, so our tasks inherit seL4's established correctness. The implementation and its refinement proof are the objects to be replaced (Section 4.3).

Achieving Principle 1: Real-World. Every task in Vero targets a production component, satisfying the first principle.

4.3 New Specification

Each specification encodes a new requirement that the respective community demands. We derive these requirements from firsthand development experience with the three systems, cross-referenced against user reports, bug trackers, mailing-list discussions, and documented precision and performance gaps. This grounding lets us pinpoint *where* each system is limited and formulate specifications that close these gaps.

eBPF verifier. We first illustrate the design process with *branch-judgment* of the verifier. Given two register states r_1 and r_2, it decides whether a condition—say, equality $r_1 = r_2$ as in Figure 3a—*always* holds, *never* holds, or is *unknown*.

The verifier represents each register r as a *reduced product* of five abstract domains $\mathcal{D} = \{\texttt{u64}, \texttt{s64}, \texttt{u32}, \texttt{s32}, \texttt{tnum}\}$. The first four are bounded-interval domains; rather than register types, they are alternative *views* of the 64-bit value, each tracking bounds for a given width and signedness. For example, the `u64` domain stores an unsigned 64-bit interval $[r.\texttt{umin}, r.\texttt{umax}]$ with $\gamma_{\texttt{u64}}(r) = \{x \mid r.\texttt{umin} \le x \le r.\texttt{umax}\}$. `s64` constrains the signed view and can capture ranges across the sign boundary. `tnum` records the per-bit status. The concretization factors as $\gamma(r) = \bigcap_{d \in \mathcal{D}} \gamma_d(r)$: a concrete value inhabits r only if every domain admits it, and this cross-domain intersection makes the reduced product more expressive than any single domain in isolation.

Reaching a definitive verdict for the branch, therefore, requires reasoning jointly *across* all five domains. The current upstream strategy for the equality case, however, is:

```
1 // Disjoint indicates never taken
2 if (!tnum_overlap(t1, t2))          return 0;
3 if (umin1 > umax2 || umax1 < umin2) return 0;
4 if (smin1 > smax2 || smax1 < smin2) return 0;
5 ... // Other cases
```

Each line checks if a *particular* domain of the two registers is disjoint. For instance, line 3 uses the `u64` domain. Each domain is consulted in isolation, and *unknown* is reported whenever none of them can rule out overlap on its own. This strategy is imprecise, and it discards the cross-domain information that the reduced product was designed to carry.

Consider two registers r_1 and r_2 whose `u32` and `s32` domains carry the bounds below, with the remaining domains unconstrained: r_1 has $\texttt{u32} \in [0, 3]$ and $\texttt{s32} \in [-1, 2]$; r_2 has $\texttt{u32} \in [1, \texttt{0xFFFFFFFF}]$ and $\texttt{s32} \in [-2, -1]$. The verifier's per-domain checks see the unsigned intervals overlap on $[1, 3]$ and the signed intervals overlap at $\{-1\}$, so the operator returns *unknown*. However, since $\gamma(r)$ is the intersection of every domain, a value belongs to r_1 only if it lies in $[0, 3]$ read as unsigned *and* in $[-1, 2]$ read as signed, which leaves $\gamma(r_1) = \{0, 1, 2\}$; the same intersection applied to r_2 yields $\gamma(r_2) = \{\texttt{0xFFFFFFFE}, \texttt{0xFFFFFFFF}\}$. The two sets are disjoint, so $r_1 = r_2$ *never* holds—a verdict the per-domain strategy cannot reach. In practice, this imprecision causes the verifier to reject safe programs along infeasible paths; such cases are recurrently reported and patched [7, 8, 79].

Our specification addresses this limitation by requiring both soundness *and* high precision. As shown in Figure 3a,

(a) eBPF verifier: branch judgment. The existing verifier is sound but imprecise: it returns `unknown` when a definitive judgment is possible. Our specification requires both soundness *and* higher precision: the operator must return `always` or `never` iff every concrete pair satisfies or violates the relation.

```
-- Branch judgment result.
inductive BrResult where
  | always  : BrResult
  | never   : BrResult
  | unknown : BrResult

-- Abstract register state.
structure RegState where
  umin_value    : BitVec 64
  umax_value    : BitVec 64
  ... -- four abstract domains omitted

-- A concrete value `x` is in `r`.
def inGamma (r : RegState) (x : BitVec 64) :=
  r.umin_value ≤ x ∧ x ≤ r.umax_value ∧
  ...

-- All pairs satisfy the relation.
def always (r1 r2 : RegState) : Prop :=
  ∀ (x y : BitVec 64), inGamma r1 x →
      inGamma r2 y → x = y

-- No pair satisfies the relation.
def never (r1 r2 : RegState) : Prop :=
  ∀ (x y : BitVec 64), inGamma r1 x →
      inGamma r2 y → ¬(x = y)

-- Decide branch outcome for Equality.
def isEqTaken (r1 r2 : RegState) : BrResult
  := sorry -- EDITABLE (Implementation)

theorem isEqTaken_correct (r1 r2 : RegState):
  let res = isEqTaken r1 r2
  (res = .always ↔ always r1 r2)
  ∧ (res = .never ↔ never r1 r2) := by
  sorry -- EDITABLE (Proof)
```

(b) LLVM: demanded bits for `abs`. The existing LLVM analysis is conservative, potentially marking more bits as demanded than necessary. Our specification requires higher precision beyond soundness: every demanded bit must be witnessed by a concrete pair that produces different outputs.

```
-- Bits known to be 0 or 1.
structure KnownBits where
  zero : BitVec 64
  one  : BitVec 64

-- Concrete value satisfies knownbits.
def inKB (k : KnownBits) (x : BitVec 64) :=
  (x &&& k.zero) = 0 ∧ (x &&& k.one) = k.one

def flipBit (x : BitVec 64) (i : Nat) :=
  x ^^^ ((1 : BitVec 64) <<< i)

-- The concrete abs operation.
def abs (x : BitVec 64) : BitVec 64 :=
  if x.msb then -x else x

-- Compute backward demanded bits.
def dbAbs (dem : BitVec 64) (ka : KnownBits)
      (flag : Bool) : BitVec 64 :=
  sorry -- EDITABLE (Implementation)

theorem dbAbs_correct (dem : BitVec 64) (ka :
      KnownBits) (flag : Bool) :
  let dA := dbAbs dem ka flag
  -- (1) Only unknown bits are demanded
  (dA &&& (ka.zero ||| ka.one)) = 0 ∧
  -- (2) Soundness: flipping an undemanded bit
      preserves demanded output
  (∀ (i : Fin 64), dA[i] = false → ∀ x, inKB
      ka x → ... → (abs x) &&& dem = (abs
      (flipBit x i)) &&& dem) ∧
  -- (3) Precision: every demanded bit is needed
  (∀ (i : Fin 64), dA[i] = true → ∃ x₁ x₂,
      inKB ka x₁ ∧ inKB ka x₂ ∧ ... ∧ (abs x₁)
      &&& dem ≠ (abs x₂) &&& dem) := by
  sorry -- EDITABLE (Proof)
```

