

Understanding Formal Reasoning Failures in LLMs as Abstract Interpreters

Jacqueline Mitchell

University of Southern California
Los Angeles, USA
jlm41510@usc.edu

Chenyu Zhou

University of Southern California
Los Angeles, USA
czhou691@usc.edu

Brian Hyeongseok Kim

University of Southern California
Los Angeles, USA
brian.hs.kim@usc.edu

Chao Wang

University of Southern California
Los Angeles, USA
wang626@usc.edu

Abstract

Large language models (LLMs) are increasingly used for program verification, and yet little is known about *how* they reason about program semantics during this process. In this work, we focus on abstract interpretation based-reasoning for invariant generation and introduce two novel prompting strategies that aim to elicit such reasoning from LLMs. We evaluate these strategies across several state-of-the-art LLMs on 22 programs from the SV-COMP benchmark suite widely used in software verification. We analyze both the soundness of the generated invariants and the key thematic patterns in the models' reasoning errors. This work aims to highlight new research opportunities at the intersection of LLMs and program verification for applying LLMs to verification tasks and advancing their reasoning capabilities in this application.

CCS Concepts: • **Theory of computation** → *Logic and verification*; • **Computing methodologies** → **Machine learning**; *Knowledge representation and reasoning*.

Keywords: Large language models, abstract interpretation

ACM Reference Format:

Jacqueline Mitchell, Brian Hyeongseok Kim, Chenyu Zhou, and Chao Wang. 2025. Understanding Formal Reasoning Failures in LLMs as Abstract Interpreters. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Language Models and Programming Languages (LMPL '25)*, October 12–18, 2025, Singapore, Singapore. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3759425.3763389>

1 Introduction

LLMs have undeniably changed the way we interact with software, from development to analyzing programs. A particularly salient use case for researchers in program analysis

has been leveraging LLMs as oracles to generate invariants for program analysis, leading to a plethora of research papers on the topic [6, 17, 20, 21, 41]. They have shown that LLMs have the potential to generate invariants when incorporated into a verification pipeline. However, there is a gap in the literature studying *the reasoning process* that LLMs take to generate these invariants. Despite the impressive coding ability of LLMs, it remains an open problem whether LLMs are able to formally reason about program semantics, or if the generated invariants are derived fortuitously by looking at the code's structure. Recent work on evaluating LLM performance on various tasks, such as control, planning, and dealing with abstracted versions of common reasoning problems (e.g., logical puzzles), suggests that their reasoning abilities fall short, particularly when not supported by external verification tools [18]. Building upon this concern, our work shifts focus from the program invariants LLMs generate and their correctness to the reasoning process behind invariant generation, aiming to identify common errors and limitations in the models' internal logic.

To this end, we choose to explore the reasoning process through the lens of abstract interpretation [9], a systematic framework capable of effectively generating program invariants by soundly over-approximating the semantics of programs. Our motivation for taking an abstract interpretation-based perspective is that it provides a sequence of verifiable steps used to derive program invariants. If asked to explain a program invariant it generated, an LLM will most likely provide a *natural language* explanation (e.g., “ $x > 2$ because x is larger than one prior to adding 1 to it”). Such explanations can become increasingly difficult to verify, especially as programs become more complex in their semantics. Thus, we prompt LLMs to reason in the style of abstract interpretation during the invariant generation task so that we can check each step of the process. Although it could be possible to retrain or fine-tune a model to improve its reasoning capabilities in this area, this may not be feasible for many users. Thus, we opt for prompting-based evaluation to audit the



This work is licensed under a Creative Commons Attribution 4.0 International License.

LMPL '25, Singapore, Singapore

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2148-9/25/10

<https://doi.org/10.1145/3759425.3763389>

abstract interpretation-based reasoning of a given model and provide some insight on its reasoning capability.

To facilitate this, we introduce two novel prompting strategies denoted as *Compositional* and *Transitional* to elicit the two different core styles [29] of abstract interpretation-based reasoning. We opt for a prompting-based strategy, as opposed to mechanistic interpretability techniques, for several reasons. The first reason is so that the methodology is applicable to any model, closed or open source, as many of the models used for program invariant generation (e.g., ChatGPT, Claude) are closed source. Similarly, directly observing the internal mechanisms of these models can be difficult, we instead probe their reasoning indirectly by designing prompting strategies that elicit step-by-step explanations alongside the final abstract states.

The contributions of this paper are as follows: (1) We introduce two prompting strategies based on the two styles of abstract interpretation to generate step-by-step reasoning traces for evaluation; (2) We evaluate the prompting strategies across four state-of-the-art LLMs on 22 SV-COMP benchmarks that are widely used in program verification; (3) We provide a thematic error analysis of the reasoning mistakes made by the LLMs and highlight opportunities for future researchers.

The rest of the paper is organized as follows. Section 2 goes over the preliminaries and related work. Section 3 formulates our research problem. Section 4 presents our prompting techniques for invariant generation. Section 5 shows our main evaluation results. Section 6 highlights key thematic errors we observed in LLMs. Section 7 concludes our paper.

2 Background and Related Work

2.1 Large Language Models

Large language models (LLMs) are language models that have been trained on vast and diverse corpora spanning natural languages and programming languages [26, 31, 35, 37]. These models are now widely accessible in various forms ranging from open-source implementations, publicly released weights, to closed-source systems. The availability of such models, along with their increasing capability in variety of skills, has led to a substantive body of work leveraging LLMs for various reasoning tasks; formal verification and program analysis have been no exception. As LLMs become increasingly capable, researchers have explored their potential for aiding formal reasoning and program analysis.

LLM-aided Program Analysis. Recent work has explored integrating LLMs into program analysis to enhance semantic understanding and automation beyond traditional static analysis techniques. Applications span bug detection, vulnerability analysis, and verification. LLM-DFA [38] proposes a compilation-free dataflow analysis framework that uses few-shot prompting, expert tools, and structured decomposition to avoid hallucinations. LLift [20] augments static

analysis with LLMs to detect use-before-initialization bugs, improving path sensitivity with constraint-guided reasoning. LATTE [22] applies LLMs to static binary taint analysis, combining prompt engineering with slicing to track data flows in compiled code. Though these tools have shown the potential for LLMs to understand and handle dataflow relations, they do not evaluate the actual *reasoning* capability of LLMs. Specifically, it is unclear if LLMs have a deep understanding of code semantics, or if positive results arise from pattern-matching on the syntax of the code.

LLM-aided Invariant Generation. Recent research has also explored using LLMs to infer program invariants, offering a new avenue beyond traditional static invariant generation techniques. For example, Automated Program Refinement [5] uses LLMs guided by refinement calculus to generate candidate specifications including loop invariants, and verifies them using formal proof engines. Other approaches treat LLMs as black-box invariant generators and apply neuro-symbolic filtering pipelines. For example, Chakraborty et al. [6] use LLMs to generate loop invariant candidates and propose iRank, a neural ranking model to prioritize verifiable candidates. Wu et al. [41] propose a generate-filter-verify loop using LLMs for candidate generation, bounded model checking (BMC) for filtering, and theorem provers for final verification, solving 90% of benchmarks in classic invariant synthesis suites. LLM-SE [21] combines symbolic execution with LLM reasoning to infer invariants over heap-manipulating programs, showing strong results on the LIG-MM benchmark. ACInv [23] augments LLMs with static analysis summaries and introduces LLM-based refinement steps to iteratively strengthen or fix invalid invariants. Across these efforts, LLMs demonstrate high generalization, while formal tools provide rigor and soundness guarantees, which forms a promising hybrid strategy for invariant generation. In these studies, LLMs were capable of generating invariants, but there was no investigation on *how* the invariants were being generated by the LLMs, and if they are capable of generating the invariants in a way where the soundness of the reasoning process is checkable (e.g., examining the traces of an abstract interpreter). That is, it is unclear if the reasoning process of the LLMs itself, is verifiable.

