PRECISE INVARIANT GENERATION VIA INTERLEAVED AGENTIC REASONING AND PROGRAM ANALYSIS

Anonymous authors
Paper under double-blind review

ABSTRACT

Programmers increasingly rely on Large Language Models (LLMs) to generate code, yet nothing guarantees the code they generate is correct. Recent work has attempted to mitigate this issue by leveraging formal verification tools to certify that LLM-written code is correct. However, existing verifiers cannot scale to real-world software unless they are provided with invariants to guide the verification process. Generating correct and useful invariants is a long-standing unsolved problem: existing formal methods approaches cannot handle larger programs, while LLM-based approaches struggle to reason about the low-level semantics necessary to generate precise invariants. We present a novel approach to invariant generation that achieves the best of both worlds, interleaving LLM-based invariant generation with structured formal analyses. Our strategy first uses program transformations to reduce the input program and tailor its structure for LLM-based reasoning. It then runs a medley of existing and novel analyses on the input program to construct a formal knowledge base of program analysis information. The combination of the reduced program with the formal knowledge base enables an invariant generation agent to produce invariants that are correct and precise enough for program verification. We implement this approach in a tool (STRIVE). STRIVE solves up to $3.9\times$ as many problems as existing LLM-based approaches and up to $5.7\times$ as many as solver-based approaches.

1 INTRODUCTION

The rise of coding agents based on large language models (LLMs) has dramatically accelerated software engineering, enabling the rapid generation of large volumes of code. Coding agents now use tools to compile, test, and debug generated code (Yang et al., 2024; Jimenez et al., 2024). However, testing is fundamentally incomplete: it cannot check every possible behavior of an LLM-written program. Fully verifying that LLM-generated code is correct remains a challenge because the tools of formal verification have not kept up with the pace of development of coding agents: current program verifiers are prohibitively slow when checking meaningful properties of realistic programs, especially when loops, complex control flow, or non-trivial interprocedural interactions are involved (Beyer, 2026).

A key stepping stone to formally verifying the correctness of these complex code patterns is the generation of program invariants (Hoare, 1969). Invariants are formulas that hold true, for example, at every iteration of a loop; they summarize a loop's semantics and are used as sub-goals of the whole-program correctness proof, which is typically discharged by a constraint solver. Unfortunately, writing invariants is challenging; indeed, the task is undecidable for programs in general (Rice, 1953). Existing approaches to invariant generation follow one of two directions. First are methods based on formal reasoning and program analysis (Ernst et al., 2001; Rodríguez-Carbonell & Kapur, 2004; Riley & Fedyukovich, 2022; Nguyen et al., 2022); these approaches either handle only a small subset of loops or are too brittle or slow-running to handle realistic code. More recent work has proposed harnessing neural techniques including LLMs to generate invariants instead (Padhi et al., 2018; Kamath et al., 2023; Wen et al., 2024; Wu et al., 2024b). While LLMs often scale better than traditional approaches to invariant generation, they sometimes struggle with the low-level semantics of loop bodies, and can produce invariants that are incorrect or imprecise, which wastes solver time.

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

In this work, we propose a new method of generating invariants for code verification that combines the precision of program analysis with the scalability of LLM-based reasoning. Our key insight is that information gleaned from program analysis can help guide LLM-based invariant generation to make program verification scalable. Our approach, STRIVE, therefore interleaves program analysis and LLM-based reasoning. Given an input program P, we first reduce P to a simpler program, P', using a set of formally verified code transformations. We then use a medley of both dynamic and static program analyses—including a novel mathematical pattern-matching analysis—to produce a knowledge base $\mathcal{K}_{P'}$ of analysis information about P'. In addition, we derive more aggressively transformed programs P_T from P'. These are variants of the program run through potentially unsound transformations aimed at simplifying the code further. These transformations are in some cases lifted from compiler intermediate representation via LLM. An agent is provided with these P_T to generate invariants $\mathcal{I}_T$ that hold over the unverified transformed code. We then provide an agent with P', $\mathcal{K}_{P'}$, P_T, and $\mathcal{I}_T$ to produce invariants $\mathcal{I}_{P'}$. These invariants are used during verification and prove the goals of the reduced program P'.

Across six benchmark suites, STRIVE proves $5.7\times$ and $4.6\times$ as many programs as the bounded model checkers CBMC and ESBMC respectively, $1.5\times$ as many as the interpolation-based verifier UAutomizer, and 1.9–$3.9\times$ as many as state-of-the-art LLM-based tools using the same underlying model. The individual components of the analyses and transformations each solve multiple unique benchmarks. With Sonnet, verified reductions uniquely solve 18, while unverified transformations uniquely solve 20, and program analysis results uniquely solve 12. The combination of these uniquely solves 11 not solved by any component individually.

In sum, we make the following contributions:

- **A novel invariant generation method that interleaves agentic reasoning and program analysis:** We present a strategy for invariant generation that combines program analysis with LLM-based reasoning in three ways: by transforming programs into a reduced form, by structuring analysis information into a knowledge base, and then by extrapolating the generated invariants back to the original program.
- **A novel mathematical pattern-matching analysis for loop invariants:** We introduce an analysis technique that suggests closed-form loop invariants by leveraging a database of number theory results in integer sequences.
- **A new invariant generation benchmark suite:** We construct a new suite for evaluating approaches to invariant generation consisting of more complex programs, a subset of which contains dead code.

2 Related Work

Dynamic Analysis: A common technique is to use dynamic analysis to perform invariant generation. This can be done through modified execution, such as symbolic or concolic execution (King, 1976; Csallner et al., 2008; Nguyen et al., 2022). Another popular method is post-execution analysis of program traces (Ernst et al., 2001; Nguyen et al., 2014). Trace analysis also led to a pairing with verifiers, where a candidate is generated from the trace and then checked by a verifier (Sharma et al., 2013). This paradigm continues in recent work using other candidate generators. The dynamic techniques which produce complete invariants require running the program through techniques which do not scale well, while those that are sufficiently fast do not produce complete invariants.