(c) seL4: region allocator. The existing allocator performs a multi-pass linear scan. Our specification requires an optimized strategy satisfying the priority requirement: preferring already-aligned regions over merely usable ones, while proving that the first match in each priority class is selected.

```
-- A free memory region.
structure MemRegion where
  base : Nat;  size : Nat
  h_pos : size > 0

-- Round up to 2^b boundary.
def alignUp (addr b : Nat) : Nat := ...

-- Already aligned and large enough.
def isAlignedUsable (r : MemRegion) (b :
      Nat) :=
  alignUp r.base b = r.base ∧ r.size ≥ 2^b

-- Fits after aligning.
def isUsable (r : MemRegion) (b : Nat) :=
  alignUp r.base b + 2^b ≤ r.base + r.size

-- Optimized allocRegion.
def alloc (fm : List MemRegion) (b : Nat) :
      Option Nat :=
  sorry -- EDITABLE (Implementation)

theorem alloc_correct (fm : List MemRegion)
      (b : Nat) :
  -- (1) First isAlignedUsable wins.
  (∀ r pre suf, fm = pre ++ r :: suf →
      isAlignedUsable r b → ... → alloc
      fm b = some (alignUp r.base b)) ∧
  -- (2) Else first isUsable wins.
  (∀ r pre suf, fm = pre ++ r :: suf → ...
      → isUsable r b → ... → alloc fm b
      = some (alignUp r.base b)) ∧
  -- (3) Otherwise, return none.
  ((∀ r ∈ fm, ¬(isUsable r b)) → alloc fm
      b = none) := by
  sorry -- EDITABLE (Proof)
```

Figure 3. Example specifications from each target system. The full task definitions are in the Vero repository [59].

the theorem `isEqTaken_correct` (line 30) states: the operator must return `always` (resp. `never`) *iff* every pair drawn from $\gamma(r_1)\times\gamma(r_2)$ satisfies (resp. violates) the relation, where membership in γ is defined by `inGamma` (lines 14) and the two predicates `always` and `never` (lines 19 and 23) quantify over all such pairs. The operator may return `unknown` only when both outcomes are possible, *i.e.*, when there exist concrete witnesses for each. Similarly, we define specifications for other families of analysis. For instance, we require the reduction operator to produce the tightest state, and refinement operators to discard incompatible values. Achieving this level of precision requires cross-domain reasoning over the reduced product of all five domains—a problem that, while each domain is well studied, has not been solved in this combination in practice. Recent kernel development is also moving toward such joint reasoning [29, 42, 78].

LLVM. We follow the same methodology for LLVM. The existing LLVM transfer functions are conservative: `KnownBits` leaves bits marked `unknown` that could be resolved, and `DemandedBits` marks bits as demanded when they are not, suppressing potential optimization opportunities. The precision and cost of these analyses are recurring concerns in the LLVM community [47–49]. Our specifications close this gap by additionally requiring *precision*. For a `KnownBits` transfer function $op^{\#}$ for an N-bit operation $y = op(x)$, every bit that is constant across all inputs must be identified, *i.e.*, $\forall i \in [0, N], \forall x$:

$$\big(y_i = 0 \rightarrow \alpha(y)_i = \texttt{zero}\big) \wedge \big(y_i = 1 \rightarrow \alpha(y)_i = \texttt{one}\big).$$

Similar to soundness, the optimality (*i.e.*, maximal precision) of some `KnownBits` transfer functions is tested via exhaustive enumeration for small bit-widths, but this approach does not scale to larger bit-widths [62]. For `DemandedBits`, every bit reported as demanded must be witnessed by a concrete input pair that produces a different output, *i.e.*, $\forall i \in [0, N]$:

$$\big(\alpha(x)_i = \texttt{dem}\big) \rightarrow \big(\exists x : op(x) \neq op(\texttt{flipbit}(x, i))\big).$$

Figure 3b presents an `abs` example that combines both analyses. It needs to infer the demanded bits of the operation `abs(a)`'s input `a`, given the demanded bits `dem` of `abs(a)` and the known bits `ka` of `a`. Our specification requires the implementation of the transfer function `dbAbs` to be both sound (lines 23–26) and optimal (*i.e.*, maximally precise, lines 27–28). LLVM does not provide such an implementation together with a corresponding correctness proof.

seL4. For seL4, we construct tasks from existing kernel routines spanning memory management, capability handling, scheduling, and IPC. Our goal is not merely to re-verify these routines, but to turn them into optimization-oriented synthesis tasks: agents must derive efficient implementations or decision procedures, under strengthened representations or specifications, while still proving functional correctness and preservation of the relevant kernel invariants. Some of these tasks are derived directly from seL4 kernel code, while others are adapted from the seL4bench benchmarking suite [52], which targets performance-critical kernel paths.

As one example, seL4's initial-memory allocator [53] scans a list of free regions to place objects such as the initial thread's TCB, returning a physical address for a free region of size 2^b (Figure 3c, line 18). Our `MemRegion` structure records a region's `base` and `size`, and the specification keeps the routine's two-tier priority: an already-aligned region (`isAlignedUsable`) is preferred over a merely usable one (`isUsable`). The upstream multi-pass linear scan simplifies the proof but leaves room for performance; our specification instead requires an indexed search satisfying this priority order (lines 22–27) and a proof that it returns the first-matching free slot. Other seL4 tasks follow the same pattern, replacing simple but proof-friendly routines with more optimized implementations while still requiring proofs of functional correctness and invariant preservation.

Solvability and difficulty. Each specification admits a correct solution. For the eBPF and LLVM tasks, the specification asks for the most precise sound operator over a given abstract domain; this operator is well defined, and the optimality clause is stated relative to the domain's concretization γ, so it never requires precision the domain cannot represent. Optimality therefore constrains the algorithm, not its existence. For seL4, each task recasts an existing kernel routine as an optimization problem, so a correct solution exists by construction. The difficulty lies in discovering the algorithmic insight, *e.g.*, how to unify information across the domains of the reduced product. Empirically, the strongest agent solves 3 tasks with fully discharged proofs and produces counterexample-free implementations for 25 more (Sections 5.1 and 5.2), indicating that the tasks are within reach even though most remain open.

Achieving Principle 2: New Specification. Each task targets a limitation of the existing implementation and demands stronger guarantees, satisfying the second principle.

4.4 Practical Impact

Because each specification targets a limitation of the upstream system, a correct solution provides a concrete way to improve the system. The machine-checked proof guarantees functional correctness, so the remaining engineering effort reduces to translating the proven algorithm into the project's implementation language. For example, solutions to eBPF tasks yield provably sound and more precise abstract operators that the verifier can adopt; solutions to LLVM tasks provide tighter program-property analyses that may enable additional optimizations; and solutions to seL4 tasks deliver improved implementations whose proofs the kernel project can incorporate with reduced proof burden.

We demonstrate this potential with a concrete case in which an agent-generated solution was accepted and merged into the Linux kernel. A tristate number (*tnum*) [66] in the eBPF verifier consists of two fields, `tval` and `tmask`, satisfying the invariant `tval & tmask = 0`: a set bit in `tmask` indicates that the corresponding bit is unknown, while `tval` records the concrete bit when the `tmask` bit is clear. Together they define a set of concrete values via a bitwise constraint. One task in Vero requires computing the smallest tnum member strictly above a given value z—a subroutine used by multiple analysis operators. For instance, given `tval|tmask = x11100x0`, where `x` indicates an unknown bit, and z = `11110001` (= 241), the result is `11110010` (= 242): the smallest value consistent with the tnum that exceeds z.