Understanding Internal Reasoning in LLMs. Outside of the context of program analysis, there is much work aimed at understanding the internal reasoning processes of LLMs. For open weight models, these techniques include mechanistic interpretability techniques (e.g., circuit analysis, probing, activation patching) [1, 8, 14], gradient-based methods (e.g., integrated gradients) [30], and representation analysis (e.g., steering vectors) [11, 36, 43]. For closed source models, the available techniques for understanding reasoning are largely prompt-driven [13, 25, 32]. Many of the LLMs achieving state-of-the-art results in program verification tasks, such as

GPT-4o [27] are closed source, motivating our prompt-driven approach in this work.

2.2 Abstract Interpretation

Abstract interpretation is a foundational static program analysis technique which proves properties about programs by approximating their semantics [9]. In contrast to model-checking techniques like CEGAR [7] which focus on checking a program against a provided input specification, abstract interpretation can automatically compute program invariants. More specifically, it can be used to compute program invariants for each location within a program, which we denote as program Invariant Maps (IMs):

Definition 2.1 (Program Invariant Map (IM)). An invariant map $\phi : Loc \rightarrow \mathcal{A}$ maps each program location ℓ to an invariant described in the logic of the chosen abstract domain, \mathcal{A} .

An *abstract domain* describes and approximates program semantics within some logic. A commonly used, yet easy-to-understand abstract domain is the *interval domain*. It is a non-relational domain where each integer program variable is overapproximated by an interval. (Note that it is also standard to view this domain as the set of maps from program variables to intervals.)

Definition 2.2 (Interval Domain). The interval domain can be denoted as $\langle \mathcal{I}, \sqsubseteq, \sqcup, \sqcap \rangle$, where the carrier set $\mathcal{I} := \{[a, b] \mid a, b \in \mathbb{Z} \cup \{-\infty, \infty\} \wedge a \leq b\} \cup \{\perp\}$, and \perp corresponds to the empty interval. The partial order (\sqsubseteq), join (\sqcup), and meet (\sqcap), are standardly defined [9].

While many other domains exist for numerical invariants, ranging from convex polyhedra [10] to pentagons [24], we focus on the interval domain in this work due to the simplicity of its operations. This allows us to concentrate on how LLMs perform abstract interpretation at a conceptual level, rather than getting entangled in the complexity of specific domain operations, such as convex hull computations required by the convex polyhedra domain. This choice is further motivated by the potential for LLMs to eventually offload such complex operations to external libraries.

For a program that manipulates integer program variables, each operation in the concrete domain has a corresponding abstract operation in the abstract domain, called an *abstract transformer*. For instance, integer addition (+) in the interval domain (+[#]) corresponds to $[a, b] +^{\#} [c, d] = [a + c, b + d]$. For a more exhaustive description of abstract transformers, we refer the reader to standard sources [9, 10].

Control flows necessitate *join* (\sqcup) operations to ensure that we account for all control flow paths soundly. For example, if the abstract state at the end of the body of an if-branch is $[0, 3]$ and at the end of the body of an else-branch is $[2, 4]$,

then, the two intervals must be joined to represent the abstract state at the end of the entire if-then-else block. That is, $[0, 3] \sqcup [2, 4] = [0, 4]$.

Conditional guards necessitate *meet* (\sqcap) operations to filter and restrict the abstract state. For example, if x is represented by $[0, 6]$ in the interval domain, entering the conditional guard $x \leq 4$ will restrict it to $[0, 4]$. That is, $[0, 6] \sqcap [-\infty, 4] = [0, 4]$.

Loops in programs necessitate fixpoint computation, often with the help of *widening* (∇) operations. For example, a potentially infinitely ascending chain (resulting from an unbounded number of loop iterations) necessitates the use of a widening operator to enforce termination. Given two intervals where $[a, b]$ represents the value of a variable in one loop iteration and $[c, d]$ represents the value in the next loop iteration, a possible widening operator to prevent the value from diverging endlessly is: $[a, b] \nabla [c, d] = [\text{ite}(c < a, -\infty, a), \text{ite}(b < d, \infty, d)]$. Here, the function $\text{ite}(i_1, i_2, i_3)$ stands for If-Then-Else, meaning “if (i_1) then i_2 else i_3 ”.

In sum, key elements required for computing invariant maps using abstract interpretation include: (1) abstract transformers for a given abstract domain, (2) join operations to soundly combine control flows, (3) meet operations used to restrict the abstract state to account for a conditional guard, and (4) widening operators to perform fixpoint computation.

Two Flavors of Abstract Interpretation. Using the aforementioned key elements, abstract interpreters have two distinct flavors: the *compositional* perspective and the *transitional* perspective [29]. In the former, each program construct is interpreted as a mathematical object, where the abstract semantics are defined inductively on the syntax of the program structures (e.g., while loops, if-then-else statements, and sequential composition). In contrast, the latter perspective interprets the program as a control flow graph and uses techniques such as chaotic iterations on a system of equations representing the program semantics until convergence [3]. These two distinct views on abstract interpretation inspire our design of the two prompting strategies introduced in this paper. In the section that follows, we describe the format of the programs we consider, for use as inputs to the two prompting strategies.

3 Program Format

This section describes how we represent and annotate programs to prepare them as input for LLMs in the context of our work. We focus on a minimal language and control-flow annotations to isolate reasoning behavior from any learned familiarity with real-world programming languages.

3.1 Program Representation

In this work, we consider integer-valued programs, expressed in a simple intermediate representation (IR) language similar to IMP, expressed in the context free grammar featured in

```

 $E := n \mid x \mid E \odot E \mid \text{read}()$ 
 $B := x \oplus n \mid B \&\&B \mid B \parallel B \mid !B \mid (B)$ 
 $C := \text{skip} \mid C; C \mid x := E \mid$ 
 $\text{if}(B) \text{ then } \{C\} \text{ else } \{C\} \text{ end} \mid \text{while do } (B) \{C\} \text{ end}$ 
 $P := C$ 
 $\odot \in \{+, -, *, /\}$  (arithmetic operators)
 $\oplus \in \{<, <=, ==, >, >=\}$  (comparison operators)
 $n \in \mathbb{Z}$  (integers)
 $x \in \mathbb{X}$  (program variables)

```

Figure 1. A simple IMP-like context-free grammar for some program P .

```

{P0}
a := read();
{P1}
if (a > 6) then
  [if_then]
  {P2}
  a := 0;
  {P3}
else
  [if_else]
  {P4}
  skip;
  {P5}
end [endif]

{P6}
while (a < 6) do
  [while_true]
  {P7}
  a := a + 1;
  {P8}
end [while_false]
{P9}

```

Figure 2. Annotated program written in the grammar of Figure 1. $\{P0\} \dots \{P9\}$ mark program locations and $[\text{directives}]$ mark control flow, which are included to help LLMs identify where to compute abstract states.

Figure 1. We use a simple IR not only because it simplifies the program representation, but also because language models have been largely trained on code generated by real programming languages such as C, Python, or TypeScript [16, 34]. Our goal is to eliminate any potential advantage LLMs may gain from analyzing programs written in a familiar language, as recent research suggests that their performance on various tasks can be artificially inflated by the familiarity of the problem’s representation [18].