Static Analysis: Another approach used for invariant generation is static analysis. In particular, there are works leveraging mathematical analyses (e.g., algebraic (Rodríguez-Carbonell & Kapur, 2004; Ji et al., 2022), recurrence relations (Farzan & Kincaid, 2015; Humenberger et al., 2018), discrete dynamical systems (Jeannet et al., 2014)) on top of abstract interpretation (Cousot & Cousot, 1977; Cousot & Halbwachs, 1978) to summarize behavior and create invariants. Another static approach is attempting coefficient fitting on a pre-established set of equations (Colón et al., 2003; Sankaranarayanan et al., 2004). These static techniques can only categorize the behavior of a small fixed class of programs. As a result, they are not general solutions, but are useful as pre-search tools to introduce candidate invariants prior to general verification (Gupta & Rybalchenko, 2009).

Program Verifiers: Program verifiers are the primary competitors to recent ML-based approaches, as well as serving as their base infrastructure. They encode the problem into either Constrained Horn

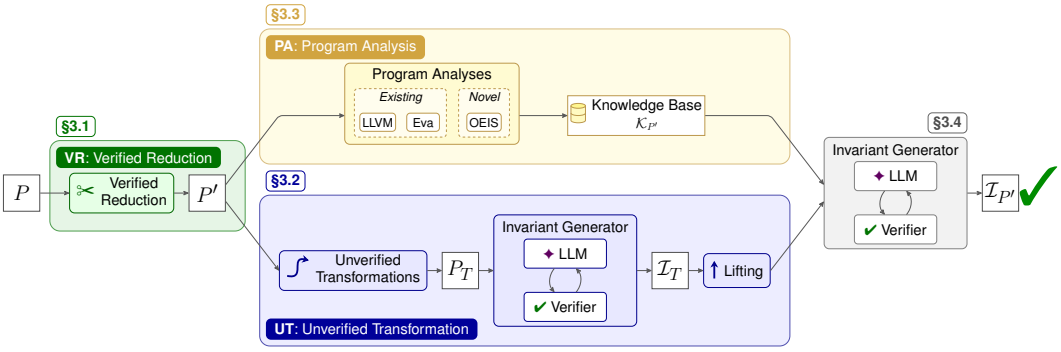


Figure 1: Overview of STRIVE. Note that the OEIS analysis is novel to our work, as well as components within VR and UT.

Clauses (CHC) or Satisfiability Modulo Theories (SMT) constraints and then dispatch a solver to prove the goals (Bjørner et al., 2015). Although this approach allows general program verification, it must be carefully implemented to scale to real-world programs with loops and interprocedural interactions. The most successful implementations are either bounded model checkers or interpolation tools. Bounded model checkers unroll loops until their maximum iteration and calculate the behavior. This handles constant-bounded loops but fails on loops that are symbolically bounded such as when the bound relies on a parameter. Interpolation tools dispatch a solver to attempt to disprove a program path that is a counterexample to the proof goals (Clarke et al., 2000; McMillan, 2003). The tool then reuses what the solver learned to avoid checking paths that share behavior. For our comparison, we use CBMC (Kroening et al., 2023) and ESBMC (Cordeiro & Fischer, 2011), which are state-of-the-art bounded model checkers, and UAutomizer (Heizmann et al., 2015), an interpolation-based tool that won SVCOMP overall most recently in 2026, as well as five other times (Beyer, 2026).

Neural Approaches: Reinforcement learning and non-LLM ML approaches attempt to learn invariant patterns (Si et al., 2018; 2020; Yu et al., 2023). Given a loop or function block, ML approaches suggest associated invariants which are passed to verifiers to check (Ryan et al., 2020; Yao et al., 2020). These approaches tend to work well but lack portability and have largely been subsumed by LLM-based methods. LLM-based tools are also built on top of program verifiers (Kamath et al., 2023; Wu et al., 2024a; Cao et al., 2025). The LLM acts as the candidate invariant generator and the verifier checks that the invariants hold. Once the complex loop and function structures are summarized by valid invariants, the verifier backend may then check the general program behavior. We compare with AutoSpec (Wen et al., 2024), LEMUR (Wu et al., 2024b), and ESBMC ibmc (Pirzada et al., 2024). These tools are similar in inference structure. AutoSpec works on a topological-style ordering of scopes, handling inner loops before outer, and function definitions before their calls. It performs an invariant generation and validation pass for each scope in order before moving on. LEMUR defines a calculus for attempting invariant generation. It focuses on one proof goal at a time and generates a bounded number of invariants to attempt to prove them. As it continues, the tool increases the number of invariants to generate. ESBMC ibmc is an extension of the ESBMC bounded model checking tool which rewrites loops as nodes of verified invariants. Once all loops are rewritten, the problem is dispatched to ESBMC.

3 The STRIVE Approach

Figure 1 gives an overview of STRIVE (**S**tatic **T**ransformations and **R**easoning **I**n**V**ariant **E**ngine)'s approach to invariant generation. Our approach interleaves formal analysis with LLM-based reasoning in three ways. Verified reduction (VR, Section 3.1) is a formal step that simplifies an input program. Unverified transformation (UT, Section 3.2) applies unverified program transformations, generates invariants using LLM-based reasoning, and then applies one more formal step to lift the invariants back to the level of the original input. Program analysis (PA, Section 3.3) builds a structured knowledge base of formal results to provide as input to the final invariant generation agent.

```
1  int f(int n) {            t0();
2    int s = 0, mode = 1;
3    while (s < n) {         t1();
4      if (mode != 1) {      t2();
5        /* complex code */ }
6      s++; }
7    return s; }
```

```
1  int f(int n) {
2    int s = 0, mode = 1;
3    while (s < n) {
4      if (mode != 1) { /* removed */ }
5      s++; }
6    return s; }
```

Figure 2: Each t_i is an empty, non-inlinable static function. CompCert proves `mode != 1` is always false and deletes the call to `t2`, so `t2` no longer appears in the symbol table. STRIVE maps it back to its branch and removes that code from the source.