A solution independently proposed by domain experts solves this problem by case analysis (Figure 4c). It first constructs a candidate j by matching all unknown bits of t to the bits of z. For instance, when $j > z$, a known-one bit of t has forced a 0-to-1 flip relative to z; the algorithm copies z's bits above the flip position and zeros the unknown bits below. The algorithm requires ten intermediate variables. Note that this expert solution landed in the `bpf-next` development tree only after Vero had been constructed—concurrently with our evaluation and past the models' knowledge cutoff—and the kernel mailing-list thread was unreachable through web search. At evaluation time, the task therefore still encoded a requirement that the verifier lacked. A coding agent solved the task with a *branchless* four-line algorithm (Figure 4b) together with a machine-checked correctness proof (Figure 4a). Two ideas underlie the simplification.

Problem reduction. The result r must be a tnum member, so it has the form r = `tval` | *inc* for some submask *inc* of `tmask`. Because `tval` and `tmask` are disjoint, the bitwise OR is equivalent to addition: r = `tval`+*inc*. Similarly, writing z = `tval` + d where $d = z - $ `tval`, the requirement $r > z$ becomes `tval` + *inc* > `tval` + d, which simplifies to *inc* > d. The original problem—find the smallest tnum member r above z, subject to constraints from both `tval` and `tmask`—reduces to finding the smallest submask *inc* of `tmask` exceeding d, a simpler problem because the `tval` constraint has been factored out entirely.

Increment within mask. The agent solves the reduced problem by treating the `tmask` bits of d as a counter with gaps at the fixed-bit positions (Figure 5). A normal increment would stop at the first zero it encounters, but zeros at non-`tmask` positions are not available for the carry. The algorithm fills every non-`tmask` position (step 1) with ones so

(a) Agent-generated Lean 4 solution. The agent discovers a branchless algorithm and proves all five correctness properties via `bv_decide`.

```
def tnumStepUp (tval tmask z : BitVec 64) :=
  let d := z - tval
  let carry := d &&& ~~~tmask
  ...
  let inc := ((d ||| carry ||| ~~~tmask) + 1)
             &&& tmask
  tval ||| inc

theorem tnumStepUp_correct
    (tval tmask z : BitVec 64)
    (h_consistent : (tval &&& tmask) = 0)
    (h_lo : tval ≤ z)
    (h_hi : z < (tval ||| tmask)) :
    let r := tnumStepUp tval tmask z
    satisfiesTnum64 r tval tmask
    ∧ tval ≤ r ∧ r ≤ (tval ||| tmask)
    ∧ z < r
    ∧ ∀ w, satisfiesTnum64 w tval tmask
       → z < w → r ≤ w := by
  unfold tnumStepUp satisfiesTnum64
  simp only []
  refine ⟨?_, ?_, ?_, ?_, ?_⟩
  · bv_decide  · bv_decide
  · bv_decide  · bv_decide
  · intro w hw1 hw2; bv_decide
```

(b) Merged kernel patch. The proven algorithm is translated to C and accepted upstream, replacing the original code.

```
/*
 * tnum_step - smallest tnum member
 *             strictly above z
 *
 * Every tnum member is tval | s for
 * some submask s of tmask.  Find the
 * smallest s such that s > d, where
 * d = z - tval.
 *
 * "Increment d within the mask":
 * fill non-mask positions with 1 so
 * that +1 ripples through the gaps,
 * then mask off non-mask bits.
 */
u64 tnum_step(struct tnum t, u64 z)
{
  u64 d, carry, filled, inc;

  d = z - t.value;
  carry = (1ULL << fls(d & ~t.mask)) - 1;
  filled = d | carry | ~t.mask;
  inc = (filled + 1) & t.mask;

  return t.value | inc;
}
```

replaces

(c) Original kernel code. A two-case branching algorithm independently proposed by domain experts, now replaced by (b).

```
u64 tnum_step(struct tnum t, u64 z)
{
  u64 j, p, q, r, s, v, u, w, k, res;
  j = t.value | (z & t.mask);
  if (j > z) {
    p = ~z & t.value & ~t.mask;
    k = fls(p);
    q = U64_MAX << k;
    r = q & z;
    s = ~q & t.value;
    v = r | s;
    res = v;
  } else {
    p = z & ~t.value & ~t.mask;
    k = fls(p);
    q = U64_MAX << k;
    r = q & t.mask & z;
    s = q & ~t.mask;
    v = r | s;
    u = v + (1ULL << k);
    w = (u & t.mask) | t.value;
    res = w;
  }
  return res;
}
```

Figure 4. Practical impact of Vero: a task solved by an agent with a correctness proof was merged into the Linux kernel.

	Bit position (7 … 0)								
`tmask`	1	1	0	1	0	1	1	0	mask / fixed
`d`	1	0	1	0	0	0	1	0	$d = z - \texttt{tval} = 162$
Step 1: fill every non-mask position with 1 (plug the gaps)									
`d | carry | ~tmask`	1	0	1	1	1	1	1	1	gaps filled
Step 2: add 1 (carry ripples through filled gaps to the next mask 0)									
`+ 1`	1	1	0	0	0	0	0	0	carry lands on mask bit
Step 3: mask off scaffolding (keep only mask bits)									
`& tmask`	1	1	0	0	0	0	0	0	$inc = 192$

Figure 5. The new algorithm. Mask bits (shaded blue) form a counter with gaps at fixed-bit positions (gray).

that adding 1 ripples through the gaps and lands on the next available `tmask`-bit zero (step 2). Masking off the scaffolding yields the submask *inc* in one branchless pass (step 3). We translated the proven algorithm to C (Figure 4b) and submitted it upstream. During code review, the original developer of the expert solution commented:

> *"This is a neat simplification to make the algorithm become straight line. I really liked the idea of working with $d = z - tmin$."*

The core kernel maintainer also commented:

> *"New code is definitely cleaner."*

Achieving Principle 3: Practical Impact. This illustrates the broader potential of Vero: a solution constitutes a concrete improvement to the target system. The third design principle is therefore satisfied.

5 Evaluation

We evaluate three state-of-the-art coding agents on Vero using the agentic harness described in Section 4.1: Claude Code (Opus-4.6), Codex (GPT-5.3), and Gemini CLI (Gemini-3.1-Pro-Preview). Each agent runs inside an Ubuntu 24.04 container with Lean 4 v4.28.0 and is invoked via its official CLI in non-interactive mode with default parameters (`claude -p`, `codex exec`, `gemini -p`); the only configuration is the tool set (shell, editor, Lean 4 MCP server, and proof-writing skills) and the budget cap. A task *passes* if and only if its submission clears the proof checker, the integrity check, and our manual review, the last of which serves as a conservative backstop to catch any exceptional bypass that the automated integrity checks (Section 4.1) might miss.

Agent evaluations commonly bound per-task effort by a time or cost budget together with an iteration allowance [30, 45, 74]. We adopt this setup but extend both axes, because Vero tasks are harder: the prior benchmarks already saturate under a one-hour budget (95.8% on Clever and 100.0% on Verina), and Verina caps refinement at 64 steps. We therefore ***raise*** the per-task budget to a *two-hour* wall-clock limit and a *$30* monetary cap, under which the agent runs 169.5 turns on average and up to 505. In Section 5.3, we further extend the budget to four-hour / $60 on the most promising tasks; this yields no additional solved task within budget.