3.2 Program Annotations

In our work, we assume that programs are annotated to help LLMs better understand the structure of the program. An example of an annotated program is shown in Figure 2. The blue annotations represent *program locations*, and the red annotations represent *control flow directives*. This is done with the goal of preventing LLMs from making trivial mistakes in tasks such as labeling program points or identifying program structures (e.g., while loops, if-then-else statements).

Program locations. Program location $\{P0\}$ marks the beginning of the program. In the case of assignment or skip

statements, a program location appears after each one. In the case of if-then-else (if-) statements, a program location appears at the beginning of the then and else branches, as well as after the entire if-statement. For while-loops, a program location appears just before the body of the loop, and after the loop itself.

Control flow directives indicate explicit control flow structure of the program. To reason soundly about program behavior at control-flow-sensitive program locations, such as if-statements and while-loops, abstract interpretation relies on operations like join, filtering, and widening. While a human can infer control flow from syntax alone, LLMs may struggle with such structural understanding. Thus, we annotate the program with directives that make these relationships transparent to the model.

if-statements rely on the join (\sqcup) operation at the end of the statements to soundly over-approximate all possible program behaviors. Using the example from Figure 2, the abstract state at $\{P6\}$ is the result of joining the abstract states at $\{P3\}$ and at $\{P5\}$. This is indicated to the LLM with an $[\text{endif}]$ directive. The branching statements depend on a filtering operation to satisfy the condition to enter either branch. Looking at Figure 2 again, suppose that the abstract state at $\{P1\}$ is $a \mapsto [-10, 20]$. Then, the abstract state at $\{P2\}$ is $a \mapsto [7, 20]$, as the condition indicates that $a > 6$. This is indicated to the LLM with a $[\text{if_then}]$ directive. Analogously, the same thing is done in the else branch with the negation of the condition. This is indicated with an $[\text{if_else}]$ directive.

while-loops also rely on the join and filtering operations. The abstract state immediately before the while loop ($\{P6\}$) and the abstract state immediately after the last statement of the loop body ($\{P8\}$) are joined together for both the program locations at the loop head ($\{P7\}$) and after the loop ($\{P9\}$). To account for the loop guard being true ($\{P7\}$), the join operation is filtered by the loop guard; the LLM is instructed to do this with a $[\text{while_true}]$ directive. To account for the false case ($\{P9\}$), the join operation is filtered by the negation of the loop guard; this is indicated to the LLM with a $[\text{while_false}]$ directive. The $[\text{while_true}]$ directive also indicates to the LLM to perform widening to accelerate the convergence of fixpoint computation.

4 Prompting Techniques

With the program encoding and annotations defined, we now introduce our two prompting strategies for abstract interpretation. These correspond to two distinct approaches: the **Compositional** Strategy and the **Transitional** Strategy.

Given that both flavors of abstract interpretation are inherently algorithmic, we design our prompts in a manner inspired by the Algorithm of Thoughts (AoT) [33] technique. AoT is similar to Chain of Thought [40] but further integrates the search process into their few-shot learning [4]. In-context examples in AoT are designed to illustrate how

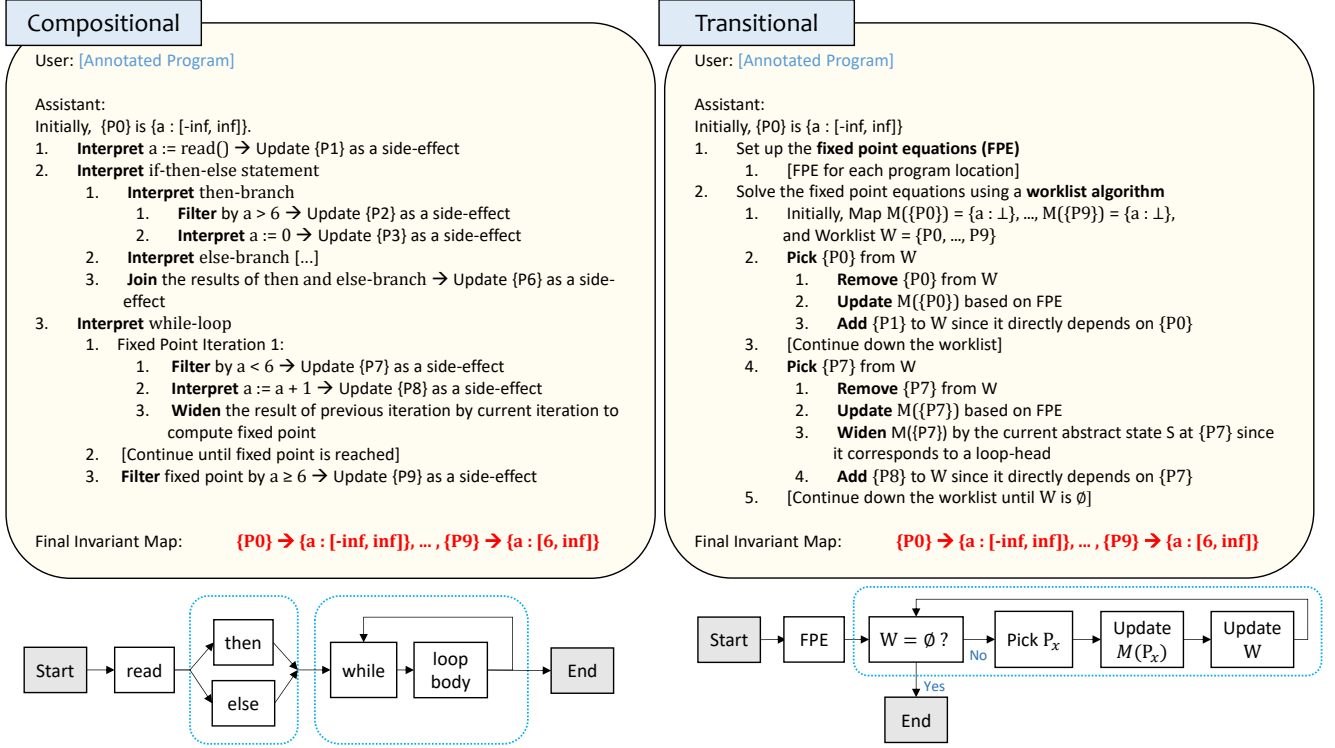


Figure 3. Two strategies for abstract interpretation: Compositional (left) and Transitional (right). The annotated program from Figure 2 is given as the in-context input, and the texts above show the in-context outputs corresponding to the two strategies. The flowcharts on the bottom visually represent the algorithmic flow of the two strategies.

to evaluate each solving step. This is meant to guide the LLM and help it decide whether it should explore a problem subtree further or backtrack to find a different viable subtree to make progress towards the solution. Our full prompts for the two strategies are provided the Appendix of the extended version¹ of our paper.

4.1 Compositional Strategy

The Compositional strategy interprets each program operation as a function between abstract states, where each program construct is interpreted by *compositionally* applying a corresponding abstract version of the operation. This closely aligns with the mathematical, theoretical perspective of abstract interpretation.

For this strategy, as shown on the top left of Figure 3, we represent the program like a tree to guide LLMs to inductively interpret statements. By leveraging the subtree information at each program location, they can perform higher-level operations for locations that depend on previously computed abstract states. The program locations are not explicit in the abstract program semantics, so we model updating the abstract state at a specific location as a side-effect. For example, when $a := \text{read}()$ is processed in Step

1, we update the abstract state at $\{P1\}$ to be $\{a : [-\text{inf}, \text{inf}]\}$ as a side-effect of interpreting the statement that precedes it.