3.1 Verified Program Reduction

The first category of transformations STRIVE performs is verified reduction of the input program, producing a reduced program P' as in Figure 1. To perform reduction, we harness transformations from the formally verified compiler CompCert (Leroy, 2009), a C compiler with Rocq proofs guaranteeing that all passes and analyses preserve the observable behavior of the code.

Applying CompCert optimizations to the invariant generation context is challenging because, like many compilers, CompCert operates on an intermediate representation rather than on the source languages used by existing benchmark sets for invariant generation. We therefore develop a strategy to map the transformations performed by CompCert back to the level of source code; in essence, we use the compiler as an oracle for which source-level transformations to apply.

Take dead code elimination as an example, shown in Figure 2. We first instrument the start of each function and each branch with a call to a dummy, non-inlinable static function. We then apply CompCert's dead code elimination. If the dummy function symbol is no longer defined in the generated executable, the compiler was able to prove the branch or function unreachable and remove it. We map these branch tags back to the source code level and remove the corresponding branches to produce P'. Assuming trust in the mapping and removal code, this lets STRIVE remove any dead branches from P at the source level while fully preserving program semantics. A comparison of the CompCert-generated assembly before and after the removal is done to further ensure equivalence.

3.2 Unverified Transformations and Invariant Lifting

Although the appeal and feasibility of proven software are increasing with LLM code generation, the current set of formally verified transformations is small. We use only dead code elimination from CompCert, and the verified version of even this pass is far more conservative than the optimizations implemented in most production compilers. Critically for the application domain of invariant generation, formally verified dead code elimination is also incapable of significantly transforming loops. To address these shortcomings, we further transform P' with unverified transformations (UT in Figure 1) to produce program P_T and then introduce a lifting step that maps invariants over P_T back to the level of the reduced program P'.

First, we target loops through the polyhedral analysis model. This model tackles affine loops and allows loops to be fused and split and their iteration spaces simplified. The most mature source-to-source polyhedral transformation tool is Pluto (Bondhugula et al., 2008). This tool is not formally verified and is known to be unsound, producing incorrect transformations in some cases. As a result, we cannot use its output naively. In order to make use of these transformations, we pass the transformed code through the LLM-based invariant generation pipeline. If the system is able to generate invariants and prove the goals on the transformed code, the original code is run through the process again, but now the prompt contains the transformed invariants. The LLM is queried to reference these invariants as potential hints for the original code.

Second, we make use of compiler transformations from LLVM (Lattner & Adve, 2004). We run the following pipeline chosen to canonicalize loops and minimize the number of terms required in invariants: `sroa` (converts memory accesses into named SSA values that are easier to reference in invariants), `early-cse` (removes redundant computation to shrink the number of terms), `instcombine` (simplifies expressions to canonical forms; capped at four iterations to avoid bit-manipulation), `simplifycfg` (simplifies branches), `loop-simplify` and `lcssa` (canonical-

ize loops), and `loop-rotate` and `licm` (run together under MemorySSA; produce loops that execute at least once, with unimportant loop-invariant computation hoisted out of the loop body). The result of these passes is a transformed version of the program in LLVM intermediate representation. Because the verifier operates on C source rather than on the intermediate representation, we have an agent lift the transformed program back to source level. The lifted code is run through the invariant generation process, and invariants proven on it are supplied as hints for proving the original program's goals.

Lifting back to source is not necessary for the polyhedrally transformed code as it is performed source-to-source. These transformed codes are run through invariant generation separately to minimize conflation and the fallout of one tool performing unsound transformation. Each transformation style contributes its own set of candidate invariants. Where their hints disagree, we make no attempt to reconcile them—both are placed in the prompt, and the model is left to determine which, if either, applies to the original code. Because every hint is subject to the same re-proof against the original program, a conflict costs at most an additional repair iteration. Candidate invariant lifting back to the original problem code is a complex process. The loops described by the transformed invariants may differ completely in style, variable names, and even behavior. As a result, the lifting of these invariants into the original code is a difficult task with reasonable success only becoming possible in recent years due to the advancement of LLMs. In practice, STRIVE implements lifting as a substep of the final invariant generation agent loop.

Note that even if all code transformations used were sound, the invariants generated and proven for the transformed code are not necessarily reusable in the original source code due to lost information. For this reason, the approach is useful mostly during the proof process as hints rather than for generating original source specification for programs.

3.3 Program Analysis Knowledge Base

In addition to the program transformations described in Sections 3.1 and 3.2, we harness formal reasoning to construct a knowledge base of analysis information that guides the invariant generator in generating summary invariants (PA in Figure 1).

First, we dump the results of `LoopAnalysis` and `ScalarEvolution` from LLVM. `LoopAnalysis` gives basic information about each loop nest, and `ScalarEvolution` finds patterns of loops, deriving mathematical expressions on the induction variable which capture linear aggregation loops and other closed-form representations closely related to the useful invariant over the loop. All of this information is added to $\mathcal{K}_{P'}$ as guiding information for invariant generation.

Eva is a plugin supplied by Frama-C that performs abstract interpretation to determine the domains of variables in the code, which can sometimes give exact values that are extremely useful for invariants. This is the second piece of information added to $\mathcal{K}_{P'}$.

The third analysis is a new technique which matches loop variable values against the Online Encyclopedia of Integer Sequences (OEIS), a curated database of integer sequences that records, for many entries, a closed form for the kth term in a sequence. We only keep a subset of the database with closed forms that are not polynomial. The other patterns can be trivially detected, saving sequence database search time.

To match these sequences with the code, we JIT-compile the loop and execute it for the first 20 iterations. While executing, we record each variable's value at every iteration. Because loop parameters have no concrete value in the extracted loop, we run it three times with the parameters fixed to different values. A variable whose values differ across these runs is not a function of the iteration count alone, so it is dropped. For each of the maintained variables and corresponding outputs, we perform polynomial fitting of degree at most four by finite differences. If the fit is exact for all the values, this closed form is added to $\mathcal{K}_{P'}$ as the invariant. We also filter out trivial sequences such as constant or arithmetic progressions, which scalar evolution already reports.