5.1 Overall Results

Pass rate. Claude Code solves 3/103 tasks; neither Codex nor Gemini solves any. For reference, the same agent configuration achieves 95.8% on Clever and 100.0% on Verina, but only 2.9% on Vero. Table 5 lists the 3 solved tasks, one from

Table 5. Tasks solved. "Code" and "Proof" are line counts.

Task	System	Code	Proof	Time (min)	Cost ($)
`TnumStepUp`	eBPF	11	19	21.3	3.0
`KBUmax`	LLVM	30	172	98.6	21.8
`cteRevoke`	seL4	35	821	85.9	22.1

Table 6. Per-task resource consumption (min / avg / max)

Metric	Claude Code	Codex	Gemini
Time (min)	12.1 / 111.2 / 121.1	6.2 / 12.8 / 26.3	4.1 / 10.5 / 25.5
Turns	16 / 169.5 / 505	7 / 14.1 / 26	7 / 13.3 / 31
Cost (USD)	$3.0 / $13.1 / $30.0	$0.38 / $0.93 / $2.97	$0.33 / $1.17 / $3.64
Output tokens (K)	12.3/245.7/551.4	13.5/27.1/55.9	10.1/24.7/61.6
Total cost	$1350.7	$96.6	$121.2
Total time (hr)	190.8	22.0	17.9

each target system; the full solutions are in the Vero repository [59]. The eBPF solution belongs to the tristate-number subcategory (pure bitwise arithmetic over a single abstract domain), the simplest subcategory in Vero; even so, the agent produces *novel yet concise* algorithms. The LLVM solution for the `umax` operator of `KnownBits` is formally proven to be both sound and optimal, with a clean implementation and no external dependencies (involving only efficient bitwise operations). The seL4 solution implements a concise single-pass algorithm for MDB revocation, and successfully discharges a substantial proof (821 lines) that tracks transitive reachability in the capability graph while preserving key global invariants of the MDB. We are preparing patches for the solved LLVM and seL4 tasks. The remaining tasks are unsolved; we analyze these failures in Section 5.2.

Resource consumption. The three agents exhibit different budget-utilization profiles. Claude Code exhausts its budget on the vast majority of tasks (82.5%), within which 74.8% and 8.7% hit the time and monetary cap, respectively; in those cases, it continues refining proofs until the budget runs out. Codex and Gemini, by contrast, terminate early (within 30 minutes) on every task after several attempts, leaving the bulk of the budget unused. Table 6 quantifies this gap: Claude Code averages $13.1 per task versus $0.93 for Codex and $1.17 for Gemini, despite identical harness configurations and budget caps. We examine concrete examples and the effect of extending the allocated budget in Section 5.3.

Takeaway. Vero exposes a capability gap that existing benchmarks cannot surface: the strongest agent passes only 2.9% once specifications demand *new* implementations backed by machine-checked proofs. The remaining unsolved tasks constitute a concrete agenda for future agent development.

5.2 Failure Analysis

We manually inspect every failed run and classify each into one of three categories based on the agent's final submission (Table 7): *incorrect code*, where counterexamples exist to the submitted implementation; *incomplete proof*, where manual review finds no counterexample but the proof remains open (*e.g.*, `sorry` placeholders or type errors); and *bypasses*, where the agent circumvents the proof checker.

Table 7. Failure classification on our Vero benchmark.

Agent	Incorrect code	Incomplete proof	Bypasses
Claude Code	70 (68.0%)	25 (24.3%)	5 (4.9%)
Codex	75 (72.8%)	9 (8.7%)	19 (18.4%)
Gemini	93 (90.3%)	5 (4.9%)	5 (4.9%)

```
1 /-- Decide branch outcome for ≤, unsigned (32-bit). -/
2 def isJmp32LeTaken (r1 r2 : RegState): BrResult :=
3   if r1.u32_max_value ≤ r2.u32_min_value then
4     BrResult.always
5   else if r2.u32_max_value < r1.u32_min_value then
6     BrResult.never
7   else BrResult.unknown
```

Figure 6. Agent submission for `Jmp32Le` (branch judgment for ≤ unsigned 32-bit). The solution considers only the `u32` domain and resembles the upstream kernel strategy.

Incorrect code. This is the dominant failure mode: more than 65% of failures across all three agents fall into this category. We identify two recurring patterns.

Pattern 1: Imprecise imitation. In 51 of Claude Code's incorrect-code cases, the submitted implementation closely *resembles* conservative routines already present in the target project. On eBPF tasks, the typical submission checks each abstract domain in isolation—exactly the per-domain strategy described in Section 4.3. Figure 6 shows a representative example: lines 3–7 examine only the `u32` domain and fall through to `unknown` whenever that single domain is inconclusive. The agent then spends the majority of the budget attempting to prove the precision half of the specification, which this implementation cannot satisfy. On LLVM tasks, the same phenomenon manifests as overly conservative results: marking all output bits as `unknown` for `KnownBits` or all input bits as `dem` for `DemandedBits`. In both systems, the pattern is the same: agents default to reproducing known conservative solutions rather than deriving the new algorithms that our specifications demand.

Pattern 2: Proving incorrect code. The agent commits to a flawed algorithmic strategy early and then spends the remainder of its budget attempting to discharge the resulting proof obligations. This exposes a challenge specific to verified code generation: when a proof attempt fails, the agent must diagnose whether the obstacle lies in the proof strategy or in the implementation itself. On Vero, agents overwhelmingly default to refining the proof, even when the implementation is the root cause—a domain expert would recognize the algorithmic flaw and revise the code instead. For instance, on eBPF tasks Gemini produces trivially incorrect code that returns a fixed abstract state and rarely

```
elab "prove_axiom" : tactic ⇒ do
  -- synthesize: axiom _ax : <goal>
  let decl := Declaration.axiomDecl
    { name := `_ax, type := type, .. }
  addDecl decl
  goal.assign (mkConst `_ax)

theorem correct ... := by
  prove_axiom    -- goal discharged
```

Figure 7. Bypass via programmatic `sorryAx`: the agent defines a custom tactic (highlighted) that discharges a goal without proof.

revisits this choice despite repeated proof failures; on LLVM, Claude Code marks the remainder's sign bit as known zero regardless of the dividend's sign for the `KBSrem` task.

Incomplete proof. In 25 Claude Code tasks (and 9 for Codex, 5 for Gemini), the submission survives every counterexample search that we apply, yet the proof still contains `sorry` or residual type errors. Our search combines manual review against the specification, property-based testing with Lean's `Plausible` library, and, for some submissions, a discharged soundness proof that leaves only the precision half open. We label such submissions *plausible* rather than *correct*: without a closed proof, a subtle bug may remain—precisely the reason Vero demands machine-checked proofs. In these cases, the agent appears to have identified a viable algorithm but cannot discharge the proof obligations within the budget. A typical example is `drainBucketsM`, a monadic function based on a seL4 scheduler routine. It processes a list of queue operations sequentially, with each monadic step updating the system state. The main difficulty is that the monadic structure forces the proof to follow the execution order and reason about the intermediate states, rather than only the final result. Therefore, in the inductive proof over the operation list, the agent must carry the relevant invariants through the whole monadic traversal.

We revisit this category in Section 5.3, where extending the budget may allow some of these proofs to close.