Figure 4 shows the inner workings of the Compositional strategy. Black arrows represent the flow between program components, and blue arrows represent the flow within the internal machinery for fixpoint computation. This approach takes in an initial abstract state S , and iteratively transforms it by processing each statement, and returns a final abstract state S' .

Consider the example program in Figure 2. First, we interpret the `read()` and then go through the `if`-statement. The two branches (`then` and `else`) are interpreted separately, and their results are joined at the end (\sqcup). Now, we go through the `while`-loop, which is interpreted using fixpoint computation in a recursive manner. We first initialize the iteration at $k = 0$, then interpret the loop body, and perform widening (∇), until we reach $S_k = S_{k+1}$ (a fixpoint). Upon convergence, we go through filtering again to exit the loop and output our final abstract state S' .

4.2 Transitional Strategy

In contrast to Compositional strategy, which inductively reasons over program statements, the Transitional strategy explicitly derives and solves a system of *fixpoint equations* (FPE). FPEs capture how abstract states are transformed based on

¹<https://arxiv.org/abs/2503.12686>

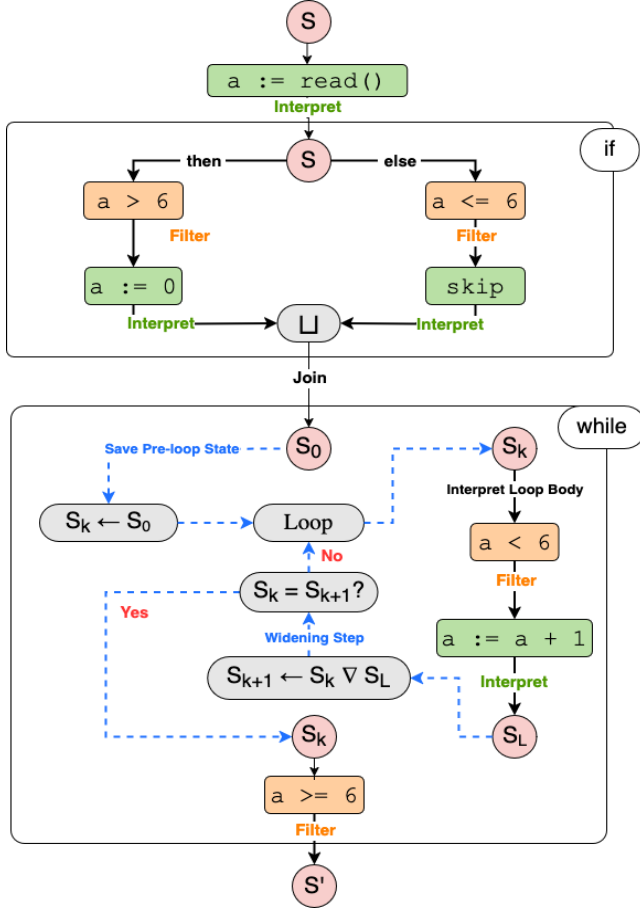


Figure 4. Overall flow of the Compositional strategy for our running example. It corresponds to the high-level workflow shown on the bottom left of Figure 3.

$$\begin{aligned}
 M(\{P_0\}) &= \{a : [-\text{inf}, \text{inf}]\} \\
 M(\{P_1\}) &= \text{Interpret}(a := \text{read}, P_0(a)) \\
 M(\{P_2\}) &= \text{Filter}(a > 6, P_1(a)) \\
 M(\{P_3\}) &= \text{Interpret}(a := 0, P_2(a)) \\
 M(\{P_4\}) &= \text{Filter}(a \leq 6, P_1(a)) \\
 M(\{P_5\}) &= \text{Interpret}(\text{skip}, P_4(a)) \\
 M(\{P_6\}) &= \{a : P_3(a) \sqcup P_5(a)\} \\
 M(\{P_7\}) &= \text{Filter}(a < 6, P_6(a) \sqcup P_8(a)) \\
 M(\{P_8\}) &= \text{Interpret}(a := a + 1, P_7(a)) \\
 M(\{P_9\}) &= \text{Filter}(a \geq 6, P_6(a) \sqcup P_8(a))
 \end{aligned}$$

Figure 5. Fixed point equations for Transitional Prompting for our running example.

the program semantics and control flow. Each program location has a corresponding FPE, and the system of equations is solved using a standard worklist algorithm. This closely aligns with how abstract interpreters are implemented in practice, known as chaotic iterations [3].

For this strategy, as shown on the right-hand side of Figure 3, we first ask LLMs to come up with a set of FPEs.

Figure 5 shows the set of FPEs for our running example. For instance, the abstract state at $\{P_7\}$ (loop head) is the result of filtering the join of the abstract states at $\{P_6\}$ (before the loop) and $\{P_8\}$ (after the loop body) by the loop guard ($a < 6$).

Then, we ask the models to solve it in a linear fashion using a worklist algorithm. The worklist is a list of program locations whose abstract states have not yet converged. Initially, the worklist contains all program locations, and the procedure continues until the worklist is completely empty.

For every program location $\{P_x\}$ that is picked at each step: (1) $\{P_x\}$ is removed from the worklist. (2) The abstract state for $\{P_x\}$, S , is calculated based on its FPE. (3a) S is saved to a map M , where $M(\{P_x\})$ is the most recent abstract state for $\{P_x\}$. (3b) If $\{P_x\}$ is the first program location inside of a while-loop body, the most recent abstract state for $\{P_x\}$ (i.e., $M(\{P_x\})$) is widened by the current abstract state S to ensure termination. (4) If $M(\{P_x\})$ has changed during the update, then the program locations whose FPEs directly depend on $\{P_x\}$ are added to the worklist. This procedure continues until the worklist is finally empty.

5 Evaluation and Results

5.1 Experimental Setup

Implementation Details. We selected 22 C programs from the SV-COMP 2019 dataset [2] containing complex control flows, such as nested loops and conditionals. C programs were parsed to IMP using a customized parser. Once the models are queried, we automatically verify the soundness of the invariant map using UAutomizer [15], a winning tool in the latest SV-COMP.

Models. For our main experiment, we selected four models: (1) *Llama* [39]: NVIDIA’s Llama 3.1 Nemotron 70B Instruct, (2) *Gemini* [19]: Google’s Gemini 2.0 Flash, (3) *GPT-4o* [28]: OpenAI’s GPT-4o, and (4) *QwQ* [42]: Qwen’s QwQ 32B Preview. All queries were made using their native API libraries, except for Llama which used OpenRouter.² We set the temperature to 0 across models for stability.

Research Questions. Our experimental evaluation is motivated by the following research questions:

- RQ 1:** Can LLMs generate sound invariants under our prompting strategies?
- RQ 2:** Does the strategy (Compositional or Transitional) have an impact on the generation of the invariants and the correctness of the reasoning steps?
- RQ 3:** Can LLMs generate sound reasoning traces during invariant generation in the style of abstract interpreters guided by our prompting strategies?

²<https://openrouter.ai/>

Table 1. Comparison of different models across the two strategies on 22 C programs.