For the remainder, we search the database. Each variable's sampled values are compared against each candidate sequence a, and we ask whether the two are related linearly: taking a as the independent variable and the sampled variable v as the dependent one, we fit a slope and an intercept

relating them, allowing the sequence to be indexed at an offset. That is, we search for a relation

$$v(k) \;=\; \frac{num \cdot a(k + \mathit{offset})}{den} \;+\; b,$$

evaluated in integer arithmetic, where the slope num/den is a fraction in lowest terms, the intercept b is an integer, and offset is an integer index shift. Slope and intercept are obtained in the usual two-point form—the slope as the ratio of the change in v to the change in a, the intercept as the residual once that slope is applied—taken at the first two index positions where both sequences are defined and take distinct values. The relation is then checked against *every* overlapping term of the two sequences, where a single disagreement rejects it and at least 12 agreeing terms are required for acceptance. Offsets are searched outward from zero in both directions. From all matching sequences and offsets we impose a total ordering that prefers no offset over any offset, a unit slope over any rescaling, simpler fractions, and smaller intercepts. The match ranked highest under this ordering is the one selected, and the description and closed form recorded for that sequence, together with the fitted relation instantiated for the associated variable, are added to $\mathcal{K}_{P'}$.

Like the hints of Section 3.2, the facts in $\mathcal{K}_{P'}$ are potentially unsound. They are used only as prompt context, and invariants generated from them are checked by the verifier against the original program.

3.4 Agentic Harness

To perform the actual invariant generation, we have an inference pipeline that is fairly standard and similar to all existing LLM-based invariant generation tools. The input code with proof goals is provided to an LLM. The LLM is prompted to generate invariants that help prove the goals. The resulting code with infilled invariants goes through three checks. First, we check that the code is equivalent to the input code and there are no axioms added. This is done via tokenization and comparison, and ensures that we are solving the same problem. Second, the code is checked via the basic syntax checker of the Frama-C WP plugin (Correnson, 2014), which checks that there are no missing definitions or other issues with the generated code and invariants. Third, the code with invariants is passed through the solver and verified. Should any of these tests fail, the LLM is reprompted with the information regarding the issue encountered. The solve process can also give a timeout instead of a failure, which results in a reprompt to add or refine the invariants provided.

3.5 Construction of looptrans

In order to create the looptrans suite, Claude Code with Opus 5 on high reasoning was prompted to create a dataset targeting 50 files of varying difficulty invariant generation problems with both concrete and symbolic proof goals. These benchmark files were requested to cover simple (e.g., constant), linear, polynomial, and non-polynomial invariants. Additionally, it was requested that some examples include dead code (as existing benchmarks are small-scale programs without this). It was told to draw examples and structure from existing benchmarks of HPC kernels and loop transformation tests. These include HecBench (Jin & Vetter, 2023), Pluto polyhedral transformation tests, PolyBench examples (Pouchet, 2012), and GCC/LLVM loop optimization tests. For all tests drawn from existing benchmarks, it was asked to provide a link to the source material. We then manually refactored each file and validated its invariants.

4 Evaluation

We conduct experiments on STRIVE to answer three research questions:

1. Can interleaved program analysis and agentic reasoning with STRIVE generate invariants for programs where existing approaches fail? (Section 4.1)
2. Which components of STRIVE contribute the most to the results? (Section 4.2)
3. Does interleaving program analyses with LLM reasoning affect token cost? (Section 4.3)

We evaluate STRIVE with the following established benchmarks: `code2inv` (Si et al., 2020), a set of mostly symbolically bounded single loop files; `SVCOMP`, a series of more difficult benchmarks assembled from the SVCOMP competition; `dig_hola`, a variant of the HOLA benchmarks

Table 1: Instances per benchmark solved on any attempt (3x300s for LLM-based, 1x900s for solver-based). Bolded values mark the best result per column.

	Method	Model	code2inv	SVCOMP	dig_hola	frama_c	fib_46	looptrans	*Total*
	# Instances		133	91	46	53	46	52	421
Solver	CBMC	–	10	2	1	43	2	0	58
Solver	ESBMC	–	14	3	3	**46**	4	2	72
Solver	UAutomizer	–	99	7	**39**	31	27	13	216
LLM-based	AutoSpec	Sonnet	117	13	0	18	23	0	171
LLM-based	AutoSpec	Gemini	106	11	0	2	22	0	141
LLM-based	AutoSpec	GPT	119	12	0	20	22	0	173
LLM-based	ESBMC ibmc	Sonnet	121	3	17	1	16	0	158
LLM-based	ESBMC ibmc	Gemini	98	3	14	0	0	0	115
LLM-based	ESBMC ibmc	GPT	113	3	17	0	15	0	148
LLM-based	LEMUR	Sonnet	31	11	15	17	11	0	85
LLM-based	LEMUR	Gemini	21	8	8	18	0	0	55
LLM-based	LEMUR	GPT	19	10	9	18	9	0	65
Ours	**STRIVE**	Sonnet	**126**	**43**	26	**46**	**45**	**42**	**328**
Ours	**STRIVE**	Gemini	115	21	4	22	30	1	193
Ours	**STRIVE**	GPT	101	14	2	18	29	0	164

with multiple loops; `frama_c`, a set of benchmarks taken from the Frama-C invariant specification language examples; and `fib_46` (Dillig et al., 2013), the original HOLA benchmark of 46 files compiled from multiple different sources. Finally, we use `looptrans` as described in Section 3.5. Unlike the other benchmarks, `frama_c` includes invariants for program features not supported by other benchmarks, including pointer alias relationships.

During evaluation, we give the LLM-based tools (STRIVE, AutoSpec, ESBMC ibmc, LEMUR) three attempts per benchmark file with a 300 s timeout per attempt. For verifier-based tools (CBMC, ESBMC, UAutomizer) we provide a single 900 s overall timeout, the same amount of total time as the LLM-based approaches. We evaluate the LLM-based tools on commercial models gpt-5.4-mini (OpenAI), gemini-3.5-flash-lite (Google) at default settings, and claude-sonnet-5 (Anthropic) through the Claude Code SDK with high reasoning and all tools and user settings disabled.