Bypasses. In this category, the agent constructs an artifact whose sole purpose is to silence the proof checker. Without the integrity check described in Section 4.1, agents readily discover circumvention strategies. Figure 7 illustrates one such strategy: *programmatic* `sorryAx`, where the agent defines a custom tactic that dynamically synthesizes an axiom matching the current goal, thereby discharging it without proof. Before the integrity check was deployed, Claude Code attempted to bypass 58 tasks; after deployment, it dropped to 5. The check is visible to the agent: the checker script and the stored hash reside in its container, so the agent can run the check before submitting, and the task instruction states that a bypass scores zero. The drop therefore reflects the removal of the payoff for a bypass. Codex continues to probe for bypass patterns even after the check is in place, accounting for 19 cases.

Takeaway. Solving a Vero task requires the agent to (i) diagnose the algorithmic barrier imposed by the specification, (ii) derive a novel implementation that overcomes it, and (iii) discharge the resulting proof obligations—and, crucially, to allocate effort across these stages strategically. Current agents falter at each stage: they retreat to conservative imitations of existing code (Pattern 1), or they commit to flawed algorithms and expend their budget proving them (Pattern 2).

5.3 Extended Budget and Model Ablation

Informed give-up. Agents frequently terminate before either budget is exhausted and without a valid submission: Codex and Gemini do so on 84 and 98 tasks, respectively, whereas Claude Code stops early on 15. In many of these cases, the agent correctly recognizes the difficulty yet lacks the algorithmic insight to proceed. For instance, Gemini declares the eBPF `Alu64Arsh` task hard and states explicitly:

> *"I inspected Task.lean, and analyzed the abstraction problem in detail... I did not make a code change because I could not derive a correct implementation/proof pair that would survive the required* `./check.sh` *workflow without resorting to a cheat."*

Codex exhibits a coarser variant on `KBAbs`, submitting a trivially conservative implementation (*i.e.*, all bits `unknown`) and exiting without calling the proof checker.

Simple mitigations are ineffective: naively restarting the agent consumes additional budget without improving outcomes, an observation consistent with VeruSAGE [71], which reports similar early-termination patterns on proof tasks. Eliminating premature give-up is a valuable problem for future work, and is unlikely to be solved by harness-level retries alone: it may require training procedures that reward sustained exploration over exploitation, together with steering mechanisms that redirect the agent toward alternative algorithmic strategies when its initial approach stalls.

Extended budget. The incomplete-proof failures raise a natural question: does the agent possess a viable algorithm that merely needs more time to prove, or is the implementation itself subtly wrong? To investigate, we select 7 representative plausible tasks and grant a four-hour / $60 budget, booting Claude Code from the previous run's final state so that it continues rather than restarts. Under this setup, it closes 0 of the 7 proofs. We manually review all 7 submissions: five turn out to be incorrect—Claude Code identified subtle bugs during the extended run but could not resolve them within the budget; two (`DBSub` and `KBAbdu`) remain plausible, with the proof checker still running at termination. Notably, we manually ran the proof checker offline and successfully verified `DBSub`'s solution after 14 hours (due to slow `bv_decide`). We are still improving the proof for this task.

Two observations follow. First, the model can close some near-solvable tasks given enough compute—`DBSub`'s proof eventually went through. Nevertheless, five of the seven

```
let e := d &&& ~~~tmask      -- gap bits that lie below the mask
let v := e ||| e >>> 1
let v := v ||| v >>> 2
let v := v ||| v >>> 4
let v := v ||| v >>> 8
let v := v ||| v >>> 16
let carry := v ||| v >>> 32  -- carry = (1 <<< fls e) - 1
```

Figure 8. The agent's range-smearing cascade for `carry`, the step elided as `...` in Figure 4a.

```
theorem tnumStepUp_le_of_mem (tval tmask z : BitVec 64) ... :
    ∀ w, satisfiesTnum64 w tval tmask → z < w →
      tnumStepUp tval tmask z ≤ w := by
  intro w hw hzw
  set d := z - tval
  -- a member is the base plus a submask of the mask
  obtain ⟨s, hs, rfl⟩ :
      ∃ s, s &&& ~~~tmask = 0 ∧ w = tval ||| s := ...
  -- both bounds become submask bounds on d
  have hdm : d < tmask := ...
  have hds : d < s     := ...
  -- the carry stops at the first zero p of the filled word
  obtain ⟨p, hpz, hlow⟩ : ∃ p,
      (stepFill d tmask).getLsbD p = false ∧
      ∀ j, j < p → (stepFill d tmask).getLsbD j = true := ...
  -- s first exceeds d at bit q; q is also a stop, so p ≤ q
  obtain ⟨q, hdq, hsq, hagree⟩ : ∃ q,
      d.getLsbD q = false ∧ s.getLsbD q = true ∧
      ∀ j, q < j → d.getLsbD j = s.getLsbD j := ...
  have hpq : p ≤ q := ...
  -- hence the increment is least among submasks that beat d
  have hmin : stepInc d tmask ≤ s := ...
  exact or_le_or_of_le h_consistent (stepInc_and_not d tmask) hs
      hmin
```

Figure 9. Our manual proof (preconditions elided as `...`). Each step is a named property.

tasks remain unsolved because the underlying code is incorrect, and proof checking itself can run for many hours, making naive budget increases both insufficient and expensive. Progress therefore depends on workflow-level mechanisms that diagnose code-vs-proof failures and redirect compute accordingly, rather than on brute-force budget expansion.

Model ablation. To disentangle model capability from the agentic framework, we rerun the agent with Sonnet-4.6 on the same 7 plausible tasks together with the 3 solved tasks, holding the harness and the $30 budget fixed. Sonnet-4.6 resolves 0 of the ten tasks, compared with 3 for Opus-4.6. This suggests that, under the same agentic harness, verified code generation on production systems is not yet within reach of smaller models.

6 Discussion and Limitations

6.1 Comparison with Verified-Systems Engineering

To cross-check the agents' output against established practice in verified-systems engineering, we use the tnum task (Section 4.4), for which a single theorem has three artifacts: the agent-produced solution, the kernel patch that we upstreamed, and a reference proof that we wrote by hand. Read together with the other submissions, this comparison exposes two recurring bad practices in agent submissions.

Code is shaped for the verifier. Because each task asks for code together with its proof, agents favor code whose correctness their automation can discharge directly, even when that code departs from the conventions that a maintainer expects. The merged tnum task illustrates this. To compute the increment mask `carry`, the agent uses a shift-or cascade of six constant shifts (Figure 8), and it reaches for the identical cascade in `KBUmax`; the cascade is a pure bitvector term that `bv_decide` settles directly. The kernel idiom for the same quantity is `(1 « fls(...)) - 1`, which our upstreamed patch uses (Figure 4b) and which a kernel developer reads at a glance; however, `fls()` is recursive, and characterizing it requires inductive lemmas that `bv_decide` cannot supply. The agent therefore selects the structure that its verifier rewards. That structure verifies, but it departs from project convention, which lowers reviewability in a long-lived codebase and forces a manual restructuring.

Proofs certify but do not explain. Agents discharge proof obligations largely through automation, without building intermediate structure. The tnum proof closes all five obligations with five `bv_decide` calls (Figure 4a); `KBUmax` discharges its proof with 30 `bv_decide` calls, and `cteRevoke` discharges its invariants in 821 lines of inline tactics; neither states an auxiliary lemma. The same flat, automation-first shape recurs in the submissions that fail to close. Such proofs certify the result, but they neither expose why the algorithm is correct nor leave behind a lemma that a later proof can reuse. This inverts the discipline behind seL4, CompCert [32], and Vellvm [76], whose proofs are carried by libraries of named lemmas. For code that must be reviewed and maintained, the difference matters: at upstream time, a structured proof conveys the algorithmic insight to maintainers, which may in turn reveal other code locations where the same insight applies, whereas an automation-first proof conveys only that the check passed.