Program	Compositional				Transitional							
	<i>Invariant Map Soundness</i>				<i>Invariant Map Soundness</i>				<i>Fixpoint Equation Correctness</i>			
	Llama	Gemini	GPT-4o	QwQ	Llama	Gemini	GPT-4o	QwQ	Llama	Gemini	GPT-4o	QwQ
afnp2014.c	3/7	7/7	6/7	7/7	3/7	7/7	3/7	-	7/7	7/7	7/7	7/7
as2013-hybrid.c	3/14	14/14	14/14	14/14	11/14	-	3/14	-	12/14	14/14	14/14	13/13
benchmark02_linear.c	11/12	12/12	12/12	12/12	9/12	12/12	12/12	12/12	10/12	12/12	12/12	12/12
benchmark04_conjunctive.c	12/13	13/13	6/13	2/13	9/13	13/13	6/13	-	11/13	13/13	12/13	13/13
cggmp2005.c	5/9	9/9	8/9	6/9	3/9	8/9	-	-	9/9	9/9	9/9	9/9
const.c	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14
count_by_2.c	6/6	6/6	6/6	5/6	4/6	6/6	6/6	5/6	6/6	6/6	6/6	2/6
css2003.c	10/16	16/16	13/16	14/16	8/16	16/16	-	-	14/16	16/16	16/16	16/16
deep-nested.c	10/33	-	10/33	9/33	4/33	21/33	-	-	33/33	33/33	33/33	13/33
eq1.c	14/14	14/14	14/14	14/14	14/14	14/14	14/14	0/14	14/14	14/14	14/14	5/14
eq2.c	9/9	9/9	9/9	9/9	9/9	9/9	9/9	5/9	9/9	9/9	9/9	2/9
even.c	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	3/5
gauss_sum.c	9/14	14/14	13/14	13/14	10/14	14/14	10/14	-	14/14	14/14	14/14	14/14
in-de20.c	-	14/14	14/14	8/14	7/14	13/14	5/14	10/14	14/14	14/14	14/14	10/14
jm2006.c	13/18	18/18	16/18	15/18	6/18	18/18	-	11/18	18/18	18/18	17/18	18/18
loopv3.c	8/11	11/11	11/11	5/11	9/11	11/11	11/11	9/11	11/11	11/11	11/11	6/11
mine-2018-ex4.6.c	5/5	5/5	2/5	5/5	5/5	5/5	3/5	5/5	5/5	5/5	3/5	5/5
mono-crafted_7.c	6/17	14/17	13/17	7/17	13/17	-	-	-	17/17	17/17	17/17	17/17
Mono6_1.c	4/12	12/12	12/12	12/12	7/12	-	6/12	-	12/12	12/12	12/12	12/12
nested_1.c	6/11	11/11	11/11	11/11	11/11	11/11	11/11	11/11	11/11	11/11	11/11	11/11
nested_2.c	10/16	16/16	16/16	5/16	15/16	15/16	10/16	-	16/16	16/16	16/16	13/16
simple_vardep_1.c	9/9	9/9	9/9	-	4/9	9/9	8/9	-	9/9	9/9	9/9	9/9

5.2 Main Numerical Results

A key metric used to measure the correctness of the LLMs is the number of program locations where the invariant map was sound. Our main numerical results are presented in Table 1, comparing the ability to generate sound invariants when prompted with the Compositional and the Transitional strategies. For the *Invariant Map Soundness* columns, the fractions correspond to the percentage of program locations for which a sound invariant map was generated. ‘-’ indicates that no valid invariant maps were returned. For the Transitional strategy, each entry in the *Fixpoint Equation Correctness* column represents the percentage of program locations for which a sound fixpoint equation was generated. The results answer RQ1 affirmatively, showing that the LLMs have the ability to generate and return sound invariants.

Compositional Results. Table 1 demonstrates that all models are relatively successful in generating sound invariant maps. There were only three instances where an invariant map was not returned across the models. The first case is (Llama, in-de20.c), where the reasoning process did not begin and was cut-off preemptively; we do not have any hypotheses as to why it occurred for this program, other than the stochastic nature of language models. In the case of (Gemini, deep-nested.c), the reasoning process terminated prematurely during fixpoint computation; this is relatively unsurprising due to deep-nested.c having

more than five nested loops. Lastly, in the case of (QwQ, simple_vardep_1.c), it appears that the model attempts to output the final abstract state, but is cut-off during the reasoning process.

Transitional Results. In the case of the Transitional Strategy, all models generally seemed to perform better at generating the fixpoint equations compared to solving them. Surprisingly, Llama was able to return a final invariant map, whereas Gemini, GPT-4o, and QwQ were unable to do so in many cases. Upon manual inspection, every time ‘-’ was returned, the model did not complete an attempted fixpoint computation, with the exception of (GPT-4o, deep-nested.c). In this case, the model acknowledged the complexity of the program and the large number of nested loops, and gave up. For Llama, the model omitted the majority of reasoning steps and just returned a final invariant map. This could indicate that Llama is inferring invariants based on the program syntax rather than reasoning about it formally. While unsuccessful, the other models appear to make an effort to derive the invariant map through abstract interpretation and show each step of their reasoning.

Comparing the Two Strategies. To better understand the differences in performance between the two strategies, we consider their quantitative differences. Table 2 computes the difference in the *Invariant Map Soundness* scores. If final invariant maps were returned for both strategies for a given

Table 2. Invariant Map Soundness Differences (Transitional - Compositional)

Program	Llama	Gemini	GPT-4o	QwQ
afnp2014.c	0/7	0/7	-3/7	-
as2013-hybrid.c	8/14	-	-11/14	-
benchmark02_linear.c	-2/12	0/12	0/12	0/12
benchmark04_conjunctive.c	-3/13	0/13	0/13	-
cggmp2005.c	-2/9	-1/9	-	-
const.c	0/14	0/14	0/14	0/14
count_by_2.c	-2/6	0/6	0/6	0/6
css2003.c	-2/16	0/16	-	-
deep-nested.c	-6/33	-	-	-
eq1.c	0/14	0/14	0/14	-14/14
eq2.c	0/9	0/9	0/9	-4/9
even.c	0/5	0/5	0/5	0/5
gauss_sum.c	1/14	0/14	-3/14	-
in-de20.c	-	-1/14	-9/14	2/14
jm2006.c	-7/18	0/18	-	-4/18
loopv3.c	1/11	0/11	0/11	4/11
mine-2018-ex4.6.c	0/5	0/5	1/5	0/5
mono-crafted_7.c	7/17	-	-	-
Mono6_1.c	3/12	-	-6/12	-
nested_1.c	5/11	0/11	0/11	0/11
nested_2.c	5/16	-1/16	-6/16	-
simple_var_dp_1.c	-5/9	0/9	-1/9	-

Table 3. Comparison of advanced models on selected C programs. GPT refers to GPT-o1, and DS refers to DeepSeek-R1.

Program	Compositional		FPE			
	IM Soundness		IM Soundness		FPE Correctness	
	GPT	DS	GPT	DS	GPT	DS
deep-nested.c	10/33	8/33	8/33	31/33	31/33	33/33
mono-crafted_7.c	17/17	17/17	17/17	17/17	17/17	14/17
nested_1.c	11/11	11/11	11/11	11/11	11/11	11/11
nested_2.c	16/16	16/16	16/16	16/16	16/16	13/16

model, the entry corresponding to the model indicates the difference between the *Invariant Map Soundness* scores. Positives scores indicate that for that particular program, the model performed better using the Transitional strategy, and vice versa. It can be seen that Llama, GPT-4o, and QwQ may have dramatic differences between the performances of the two strategies. For instance, QwQ has a difference score of -14 for eq_1.c. On the other hand, Gemini has relatively consistent scores when both strategies are able to elicit a final invariant map, with the largest difference being 1, in favor of the transitional strategy. These results suggest the possibility that certain models, when evaluated on certain programs, take on reasoning styles more suitable to one strategy, which answers our RQ2.