4.1 RQ1: Comparison Against Existing Tools

In Table 1, we report the results of running STRIVE and several existing tools on each benchmark set. For benchmarks `code2inv`, `SVCOMP`, `fib_46`, and `looptrans`, STRIVE outperforms all solver-based and LLM-based existing approaches, for example solving 43 `SVCOMP` instances compared to 13 for AutoSpec. On the `frama_c` benchmark suite, STRIVE ties with ESBMC for the best performance; ESBMC performs well on `frama_c` because the benchmark contains several concretely bounded loops and non-loop functions that it can fully unroll, making the problems trivial. UAutomizer also outperforms STRIVE on the `dig_hola` suite; this benchmark set differs from the others because it passes symbolic variables as function parameters, forcing STRIVE to reason about the relationship of these parameters to the function's result.

On `looptrans`, STRIVE is the only tool using LLM inference to solve any instance. ESBMC can solve 2, and UAutomizer can solve 13, significantly fewer than the 42 solved by STRIVE with Sonnet. AutoSpec, which uses the Frama-C WP backend (Correnson, 2014), the same verifier as STRIVE, cannot handle the dead code in this benchmark suite. It traverses scope-to-scope, including the dead scopes, and tries to rigorously prove invariants over them before moving on. This causes AutoSpec to time out generating invariants over complex code which has no impact on proof goals. Overall, STRIVE achieves the best overall performance, solving 328 instances, 112 more than the next best tool, UAutomizer, at 216.

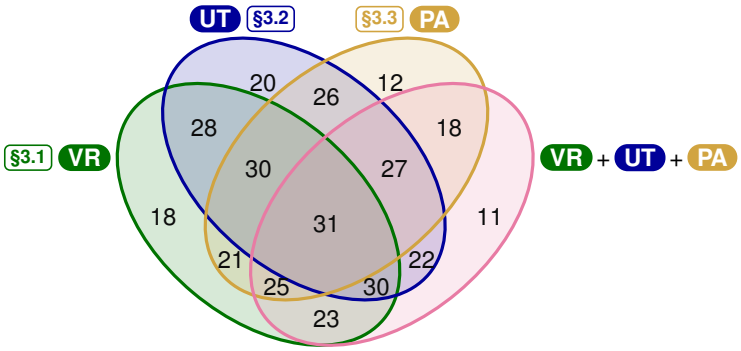


Figure 3: Four-way Venn diagram of unique solves by different variants of STRIVE using Sonnet over only the agentic harness. Overlapping regions contain the union of unique cases solved.

4.2 RQ2: Contribution of Different Components

In order to measure the contributions of each component of STRIVE, we perform an ablation study with five variants of STRIVE. The first variant is our inference harness alone (Section 3.4). We then enable each of the three STRIVE components shown in Figure 1 individually, and finally run STRIVE with all components enabled at once.

Agentic Harness: Using only our harness with the Sonnet model, STRIVE was able to solve 125 problems not solved by other LLM-based tools, and failed to solve 19 benchmarks solved by at least one competing LLM-based tool, a net advantage of 106. For Gemini and GPT, this relationship is flipped: the harness solves 4 and 55 fewer in total, respectively. The results suggest that the context engineering and system design of the other LLM-based tools is particularly helpful for weaker models but reduces efficacy with stronger ones.

Static Transformations and Reasoning: To see the impact of the static transformations and agentic reasoning, we investigate how many unique problems are solved by each variant of STRIVE; the results are given in Figure 3. No single component substantially outperforms the agentic harness on its own, but each adds a few unique solves on certain benchmarks. VR was able to solve 18 problems that were unsolved by the harness or any other variant; likewise, UT solved 20, and PA solved 12. The full STRIVE with all components solved 11 benchmarks on which all individual components failed. Surprisingly, the union of the individual variants solves 31 cases for which the harness fails, compared to just 11 with the combined STRIVE. This difference suggests that combining the specialized analyses and transformations we employ improves performance, but the costs of combination sometimes outweigh the benefit. Part of the reason for this phenomenon is that the UT stage spends a substantial part of the time budget translating back from LLVM intermediate representation to source code level. If the transformations performed by UT are not useful, mapping to source wastes time which could be used on repair. It is also possible that UT and PA may give conflicting or irrelevant information to the generation agent, increasing its reasoning burden.

4.3 RQ3: Effects on Token Usage

Last, we investigate the impact of our approach on inference costs. Figure 4 reports the token usage of each STRIVE variant on the Gemini experiments. On the `looptrans` benchmark, VR significantly decreases token usage, suggesting that the verified dead code elimination step is useful in reducing inference cost on programs that contain dead code—most other benchmarks have relatively little dead code. PA, by contrast, slightly increases token use. UT also adds to inference cost because it performs LLVM IR to source code translation and two separate invariant generation processes. On average, running the process to get candidate invariants on the unverified transformed programs roughly doubles token usage over the harness (+107% input tokens with Gemini), and in the worst case (`code2inv`) it roughly triples it (+205% total tokens). Even with this increase, however, STRIVE sits in the middle of the pack for inference cost, using more tokens than AutoSpec and LEMUR, but only a fraction as many as ESBMC ibmc.