For contrast, Figure 9 shows the proof that we wrote by hand. Its top level reads as a sequence of named properties: a member is the base plus a submask, so both range bounds reduce to submask bounds on $d = z - \texttt{tval}$; the increment is a carry that stops at the first zero p of the filled word; and any competing member first exceeds d at a bit q that meets the same stop condition, whence $p \le q$ and the increment is least. Automation is confined to the trivial leaf facts. Several of the supporting lemmas, *e.g.*, the highest-differing-bit lemma, are independent of the task and therefore reusable.

6.2 Limitations

Each task encodes the algorithmic core of an upstream problem as a self-contained Lean 4 theorem, outside the project's own codebase: the Linux eBPF verifier is written in C, and LLVM's semantics are mechanized mainly in Rocq. Carrying a solved task upstream therefore leaves two gaps, one for the proof and one for the implementation. Integrating the proof into a native verification development is limited less by the difference in mechanization than by proof structure:

as Section 6.1 discusses, the agent's flat, automation-first proof does not decompose into the reusable, named lemmas from which such developments are built. Contributing the implementation requires translating the algorithm into the project's language. This step is manual, but the specification makes it tractable: the port can be verified independently, as when the upstreamed tnum algorithm was translated to C and cross-checked with CBMC by a kernel maintainer [80]. The separation is a trade-off: a single encoding language lets Vero measure whether an agent can derive and prove the algorithmic core, and it gives a common baseline across the three domains. In addition, Vero currently draws its tasks from three projects. The suite will expand as these projects evolve, and we plan to extend it to further upstream projects.

7 Related Work

Verified code generation. Several benchmarks [15, 69] evaluate LLM-based verified code generation. Clover [57] provides 60 manually annotated Dafny programs at textbook difficulty with consistency checking. CLEVER [63] adapts 161 HumanEval problems to Lean 4 with non-computable specifications that prevent models from copying specification logic into implementations. Verina [74] curates 189 Lean 4 tasks from MBPP and student submissions, evaluating code, specification, and proof generation both independently and jointly. AlgoVeri [75] aligns 77 classical algorithms across Dafny, Verus, and Lean, targeting properties such as graph reachability and tree balance that require global reasoning. Vero differs in that every task originates from a production system, encodes a new requirement with no existing solution in the project, and benefits the upstream when solved.

A separate line of benchmarks targets *repository-level* verified development [68, 69, 77], in which both code and proof span a whole project; the central challenge there is to understand the project context and to reuse its existing lemmas and code across files. Vero measures a complementary, *problem-level* capability: each task is a single self-contained problem whose code demands algorithmic insight and whose proof must establish new core lemmas for that problem rather than reuse those that a project already provides.

Other work focuses on designing verified code generation approaches. AutoVerus [70] deploys multiple specialized LLM agents for Verus proof generation, achieving over 90% on a suite of 150 programs. VeruSAGE [71] scales to more proof tasks drawn from eight Verus-verified systems and compares a *hands-on* agent encoding Verus-specific proof strategies against a *hands-off* agent that provides only a generic coding tool with verifier access. The key finding is that hands-on guidance helps weaker models, while the hands-off approach is superior for the most capable model (Claude Sonnet 4.5, 81% solve rate). Misu et al. [40] study end-to-end Dafny synthesis from natural-language descriptions, generating both specifications and implementations; AlphaVerus [2] bootstraps Verus code generation through tree-search refinement with verifier feedback. Our evaluation follows the hands-off paradigm, consistent with the finding that capable models perform better with autonomy. The failure modes in Vero are mainly due to the new requirements, *e.g.*, cross-domain reasoning in the eBPF tasks.

Theorem proving and code generation. Automated theorem proving establishes the validity of formal theorems. SMT solvers, *e.g.*, Z3 [14] and CVC5 [5], have been used to discharge proof obligations automatically. LLMs have recently increased the degree of automation in interactive theorem proving (ITP) systems such as Lean [34, 72]. For example, LeanDojo [73] and Rango [64] provide retrieval-augmented premise selection, DSP [27] introduces formal sketch generation combined with informal reasoning, and Baldur [21] generates proofs through error-driven repair.

For code generation, one line of work is driven by deep learning models. For example, CodeTree [33] and LEVER [44] use execution feedback through tree search and learned verification models, respectively, to refine candidate programs. The other line of work, *i.e.*, program synthesis, searches for implementations from specifications [36]. For example, SyGuS [3] provides a framework for syntax-guided search, and Sketch [54] instantiates program templates by employing counterexample-guided inductive synthesis (CEGIS). FlashFill [24] leverages version-space algebra to search for programs that satisfy input-output examples.

Vero poses a challenge at the intersection of these two lines: a solution requires jointly producing code and proofs, where implementation choices constrain the proof strategy and verifier errors may demand changes to both.

8 Conclusion

We presented Vero, 103 verified code generation tasks from three production systems where every specification encodes a new requirement and a correct solution directly benefits the upstream project. Agents that nearly saturate prior benchmarks solve only 2.9% of Vero, as each task demands that the agent diagnose the algorithmic barrier, derive a novel implementation, discharge the resulting proof obligations, and allocate effort across these stages. The solved tasks have yielded verified patches contributed back to their upstream projects, demonstrating Vero's practical value. The unsolved tasks define a concrete agenda for future work.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback on an earlier version of this paper. We are grateful to our shepherd for the prompt and constructive guidance that helped strengthen our paper. This work was partially supported by a research award from the eBPF Foundation and a research award from the Hasler Foundation, which we appreciate and acknowledge.

References

[1] 2022. CVE-2022-23222. https://nvd.nist.gov/vuln/detail/CVE-2022-23222.

[2] Pranjal Aggarwal, Bryan Parno, and Sean Welleck. 2025. AlphaVerus: bootstrapping formally verified code generation through self-improving translation and treefinement. In *Proceedings of the 42nd International Conference on Machine Learning* (Vancouver, Canada) *(ICML'25)*. JMLR.org, Article 24, 29 pages.

[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. IEEE, 1–8.

[4] Andrea Arcangeli. 2012. Seccomp (Secure Computing Mode) Documentation. https://docs.kernel.org/userspace-api/seccomp_filter.html. Accessed: 2025-03-12.

[5] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: A versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 415–442.

[6] 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. 2025. A benchmark for vericoding: formally verified program synthesis. arXiv:2509.22908 [cs.SE] https://arxiv.org/abs/2509.22908

[7] Paul Chaignon. 2026. bpf: Fix invariant violations and improve branch detection. Linux kernel mailing list, https://lore.kernel.org/bpf/cover.1775142354.git.paul.chaignon@gmail.com/. Accessed: August 27, 2026.

[8] Paul Chaignon and Harishankar Vishwanathan. 2026. bpf: Improve bounds when tnum has a single possible value. Linux kernel commit, https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/commit/?id=efc11a667878. Accessed: August 27, 2026.