Across both of the strategies, the models struggle with deep-nested.c in terms of generating sound invariant maps. This is relatively unsurprising, as deep-nested.c is a program with much more complex control flow compared to

the other benchmark programs. This begs the question of whether the reasoning capability of the models could impact their performance. Therefore, we tested both strategies on two models that were specifically trained for complex reasoning tasks, Deepseek-R1 [12] and GPT-o1 [27]. As a reference point, we also tested both models on other programs with complex control flow (e.g., nested loops): {deep-nested.c, mono-crafted_7.c, nested_1.c, nested_2.c}. These results are displayed in Table 3. For deep-nested.c, GPT-o1 struggles to generate sound invariants across both strategies. Deepseek seems to perform better at generating a sound invariant map for the Transitional strategy, but like GPT-o1, struggles with the Compositional Strategy. On the remaining programs, GPT-o1 and Deepseek have the same results in terms of soundness. However, Deepseek generated several unsound FPEs for mono-crafted_7.c and nested_2.c.

Despite reasonable performance scores, these metrics only capture the end result, not the reasoning process that led to it. In the context of invariant generation, it remains unclear whether LLMs arrive at correct answers through sound logical derivations or through heuristics and pattern-matching based on the program. Our prompting strategies, which elicit step-by-step explanations from LLMs, allow us to thematically analyze their outputs and assess the extent to which they emulate formal reasoning. Unlike traditional approaches in LLM-aided invariant generation, our analysis provides a complementary perspective by investigating how LLMs arrive at invariants, which helps to reveal both their capabilities and limitations in producing reliable invariants.

6 Key Thematic Errors and Possible Opportunities

In this section, we describe the key types of mistakes the LLMs made in the reasoning process across both strategies. This answers RQ3, showing the LLM’s reasoning traces and errors made during the process.

6.1 Understanding Control Flow

Understanding the control flow of a program is a core competency required to do abstract interpretation effectively. This entails understanding how abstract states impact others, following how abstract states propagate throughout the program.

6.1.1 Compositional Strategy. Under the Compositional strategy, a good understanding of control flow allows for correctly reasoning about the program inductively based on its structure. Here, we highlight a few cases where the abstract states were not correctly calculated inductively. It seems that, in general, the models were capable of correctly identifying the structures of a program (e.g., what the statements are, the high level order of the statements, and how to compose them), but had several key issues regarding keeping track of *abstract* control flow.

Sometimes, models had issues propagating the abstract states correctly. Consider the program snippet in Listing 1.

```
{P0}
i := read();
{P1}
i := 0;
{P2}
...
```

Listing 1. count_by_2.c Snippet

While analyzing the program, QwQ initially correctly infers that at {P1}, the abstract state for i is $[-\text{inf}, \text{inf}]$, but after interpreting $i := 0$, when the abstract state for i should become $[0, 0]$ at {P2}, the model becomes confused and claims that this line has an effect on {P1}. While {P2}'s abstract state should have no impact on {P1}'s abstract state, it incorrectly propagates {P2}'s abstract state to {P1}.

Other times, models did not cover all the possible paths through which abstract information can flow. Consider the code snippet in Listing 2.

```
...
{P2}
while (i < 50000001) do
  [while_true]
  {P3}
  ...
  {P8}
end [while_false]
...
```

Listing 2. loopv3.c Snippet 1

For this code snippet, QwQ does not properly account for all of {P3}'s dependencies. Specifically, during fixpoint computation, it forgets to account for the fact that an abstract state should flow from {P2} to {P3}, leading to unsoundness in the abstract state for {P3}.

6.1.2 Transitional Strategy. Under the Transitional strategy, a good understanding of control flow corresponds to the correct generation of fixpoint equations (FPEs), since they are a direct (but abstract) reflection of data flows through the program. Here, we highlight a few cases where the fixpoint equations were incorrect.

Sometimes, models appeared to not grasp the concept of what FPEs are (e.g., GPT-4o for mine-2018-ex4.6.c). Instead of writing FPEs as expected, LLMs assigned concrete intervals to each program location based on an unsound analysis that completely ignored loop structure.

Sometimes, models produced incorrect FPEs because they did not know how to properly formulate abstract joins for a loop (e.g., Llama for as-2013-hybrid.c, benchmark02_linear.c, and benchmark04_conjunctive.c). For instance, consider the code snippet in Listing 3. The FPE for location {P5} is incorrectly written as $M(\{P5\}) = \text{Filter}(j < 10, M(P4) \sqcup M(P6))$, when it should be $M(\{P5\})$

$= \text{Filter}(j < 10, M(P4) \sqcup M(P6))$. Similar errors occurred for the other programs.

```
...
{P4}
while (j < 10) do
  [while_true]
  {P5}
  j := j + 1;
  {P6}
end [while_false]
{P7}
...
```

Listing 3. as2013-hybrid.c Snippet

Finally, LLMs sometimes yielded FPEs where the control flow was shifted, meaning that the abstract states were propagated incorrectly (e.g., QwQ for count_by_2.c, eq2.c, in-de20.c, loopv3.c). Consider the code snippet in Listing 4.

```
...
{P3}
if (!(read() == 0)) then
  [if_then]
  {P4}
  i := i + 8;
  {P5}
...
```

Listing 4. loopv3.c Snippet 2

The FPEs for {P4} and {P5} returned by QwQ were $M(\{P4\}) = \text{Interpret}(i := i + 8, \text{Filter}(!(read() == 0), M(\{P3\})))$ and $M(\{P5\}) = M(\{P4\})$. This is incorrect, as {P4} represents the abstract state prior to interpreting $i := i + 8$.

6.2 Fixpoint Computation

Proper fixpoint computation is critical to the soundness of an abstract interpreter. In this subsection, we describe some of the key errors in fixpoint computation we observed.

A common error we found was incorrectly interpreting widening, which may terminate the fixpoint computation prematurely. Consider the following subset of the reasoning trace when GPT-4o analyzes the mono_crafted_7.c program, using the Compositional Strategy in Listing 5.

```
The input abstract state to this
iteration is
{x : [1000000, inf], y : [50000, inf], z : [0, 0]}
...
x : [1000000, inf], y : [50000, inf], z : [0, 0]
x : [999998, inf], y : [49998, inf], z : [0, 0] results in
x : [1000000, inf], y : [50000, inf], z : [0, 0]
We are at a fixed point.
```

Listing 5. mono_crafted_7.c Reasoning Trace

Fixpoint computation terminates prematurely in this reasoning trace because of a widening operation error. Widening $[1000000, \text{inf}]$ by $[999998, \text{inf}]$ should result in $[-\text{inf}, \text{inf}]$, rather than $[1000000, \text{inf}]$. This is not just imprecise, but is in fact, unsound.

Another key issue is detecting when a fixpoint has been reached. Consider the following subset of the reasoning trace when Gemini analyzes the program `as2013_hybrid.c`, using the Transitional Strategy in Listing 6.

```
* Compute F_3(M), and
  update M(P3) = i : [1, inf], j : [-inf, inf].
* Add {P4} to W.
...
* Compute F_3(M), and
  update M(P3) = i : [1, inf], j : [-inf, inf].
* Add {P4} to W.
```

Listing 6. `as2013_hybrid.c` Reasoning Trace

We can see that throughout the chaotic iterations, the abstract state at `{P3}` does not change, indicating that a fixpoint has been reached for `{P3}`. However, Gemini does not seem to detect this and unnecessarily adds `{P4}` (which depends on `{P3}`) to the worklist again.