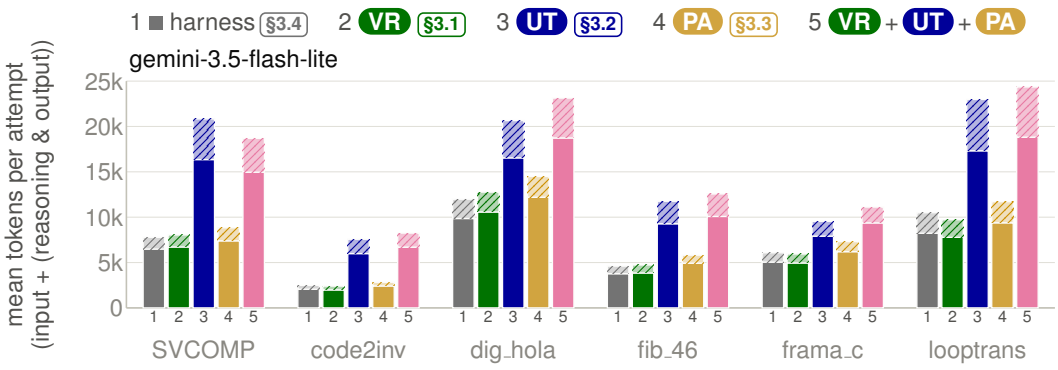


Figure 4: Mean token usage per attempt with Gemini by benchmark and variant. Input tokens are on bottom with solid shading; reasoning and output tokens are on top and striped.

5 Discussion

Our work has several threats to validity to consider. Many benchmarks exist only in the format used by both the CBMC and ESBMC model checkers and UAutomizer, or only in the format used by the Frama-C backend tools STRIVE and AutoSpec. We therefore translated each tool-specific benchmark to make it amenable to the other tools using an LLM-generated script; while we manually inspected the translations and tested them for correctness, there is a small risk of unsoundness in the translation of benchmarks from one tool to another.

Second, the evaluation benchmarks we use have been publicly available long enough to appear in the training data of commercial LLMs. Many of these benchmarks also publish the expected invariants for each case, raising the risk of model memorization. We addressed this limitation by constructing a novel benchmark set, looptrans, which has not been previously published. We also mitigated the risk of memorization on existing benchmarks by using multiple models for each LLM-based tool, allowing paired comparisons between tools using the same model with the same training.

Last, the compared LLM-based tools do not share a verifier backend, so some differences in performance between STRIVE and LEMUR or ESBMC ibmc may reflect differences in the verifier rather than performance on invariant generation. AutoSpec shares a backend with STRIVE, so results are directly comparable.

A key limitation is the small set of available verified transformations. Extending VR is heavily reliant on future developments in verified code transformation. Any new verified code optimization could be added as a component of VR with minimal engineering effort. Our work would also benefit from the development of formally verified source-to-source compilers, which would enable further optimization without the potentially unsound lifting procedures we employ.

Future work could recover invariants directly from transformed invariants without relying on the agent. For example, most of the affine transformations performed in polyhedral transformations are reversible and non-destructive, so it is possible in theory to backtrack the steps and derive invariants on the original loop nests.

6 Conclusion

In this paper, we have presented STRIVE, an invariant generation tool interleaving an agentic reasoning inference pipeline with program analysis techniques. On the tested benchmarks, STRIVE shows great success, consistently showing the best performance out of all tools with better generalizability between benchmarks. Each of the three elements of static transformations and reasoning contributes unique solves that are unable to be achieved without their information. These elements also solve 11 additional instances that are uniquely solved when combining this information. Overall, STRIVE solves 112 more instances than the next-best tool while maintaining a modest, middle-of-the-pack token cost.

AI USE STATEMENT

Generative AI was used to evaluate and polish human-written paragraphs for the paper. It was also used to generate summaries of the LLVM optimization passes used in Section 3.2, although these summaries were significantly edited for clarity and brevity.

Generative AI also assisted in finding related work in the area. In particular, it was used to find recent papers on mathematical static analyses and to ensure no categories of work were missing.

Generative AI was used in execution of the research, in particular for generating the multi-vendor LLM query library shared between all tools, as well as refactoring of the existing tools to use this library. Also, as many benchmarks are either only in a form accepted by Frama-C (ACSL-style) or by program verifiers such as bounded model checkers (CBMC-style), generative AI was used to create translation scripts between these so all tools may be tested. Additionally, it was used to generate the base template of the STRIVE workflow. It also was used to generate the scripts for running the evaluations, and a script to parse the data logs and generate tables for this paper. Lastly, it was used for packaging the anonymized artifact.

Generative AI was also used in the initial creation of the `looptrans` dataset. It was prompted to create a dataset targeting around 50 files of varying difficulty invariant generation problems with both concrete and symbolic proof goals. These benchmark files were requested to cover simple, linear, polynomial, and non-polynomial invariants. Additionally, at least some examples should include dead code (as the existing benchmarks are small-scale toy programs without this). It was told to draw examples from existing loop-based benchmark PolyBench, GCC and LLVM loop optimization/transformation unit tests, and Pluto tests for polyhedral loop rewriting. For all tests drawn from existing benchmarks, it was asked to provide a link to the source material.

For all of these tasks, Claude Code with Opus 5 on high reasoning was used.

REPRODUCIBILITY STATEMENT

All code and required materials to replicate the results of this paper are provided in the anonymous artifact available at Artifact. This contains STRIVE's source code, the benchmarks, the comparison tools with edited source to perform equivalent multi-provider LLM queries, materials on how to run the experiments, and the raw results from all of our runs. For our experiments, we used a node with 192 AMD "Turin" x86-64 cores (8 sockets $\times$ 24 cores, 1 thread/core) and 768 GB RAM. The readme file provided with our anonymized artifact details the exact versions of each software used.

REFERENCES

Dirk Beyer. Evaluating software verifiers for C, Java, and SV-LIB (report on SV-COMP 2026). In *TACAS*, 2026. doi: 10.1007/978-3-032-22749-2_23.

Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II*, volume 9300 of *LNCS*, pp. 24–51. Springer, 2015. doi: 10.1007/978-3-319-23534-9_2.

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pp. 101–113, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595938602. doi: 10.1145/1375581.1375595. URL https://doi.org/10.1145/1375581.1375595.

Weining Cao, Guangyuan Wu, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. Clause2Inv: A generate-combine-check framework for loop invariant inference. *Proceedings of the ACM on Software Engineering*, 2025. doi: 10.1145/3728920.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, pp. 154–169, 2000. doi: 10.1007/10722167_15.