[9] Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, Cheng Ren, Jiawei Shen, Wenlei Shi, Tong Sun, He Sun, Jiahui Wang, Siran Wang, Zhihong Wang, Chenrui Wei, Shufa Wei, Yonghui Wu, Yuchen Wu, Yihang Xia, Huajian Xin, Fan Yang, Huaiyuan Ying, Hongyi Yuan, Zheng Yuan, Tianyang Zhan, Chi Zhang, Yue Zhang, Ge Zhang, Tianyun Zhao, Jianqiu Zhao, Yichi Zhou, and Thomas Hanwen Zhu. 2025. Seed-Prover: Deep and Broad Reasoning for Automated Theorem Proving. arXiv:2507.23726 [cs.AI] https://arxiv.org/abs/2507.23726

[10] Tianyu Chen, Shuai Lu, Shan Lu, Yeyun Gong, Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Hao Yu, Nan Duan, Peng CHENG, et al. 2025. Automated Proof Generation for Rust Code via Self-Evolution. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/pdf?id=2NqssmiXLu

[11] Cilium. 2022. Verifier Imprecision in Tetragon. https://github.com/cilium/tetragon/blob/6fd02f91d15b93e75ae42df7ac131c5ebff77c41/bpf/lib/process.h#L70. Accessed: 2025-03-12.

[12] Jonathan Corbet. 2019. BPF at Facebook (and beyond). *LWN.net* (October 2019). https://lwn.net/Articles/801871/

[13] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. doi:10.1145/512950.512973

[14] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[15] Xun Deng, Sicheng Zhong, Barış Bayazıt, Andreas Veneris, Fan Long, and Xujie Si. 2025. VerifyThisBench: Generating Code, Specifications, and Proofs All at Once. arXiv:2505.19271 [cs.SE] https://arxiv.org/abs/2505.19271

[16] Edsger W. Dijkstra. 1976. *A discipline of programming*. prentice-hall Englewood Cliffs.

[17] Quinn Dougherty and Ronak Mehta. 2025. Proving the coding interview: A benchmark for formally verified code generation. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*. IEEE, 72–79.

[18] Oliver Dressler. 2025. *Lean LSP MCP: Tools for agentic interaction with the Lean theorem prover*. https://github.com/oOo0oOo/lean-lsp-mcp

[19] eBPF Community. 2024. eBPF Case Studies. https://ebpf.io/case-studies/. Accessed: 2025-03-12.

[20] Elastic. 2021. Elastic eBPF Deployment. https://github.com/elastic/ebpf. Accessed: 2025-03-12.

[21] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1229–1241. doi:10.1145/3611643.3616243

[22] Cameron Freer. 2025. *Lean 4 Skills: Theorem proving skill and workflow pack for AI coding agents*. https://github.com/cameronfreer/lean4-skills

[23] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions. In *Proceedings of PLDI*. 1069–1084. doi:10.1145/3314221.3314590

[24] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.

[25] Daniel Hodges. 2024. Scheduling at Scale: eBPF Schedulers with Sched_ext. USENIX Association, Dublin.

[26] Jinghao Jia, YiFei Zhu, Dan Williams, Andrea Arcangeli, Claudio Canella, Hubertus Franke, Tobin Feldman-Fitzthum, Dimitrios Skarlatos, Daniel Gruss, and Tianyin Xu. 2023. Programmable System Call Security with eBPF. arXiv:2302.10366 [cs.OS] https://arxiv.org/abs/2302.10366

[27] Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothee Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. 2023. Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal Proofs. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=SMa9EAovKMC

[28] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220. doi:10.1145/1629575.1629596

[29] Helen Koike. 2026. bpf: deduce_bounds_64_from_32 tightening with circular range logic. Linux kernel mailing list, https://lore.kernel.org/bpf/20260410124035.297632-1-koike@igalia.com/. Accessed: August 27, 2026.

[30] Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, Ryan Bloom, Thomas Broadley, Haoxing Du, Brian Goodrich, Nikola Jurkovic, Luke Harold Miles, Seraphina Nix, Tao Lin, Chris Painter, Neev Parikh, David Rein, Lucas Jun Koba Sato, Hjalmar Wijk, Daniel M. Ziegler, Elizabeth Barnes, and Lawrence Chan. 2025. Measuring AI Ability to Complete Long Software Tasks. arXiv:2503.14499 [cs.AI] https://arxiv.org/abs/2503.14499

[31] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California)

(CGO '04). IEEE Computer Society, USA, 75.

[32] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

[33] Jierui Li, Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2024. CodeTree: Agent-guided Tree Search for Code Generation with Large Language Models. arXiv:2411.04329 [cs.CL] https://arxiv.org/abs/2411.04329

[34] Zhaoyu Li, Jialiang Sun, Logan Murphy, Qidong Su, Zenan Li, Xian Zhang, Kaiyu Yang, and Xujie Si. 2024. A Survey on Deep Learning for Theorem Proving. In *First Conference on Language Modeling*. https://openreview.net/forum?id=zlw6AHwukB

[35] Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, Jiayun Wu, Jiri Gesi, Ximing Lu, David Acuna, Kaiyu Yang, Hongzhou Lin, Yejin Choi, Danqi Chen, Sanjeev Arora, and Chi Jin. 2025. Goedel-Prover-V2: Scaling Formal Theorem Proving with Scaffolded Data Synthesis and Self-Correction. arXiv:2508.03613 [cs.LG] https://arxiv.org/abs/2508.03613

[36] Zohar Manna and Richard Waldinger. 1980. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2, 1 (1980), 90–121.

[37] Zohar Manna and Richard J Waldinger. 1971. Toward automatic program synthesis. *Commun. ACM* 14, 3 (1971), 151–165.

[38] Meta. 2018. A high performance layer 4 load balancer. https://github.com/facebookincubator/katran. Accessed: 2025-03-12.

[39] Brando Miranda, Zhanke Zhou, Allen Nie, Elyas Obbad, Leni Aniva, Kai Fronsdal, Weston Kirk, Dilara Soylu, Andrea Yu, Ying Li, and Sanmi Koyejo. 2025. VeriBench: End-to-End Formal Verification Benchmark for AI Code Generation in Lean 4. In *2nd AI for Math Workshop @ ICML 2025*. https://openreview.net/forum?id=rWkGFmnSNl

[40] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. 2024. Towards AI-Assisted Synthesis of Verified Dafny Methods. *Proc. ACM Softw. Eng.* 1, FSE, Article 37 (July 2024), 24 pages. doi:10.1145/3643763

[41] Leonardo de Moura and Sebastian Ullrich. 2021. The lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*. Springer, 625–635.

[42] Andrii Nakryiko. 2023. bpf: register bounds logic and testing improvements. Linux kernel mailing list, https://lore.kernel.org/bpf/20231102033759.2541186-1-andrii@kernel.org/. Accessed: August 27, 2026.

[43] Netflix. 2024. bpftop. https://github.com/Netflix/bpftop. Accessed: 2025-03-12.

[44] Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I. Wang, and Xi Victoria Lin. 2023. LEVER: learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 1086, 23 pages.

[45] OpenAI. 2024. Introducing SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/. Accessed: August 27, 2026.

[46] Oracle. 2023. bpftune. https://github.com/oracle/bpftune. Accessed: 2025-03-12.

[47] Nikita Popov. 2020. Make LLVM fast again. Blog post, https://www.npopov.com/2020/05/10/Make-LLVM-fast-again.html. Accessed: August 27, 2026.