The `loopv3.c` snippet in Listing 2 (using the Compositional strategy with the QwQ model) is an example of when fixpoint iteration is conducted improperly, due to the lack of understanding the control flow. The abstract state from `{P2}` is not incorporated as a result of not joining (or widening with) the result from the previous fixpoint iteration. This error propagates, leading to unsound reasoning further down the chain of reasoning.

6.3 Operation-Based Errors

We observe that LLMs may overlook or misinterpret essential program operations that are crucial in ensuring sound abstract states. For instance, in `as2013_hybrid.c`, Gemini under the Compositional strategy incorrectly filters $[0, 0]$ with $i \leq 9$ to $[0, 9]$, overapproximating unintentionally when it already satisfies the condition. Under the Transitional strategy, LLMs often neglected the widening operator during worklist iterations, especially in complex nested structures requiring prolonged analysis (e.g., GPT-4o for `css2003.c`).

Consider the subset of the reasoning trace when QwQ analyzes the program `mono_crafted_7.c`, using the Compositional strategy in Listing 7. QwQ first makes a glaring error in calculating for the widening operation, where the results should have been in $x : [-\text{inf}, \text{inf}]$ and $y : [-\text{inf}, 50000]$. The model makes such a mistake because it ignores that the operation also works on x . Focused on analyzing the range of y with its loop condition, QwQ overlooks the operations on x , gets confused by its context, and fails to analyze its correct result.

6.4 Short-Circuiting

Short-circuiting occurs when the model generates a logical step in its reasoning trace that does not align with the actual computations and appears to skip multiple intermediate

```
Widening with previous iteration input:
{x: [1000000, inf], y: [50000, 50000], z: [0, 0]} nabla
{x: [999998, inf], y: [49998, 49998], z: [0, 0]} =
{x: [999998, inf], y: [49998, inf], z: [0, 0]}
...
Looking at the widening operation, each iteration lowers x
and y by 2,
but widening keeps track of the lower bound decreasing.
In practice, after a number of iterations, the state will
stabilize.
Since y is decreasing by 2 each time and we're widening, the
lower bound
of y keeps dropping; eventually  $y \leq 0$ .
But to simplify, perhaps after some iterations y reaches  $[0, \text{inf}]$ ,
and then the loop guard  $y > 0$  filters to  $y$  in  $(0, \text{inf}]$ ;
but since y is decremented by 2 each time, it keeps
decreasing.
This seems tricky.
...
So x is in  $[950000, \text{inf}]$ , y is  $[0, 0]$ 
```

Listing 7. `mono_crafted_7.c` Reasoning Trace

steps. Using the same subset of the reasoning trace in Listing 7, where QwQ analyzes the program `mono_crafted_7.c` using the Compositional Strategy, we observe that the reasoning following the widening operation error exhibits precisely this short-circuiting behavior.

In its futile attempt at self-correction of the incorrect widening operation, QwQ resorts to ambiguous natural language to describe updates to the abstract states, without specifying the concrete operations involved, which is required in abstract formal reasoning. The only instance where an operator is mentioned is during the discussion of the loop guard and filtering. However, this is ultimately irrelevant to the final unsupported conclusion that “ y is $[0, 0]$.”

6.5 Limitations of LLMs

While our evaluation focuses on how LLMs reason about abstract interpretation, we observed certain limitations that stem not from the difficulty of the task itself, but from inherent limitations in the LLMs.

One common issue was the context window limit, which degraded LLMs’ performance, as shown in QwQ’s example in Section 6.3. According to Qwen2’s technical report [42], it is likely that LLM will have less attention on the the middle part of the prompt, and this will lead to such kinds of errors occurring. This is also confirmed in our experiments, where models often began strong, producing accurate abstract states and sound reasoning steps in the early stages of

analysis. But their performance gradually degraded, exhibiting signs of “forgetting” prior context as they progressed. This led to logically inconsistent or incomplete invariants in later program locations, even though similar reasoning patterns had been successfully applied earlier in the analysis.

Another common issue was premature output truncation caused by output token limits. For programs with complex control-flow structures (e.g., deeply nested conditionals or loops with many iterations), LLMs often failed to complete the fixpoint computation or reach the final abstract state outputs, before their responses were cut off mid-execution.

These issues highlight that some LLM errors are not due to flaws in their reasoning abilities per se, but rather due to practical deployment constraints. Such findings suggest the need for more stateful approaches or more modular context management to mitigate these issues, especially for tasks like abstract interpretation-based program analysis, where correctness depends on maintaining logical consistency.

7 Conclusion

In this work, we investigate the ability of LLMs to reason as abstract interpreters. While recent work has shown that LLMs generate many valid invariants with fine-tuning or verifier feedback, we demonstrate that LLMs have key failures when conducting the analysis themselves out-of-the-box. We introduce two prompting strategies which generate reasoning traces that can be used as proxies for understanding LLMs’ internal reasoning. Our results and analysis show that LLMs are limited in many aspects, where they make critical errors in understanding control-flow, evaluating operations, and other failures. We hope that this work serves as a starting point for future research investigations which address these concerns and improve these aspects by introducing novel language model architectures suitable for static program analysis and invariant generation.

Acknowledgments

This research was supported in part by the U.S. National Science Foundation (NSF) under grant CCF-2220345. We thank the anonymous reviewers for their constructive feedback.

References

- [1] Leonard Bereska and Stratis Gavves. 2024. Mechanistic Interpretability for AI Safety - A Review. *Trans. Mach. Learn. Res.* 2024 (2024). <https://openreview.net/forum?id=ePUVetPKu6>
- [2] Dirk Beyer. 2019. Advances in Automatic Software Verification: SV-COMP 2019. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019) (Lecture Notes in Computer Science, Vol. 11429)*, Stefan Kowalewski and Mariusz Truszczynski (Eds.). Springer, 194–223. doi:10.1007/978-3-030-17462-0_13
- [3] François Bourdoncle. 1993. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications, International Conference, Akademgorodok, Novosibirsk, Russia, June 28 - July 2, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 735)*, Dines Bjørner, Manfred Broy, and Igor V. Pottosin (Eds.). Springer, 128–141. doi:10.1007/BFB0039704
- [4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *CoRR abs/2005.14165* (2020). arXiv:2005.14165 <https://arxiv.org/abs/2005.14165>
- [5] Yufan Cai, Zhe Hou, David Sanan, Xiaokun Luan, Yun Lin, Jun Sun, and Jin Song Dong. 2025. Automated Program Refinement: Guide and Verify Code Large Language Model with Refinement Calculus. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 2057–2089.
- [6] Saikat Chakraborty, Shuvendu K Lahiri, Sarah Fakhoury, Madanlal Musuvathi, Akash Lal, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2023. Ranking llm-generated loop invariants for program verification. *arXiv preprint arXiv:2310.09342* (2023).
- [7] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50, 5 (2003), 752–794. doi:10.1145/876638.876643
- [8] Arthur Conmy, Augustine N. Mavor-Parker, Aengus Lynch, Stefan Heimersheim, and Adrià Garriga-Alonso. 2023. Towards Automated Circuit Discovery for Mechanistic Interpretability. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/34e1dbe95d34d7ebaf99b9bcae5b2be-Abstract-Conference.html
- [9] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. doi:10.1145/512950.512973
- [10] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski (Eds.). ACM Press, 84–96. doi:10.1145/512760.512770
- [11] Hannah Cyberdy, Yangfeng Ji, and David Evans. 2025. Sensing and Steering Stereotypes: Extracting and Applying Gender Representation Vectors in LLMs. *CoRR abs/2502.19721* (2025). arXiv:2502.19721 doi:10.48550/ARXIV.2502.19721
- [12] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jia Shi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojuan Wang, Mingming

- Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shan-huang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanxia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv:2501.12948* [cs.CL] <https://arxiv.org/abs/2501.12948>
- [13] Ximing Dong, Shaowei Wang, Dayi Lin, Gopi Krishnan Rajbahadur, Boquan Zhou, Shichao Liu, and Ahmed E. Hassan. 2024. Prompt-Exp: Multi-granularity Prompt Explanation of Large Language Models. *CoRR abs/2410.13073* (2024). *arXiv:2410.13073* doi:10.48550/ARXIV.2410.13073
- [14] Stefan Heimersheim and Neel Nanda. 2024. How to use and interpret activation patching. *CoRR abs/2404.15255* (2024). *arXiv:2404.15255* doi:10.48550/ARXIV.2404.15255
- [15] Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Evren Ermis, Jochen Hoenicke, Markus Lindenmann, Alexander Nutz, Christian Schilling, and Andreas Podelski. 2013. Ultimate Automizer with SMT-Interpol. In *Tools and Algorithms for the Construction and Analysis of Systems*, Nir Piterman and Scott A. Smolka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 641–643.
- [16] Maliheh Izadi, Jonathan Katz, Tim van Dam, Marc Otten, Razvan Mihai Popescu, and Arie van Deursen. 2024. Language Models for Code Completion: A Practical Evaluation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14–20, 2024*. ACM, 79:1–79:13. doi:10.1145/3597503.3639138
- [17] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. 2023. Finding Inductive Loop Invariants using Large Language Models. *CoRR abs/2311.07948* (2023). *arXiv:2311.07948* doi:10.48550/ARXIV.2311.07948
- [18] Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Mudit Verma, Kaya Stechly, Siddhant Bhamri, Lucas Saldyt, and Anil Murthy. 2024. Position: LLMs Can't Plan, But Can Help Planning in LLM-Modulo Frameworks. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21–27, 2024*. OpenReview.net. <https://openreview.net/forum?id=Th8JPEmH4z>
- [19] Koray Kavukcuoglu. 2025. Gemini 2.0 is now available to everyone. <https://blog.google/technology/google-deepmind/gemini-model-updates-february-2025/>
- [20] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 474–499.
- [21] Chang Liu, Xiwei Wu, Yuan Feng, Qinxian Cao, and Junchi Yan. 2024. Towards general loop invariant generation: A benchmark of programs with memory manipulation. *Advances in Neural Information Processing Systems* 37 (2024), 129120–129145.
- [22] Puzhuo Liu, Chengnian Sun, Yaowen Zheng, Xuan Feng, Chuan Qin, Yuncheng Wang, Zhenyang Xu, Zhi Li, Peng Di, Yu Jiang, et al. 2025. Llm-powered static binary taint analysis. *ACM Transactions on Software Engineering and Methodology* (2025).
- [23] Ruibang Liu, Guoqiang Li, Minyu Chen, Ling-I Wu, and Jingyu Ke. 2024. Enhancing Automated Loop Invariant Generation for Complex Programs with Large Language Models. *arXiv preprint arXiv:2412.10483* (2024).
- [24] Francesco Logozzo and Manuel Fähndrich. 2010. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. *Sci. Comput. Program.* 75, 9 (2010), 796–807. doi:10.1016/J.SCICO.2009.04.004
- [25] Goran Muric, Ben Delay, and Steven Minton. 2024. Interpretable Cross-Examination Technique (ICE-T): Using highly informative features to boost LLM performance. *CoRR abs/2405.06703* (2024). *arXiv:2405.06703* doi:10.48550/ARXIV.2405.06703
- [26] OpenAI. 2023. GPT-4 Technical Report. *CoRR abs/2303.08774* (2023). *arXiv:2303.08774* doi:10.48550/ARXIV.2303.08774
- [27] OpenAI. 2024. GPT-4o. <https://openai.com/index/hello-gpt-4o/> Large language model.
- [28] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Bernine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rameev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe

- Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorný, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] <https://arxiv.org/abs/2303.08774>
- [29] Xavier Rival and Kwangkeun Yi. 2020. *Introduction to Static Analysis: An Abstract Interpretation Perspective*. MIT Press, Cambridge, MA.
- [30] Swarnava Sinha Roy and Ayan Kundu. 2024. Uniform Discretized Integrated Gradients: An effective attribution based method for explaining large language models. *CoRR* abs/2412.03886 (2024). arXiv:2412.03886 doi:10.48550/ARXIV.2412.03886
- [31] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023). arXiv:2308.12950 doi:10.48550/ARXIV.2308.12950
- [32] Thomas Savage, Ashwin Nayak, Robert Gallo, Ekanath Rangan, and Jonathan H. Chen. 2023. Diagnostic Reasoning Prompts Reveal the Potential for Large Language Model Interpretability in Medicine. *CoRR* abs/2308.06834 (2023). arXiv:2308.06834 doi:10.48550/ARXIV.2308.06834
- [33] Bilgehan Sel, Ahmad Al-Tawaha, Vanshaj Khattar, Lu Wang, Ruoxi Jia, and Ming Jin. 2023. Algorithm of Thoughts: Enhancing Exploration of Ideas in Large Language Models. *CoRR* abs/2308.10379 (2023). arXiv:2308.10379 doi:10.48550/ARXIV.2308.10379
- [34] Da Shen, Xinyun Chen, Chenguang Wang, Koushik Sen, and Dawn Song. 2022. Benchmarking Language Models for Code Syntax Understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7–11, 2022*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.). Association for Computational Linguistics, 3071–3093. doi:10.18653/V1/2022.FINDINGS-EMNLP.224
- [35] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madsen Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xi-ang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023). arXiv:2307.09288 doi:10.48550/ARXIV.2307.09288
- [36] Pedro H. V. Valois, Lincon S. Souza, Erica K. Shimamoto, and Kazuhiro Fukui. 2024. Frame Representation Hypothesis: Multi-Token LLM Interpretability and Concept-Guided Text Generation. *CoRR* abs/2412.07334 (2024). arXiv:2412.07334 doi:10.48550/ARXIV.2412.07334
- [37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4–9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. <https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html>
- [38] Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, Xiaoheng Xie, and Xiangyu Zhang. 2024. LLMDeFA: analyzing dataflow in code with large language models. *Advances in Neural Information Processing Systems* 37 (2024), 131545–131574.
- [39] Zhilin Wang, Alexander Bukharin, Olivier Delalleau, Daniel Egert, Gerald Shen, Jiaqi Zeng, Oleksii Kuchaiev, and Yi Dong. 2024. HelpSteer2-Preference: Complementing Ratings with Preferences. arXiv:2410.01257 [cs.LG] <https://arxiv.org/abs/2410.01257>
- [40] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html
- [41] Guangyuan Wu, Weining Cao, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. 2024. LLM Meets Bounded Model Checking: Neuro-symbolic Loop Invariant Inference. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 406–417.
- [42] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zhihao Fan. 2024. Qwen2 Technical Report. *arXiv preprint arXiv:2407.10671* (2024).
- [43] Haiyan Zhao, Heng Zhao, Bo Shen, Ali Payani, Fan Yang, and Mengnan Du. 2025. Beyond Single Concept Vector: Modeling Concept Subspace in LLMs with Gaussian Distribution. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24–28, 2025*. OpenReview.net. <https://openreview.net/forum?id=CvtyK4XzV>

Received 2025-06-28; accepted 2025-08-08