Michael A. Colón, Sriram Sankaranarayanan, and Henny B. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, pp. 420–432, 2003. doi: 10.1007/978-3-540-45069-6_39.

Lucas Cordeiro and Bernd Fischer. Verifying multi-threaded software using smt-based context-bounded model checking. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pp. 331–340, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304450. doi: 10.1145/1985793.1985839. URL https://doi.org/10.1145/1985793.1985839.

Loïc Correnson. Qed. computing what remains to be proved. In Julia M. Badger and Kristin Yvonne Rozier (eds.), *NASA Formal Methods*, pp. 215–229, Cham, 2014. Springer International Publishing. ISBN 978-3-319-06200-6.

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pp. 238–252, 1977. doi: 10.1145/512950.512973.

Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pp. 84–96, 1978. doi: 10.1145/512760.512770.

Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. DySy: dynamic symbolic execution for invariant inference. In *ICSE*, pp. 281–290, 2008. doi: 10.1145/1368088.1368124.

Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. Inductive invariant generation via abductive inference. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2013.

Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001. doi: 10.1109/32.908957.

Azadeh Farzan and Zachary Kincaid. Compositional recurrence analysis. In *FMCAD*, pp. 57–64, 2015.

Ashutosh Gupta and Andrey Rybalchenko. InvGen: An efficient invariant generator. In *CAV*, pp. 634–640, 2009. doi: 10.1007/978-3-642-02658-4_48.

Matthias Heizmann, Daniel Dietsch, Jan Leike, Betim Musa, and Andreas Podelski. Ultimate automizer with array interpolation. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*, pp. 455–457, Berlin, Heidelberg, 2015. Springer-Verlag. ISBN 9783662466803. doi: 10.1007/978-3-662-46681-0_43. URL https://doi.org/10.1007/978-3-662-46681-0_43.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL https://doi.org/10.1145/363235.363259.

Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács. Aligator.jl – a Julia package for loop invariant generation. In *CICM*, pp. 111–117, 2018. doi: 10.1007/978-3-319-96812-4_10.

Bertrand Jeannet, Peter Schrammel, and Sriram Sankaranarayanan. Abstract acceleration of general linear loops. *SIGPLAN Not.*, 49(1):529–540, January 2014. ISSN 0362-1340. doi: 10.1145/2578855.2535843. URL https://doi.org/10.1145/2578855.2535843.

Yucheng Ji, Hongfei Fu, Bin Fang, and Haibo Chen. Affine loop invariant generation via matrix algebra. In *CAV*, pp. 257–281, 2022. doi: 10.1007/978-3-031-13185-1_13.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Zheming Jin and Jeffrey S. Vetter. A benchmark suite for improving performance portability of the sycl programming model. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 325–327, 2023. doi: 10.1109/ISPASS57527.2023.00041.

Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. Finding inductive loop invariants using large language models, 2023. URL https://arxiv.org/abs/2311.07948.

James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7): 385–394, 1976. doi: 10.1145/360248.360252.

Daniel Kroening, Peter Schrammel, and Michael Tautschnig. Cbmc: The c bounded model checker, 2023. URL https://arxiv.org/abs/2302.02384.

Chris Lattner and Vikram Adve. 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*, CGO '04, pp. 75, USA, 2004. IEEE Computer Society. ISBN 0769521029.

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

Kenneth L. McMillan. Interpolation and SAT-based model checking. In *CAV*, pp. 1–13, 2003. doi: 10.1007/978-3-540-45069-6_1.

ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. DIG: A dynamic invariant generator for polynomial and array invariants. *ACM Transactions on Software Engineering and Methodology*, 23(4):30:1–30:30, 2014. doi: 10.1145/2556782.

ThanhVu Nguyen, KimHao Nguyen, and Hai Duong. Syminfer: inferring numerical invariants using symbolic states. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ICSE '22, pp. 197–201, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392235. doi: 10.1145/3510454.3516833. URL https://doi.org/10.1145/3510454.3516833.

Saswat Padhi, Rahul Sharma, and Todd Millstein. Loopinvgen: A loop invariant generator based on precondition inference. *CoRR*, abs/1707.02029, 2018. URL http://arxiv.org/abs/1707.02029.

Muhammad A. A. Pirzada, Giles Reger, Ahmed Bhayat, and Lucas C. Cordeiro. Llm-generated invariants for bounded model checking without loop unrolling. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pp. 1395–1407, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712487. doi: 10.1145/3691620.3695512. URL https://doi.org/10.1145/3691620.3695512.

Louis-Noël Pouchet. PolyBench: The polyhedral benchmark suite. https://sourceforge.net/projects/polybench/, 2012. Version 4.2.1.

H. Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953. URL https://api.semanticscholar.org/CorpusID:120980829.

Daniel Riley and Grigory Fedyukovich. Multi-phase invariant synthesis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pp. 607–619, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394130. doi: 10.1145/3540250.3549166. URL https://doi.org/10.1145/3540250.3549166.

Enric Rodríguez-Carbonell and Deepak Kapur. Automatic generation of polynomial loop invariants: Algebraic foundations. In *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, pp. 266–273, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 158113827X. doi: 10.1145/1005285.1005324. URL https://doi.org/10.1145/1005285.1005324.

Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. CLN2INV: Learning loop invariants with continuous logic networks. In *ICLR*, 2020.

Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Non-linear loop invariant generation using Gröbner bases. In *POPL*, pp. 318–329, 2004. doi: 10.1145/964001.964028.

Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, pp. 574–592, 2013. doi: 10.1007/978-3-642-37036-6_31.

Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In *NeurIPS*, pp. 7762–7773, 2018.

Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. Code2inv: A deep learning framework for program verification. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II*, pp. 151–164, Berlin, Heidelberg, 2020. Springer-Verlag. ISBN 978-3-030-53290-1. doi: 10.1007/978-3-030-53291-8_9. URL https://doi.org/10.1007/978-3-030-53291-8_9.

Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. Enchanting program specification synthesis by large language models using static analysis and program verification. In *Computer Aided Verification: 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part II*, pp. 302–328, Berlin, Heidelberg, 2024. Springer-Verlag. ISBN 978-3-031-65629-3. doi: 10.1007/978-3-031-65630-9_16. URL https://doi.org/10.1007/978-3-031-65630-9_16.

Guangyuan Wu, Weining Cao, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. LLM meets bounded model checking: Neuro-symbolic loop invariant inference. In *ASE*, pp. 406–417, 2024a. doi: 10.1145/3691620.3695014.

Haoze Wu, Clark Barrett, and Nina Narodytska. Lemur: Integrating large language models in automated program verification. In *The Twelfth International Conference on Learning Representations*, 2024b. URL https://openreview.net/forum?id=Q3YaCghZNt.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=mXpq6ut8J3.

Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. Learning nonlinear loop invariants with gated continuous logic networks. In *PLDI*, pp. 106–120, 2020. doi: 10.1145/3385412.3385986.

Shiwen Yu, Ting Wang, and Ji Wang. Loop invariant inference through smt solving enhanced reinforcement learning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, pp. 175–187, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702211. doi: 10.1145/3597926.3598047. URL https://doi.org/10.1145/3597926.3598047.

A Additional Figures

Figures 5 and 6 show token usage for the two models not shown in Figure 4. Input tokens were not logged for Sonnet, so its chart shows only reasoning and output tokens.

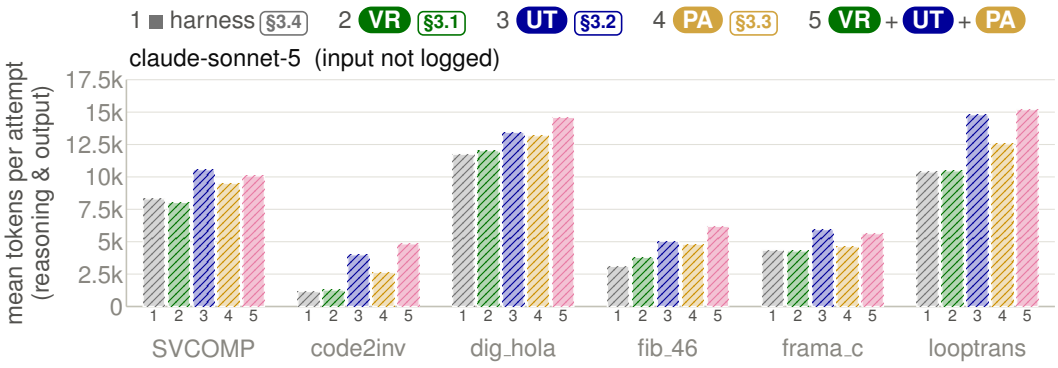


Figure 5: Mean token usage per attempt with Sonnet by benchmark and variant (input tokens not logged)

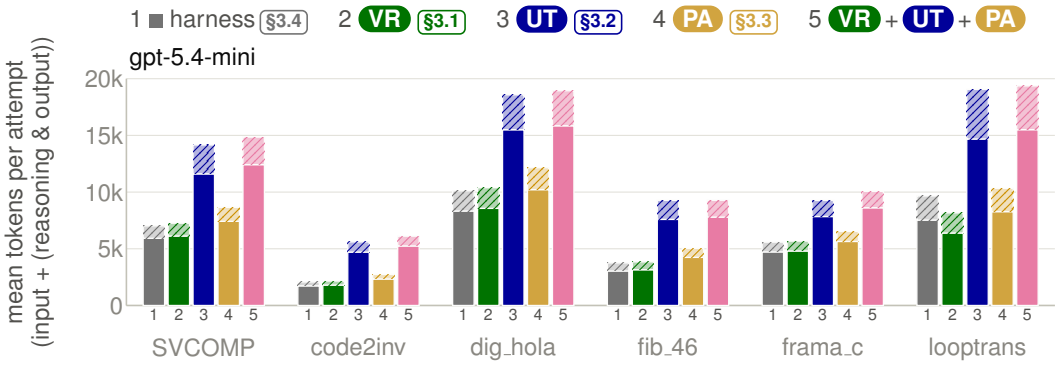


Figure 6: Mean token usage per attempt with GPT by benchmark and variant

Table 2: Instances per benchmark solved on any attempt (3x300s) by each STRIVE variant. Bolded values mark the best variant per column for each model. **Sonnet** = claude-sonnet-5; **Gemini** = gemini-3.5-flash-lite; **GPT** = gpt-5.4-mini.

	Variant	Model	code2inv	SVCOMP	dig_hola	frama_c	fib_46	looptrans	*Total*
	# Instances		133	91	46	53	46	52	421
STRIVE variants	harness	Sonnet	**126**	43	32	48	**46**	45	340
		Gemini	107	**23**	**4**	**24**	**34**	0	192
		GPT	**102**	18	**5**	**20**	29	0	**174**
	VR	Sonnet	125	45	**36**	46	44	**51**	**347**
		Gemini	105	20	3	**24**	29	0	181
		GPT	100	**19**	4	19	27	0	169
	UT	Sonnet	**126**	**50**	32	48	44	44	344
		Gemini	110	12	**4**	20	27	**1**	174
		GPT	101	18	2	18	25	0	164
	PA	Sonnet	125	36	35	**49**	45	48	338
		Gemini	112	15	**4**	19	32	0	182
		GPT	98	16	2	18	**31**	0	165
	VR + UT + PA	Sonnet	**126**	43	26	46	45	42	328
		Gemini	**115**	21	**4**	22	30	**1**	**193**
		GPT	101	14	2	18	29	0	164

B FORMALLY VERIFIED TRANSFORMATION EXTENSION

It is possible to use similar instrumentation as described in Section 3.1 to backport constant propagation from the verified compiler CompCert's intermediate representation, although this requires more engineering effort and introduces more code that could perform semantics-breaking transformations. This can be done by instrumenting function calls with values of interest as arguments. If every callsite to that function has a constant argument after compiler passes run, the value of that variable or expression has been proven to be constant.