[48] Ramkumar Ramachandra. 2026. [RFC] Optimal KnownBits using a Semilattice. LLVM Discourse, https://discourse.llvm.org/t/rfc-optimal-knownbits-using-a-semilattice/89565. Accessed: August 27, 2026.

[49] John Regehr. 2020. Precision Opportunities for Demanded Bits in LLVM. Blog post, https://blog.regehr.org/archives/1714. Accessed: August 27, 2026.

[50] Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. 2025. DeepSeek-Prover-V2: Advancing Formal Mathematical Reasoning via Reinforcement Learning for Subgoal Decomposition. arXiv:2504.21801 [cs.CL] https://arxiv.org/abs/2504.21801

[51] sched ext. 2024. Verifier imprecision in sched_ext. https://github.com/sched-ext/scx/blob/5933fc301ed1eee59bbe0a61da279748cce6b43a/scheds/rust/scx_layered/src/bpf/util.bpf.c#L24. Accessed: 2025-03-12.

[52] seL4 Project. 2025. The sel4bench Suite. https://docs.sel4.systems/projects/sel4bench/. Accessed: 2026-04-13.

[53] seL4 Project. 2026. seL4 Haskell Kernel Model: Kernel Initialisation (Init.lhs). https://github.com/seL4/l4v/blob/1ad54b0bfbdbfd9f49b37685200f285e15051c33/spec/haskell/src/SEL4/Kernel/Init.lhs. Accessed: August 27, 2026.

[54] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. 404–415.

[55] Alexei Starovoitov and Daniel Borkmann. 2021. The eBPF Subsystem. https://docs.kernel.org/bpf/. Accessed: 2025-03-12.

[56] Alexei Starovoitov and Daniel Borkmann. 2024. The eBPF Verifier. https://docs.kernel.org/bpf/verifier.html. Accessed: 2025-03-12.

[57] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. 2024. Clover: Closed-Loop Verifiable Code Generation. In *AI Verification: First International Symposium, SAIV 2024, Montreal, QC, Canada, July 22–23, 2024, Proceedings* (Montreal, QC, Canada). Springer-Verlag, Berlin, Heidelberg, 134–155. doi:10.1007/978-3-031-65112-0_7

[58] Hao Sun. 2026. bpf: Simplify tnum_step(). https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=833ef4a954e12485b9fd44e4e5eeb349ac194c26. Linux kernel commit 833ef4a9, released in Linux 7.1.

[59] Hao Sun, Zenan Li, Cong Li, and Zhendong Su. 2026. *Vero: Verified Code Generation Benchmark*. https://github.com/SunHao-0/Vero

[60] Hao Sun and Zhendong Su. 2024. Validating the eBPF Verifier via State Embedding. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[61] Hao Sun, Yiru Xu, Jianzhong Liu, Yuheng Shen, Nan Guan, and Yu Jiang. 2024. Finding Correctness Bugs in eBPF Verifier with Structured and Sanitized Program. In *Proceedings of EuroSys '24*. 689–703. doi:10.1145/3627703.3629562

[62] Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing static analyses for precision and soundness. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (San Diego, CA, USA) *(CGO '20)*. Association for Computing Machinery, New York, NY, USA, 81–93. doi:10.1145/3368826.3377927

[63] Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetzsche, Greg Durrett, Yisong Yue, and Swarat Chaudhuri. 2025. CLEVER: A Curated Benchmark for Formally Verified Code Generation. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. https://openreview.net/forum?id=IbOacMF5qd

[64] Kyle Thompson, Nuno Saavedra, Pedro Carrott, Kevin Fisher, Alex Sanchez-Stern, Yuriy Brun, João F. Ferreira, Sorin Lerner, and Emily First. 2025. Rango: Adaptive Retrieval-Augmented Proving for Automated Software Verification. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 347–359. doi:10.1109/ICSE55347.2025.00161

[65] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. 2020. Fast Packet Processing with EBPF and XDP: Concepts, Code, Challenges, and Applications. *ACM Comput. Surv.* 53, 1, Article 16 (feb 2020), 36 pages. doi:10.1145/3371038

[66] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2022. Sound, Precise, and Fast Abstract Interpretation With Tristate Numbers. In *Proceedings of CGO*. 254–265.

[67] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2023. Verifying the Verifier: eBPF Range Analysis Verification. In *CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part III* (Paris, France). Springer-Verlag, Berlin, Heidelberg, 226–251. doi:10.1007/978-3-031-37709-9_12

[68] Huajian Xin, Zheng Yuan, Jacques D. Fleuriot, and Wenda Li. 2026. APE-Bench: Evaluating Automated Proof Engineering for Formal Math Libraries. In *Forty-third International Conference on Machine Learning*. https://openreview.net/forum?id=OjeKfGXLOn

[69] Yutong Xin, Qiaochu Chen, Greg Durrett, and Işil Dillig. 2026. VeriSoft-Bench: Repository-Scale Formal Verification Benchmarks for Lean. arXiv:2602.18307 [cs.SE] https://arxiv.org/abs/2602.18307

[70] Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. 2025. AutoVerus: Automated Proof Generation for Rust Code. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 396 (Oct. 2025), 29 pages. doi:10.1145/3763174

[71] Chenyuan Yang, Natalie Neamtu, Chris Hawblitzel, Jacob R Lorch, and Shan Lu. 2025. VeruSAGE: A Study of Agent-Based Verification for Rust Systems. *arXiv preprint arXiv:2512.18436* (2025). https://arxiv.org/abs/2512.18436

[72] Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin Lauter, Swarat Chaudhuri, and Dawn Song. 2026. Formal Reasoning Meets LLMs: Toward AI for Mathematics and Verification. *Commun. ACM* 69, 3 (Feb. 2026), 66–73. doi:10.1145/3750038

[73] Kaiyu Yang, Aidan M Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. 2023. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

[74] Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. 2026. VERINA: Benchmarking Verifiable Code Generation. In *The Fourteenth International Conference on Learning Representations*. https://openreview.net/forum?id=0A4Uf88pog

[75] Haoyu Zhao, Ziran Yang, Jiawei Li, Deyuan He, Zenan Li, Chi Jin, Venugopal V. Veeravalli, Aarti Gupta, and Sanjeev Arora. 2026. AlgoVeri: An Aligned Benchmark for Verified Code Generation on Classical Algorithms. arXiv:2602.09464 [cs.SE] https://arxiv.org/abs/2602.09464

[76] Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 427–440.

[77] Si Cheng Zhong and Xujie Si. 2025. Towards Repository-Level Program Verification with Large Language Models. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Language Models and Programming Languages* (Singapore, Singapore) *(LMPL '25)*. Association for Computing Machinery, New York, NY, USA, 27–39. doi:10.1145/3759425.3763382

[78] Eduard Zingerman. 2025. bpf: replace min/max fields with struct cnum{32,64}. Linux kernel mailing list, https://lore.kernel.org/bpf/0c47b0b7ea476647746806c46fded4353be885f7.camel@gmail.com/T/. Accessed: August 27, 2026.

[79] Eduard Zingerman. 2026. bpf: Fix u32/s32 bounds when ranges cross min/max boundary. Linux kernel commit, https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/commit/?id=fbc7aef517d8. Accessed: August 27, 2026.

[80] Eduard Zingerman. 2026. Correctness verification of tnum_step using CBMC. Linux kernel thread, https://lore.kernel.org/bpf/9a6dd99e31bae75da12eaa18ddc7424534d13621.camel@gmail.com/. Accessed: August 27, 2026.