Formal Methods in Computer-Aided Design 2024

# Leveraging LLMs for Program Verification

Adharsh Kamath*, Nausheen Mohammed*, Aditya Senthilnathan†, Saikat Chakraborty‡, Pantazis Deligiannis*, Shuvendu K. Lahiri‡, Akash Lal*, Aseem Rastogi*, Subhajit Roy§, Rahul Sharma*
* Microsoft Research, Bangalore, India
† Cornell University, Ithaca, USA
‡ Microsoft Research, Redmond, USA
§ Indian Institute of Technology, Kanpur, India

***Abstract*—We investigate code reasoning skills of Large Language Models (LLMs) in the context of formal program verification. Specifically, we look at the problem of inferring loop invariants as well as ranking functions for proving safety properties and loop termination, respectively. We demonstrate how emergent capabilities of LLMs can be exploited through a combination of prompting techniques as well as by using them in conjunction with symbolic algorithms. We curate and contribute a dataset of verification problems inspired by past work. We perform a rigorous evaluation on this dataset to establish that LLMs have the potential of improving state-of-the-art in program verification.**

## I. Introduction

Formal verification seeks to establish a proof of correctness of a program with respect to a given property. Broadly speaking, this involves *proof construction* (e.g., finding loop invariants), and *proof checking* (e.g., establishing their inductiveness). While proof checking has benefited from mechanical automation enabled by SMT solvers, proof construction still requires *ingenuity* and has been harder to automate.

The *guess-and-check* methodology seeks to reduce burden on the proof construction by allowing it to *guess* a proof that potentially may have mistakes. The soundness comes solely from proof checking. This methodology has allowed for Machine-learning (ML) based techniques to enter program verification. While it is hard for ML techniques to guarantee soundness, it can still be a source of good "guesses" on why a given program is correct, and better guesses lead to faster verification. Work in this space includes generating data from program executions and guess invariants through classical learning techniques [1], [2], active learning over decision trees [3], continuous logic networks [4], [5], as well as training neural networks to directly predict invariants from program text [6], [7]. The trend of training models for individual tasks, thus requiring independent datasets, is changing with the advent of Large Language Models (LLMs) and this forms the inspiration for our paper.

Latest foundational models such as GPT-4 [8], PaLM-2 [9], Llama-2 [10] have been trained on vasts amount of data, and have shown remarkable ability in solving a diverse set of tasks. One can supply a set of instructions in natural language to guide the model towards a certain task of interest [11]. LLMs are already aiding many software developers in writing code [12], [13]. We study the use of these foundational models for constructing proofs that can be discharged by a formal proof checker, following the guess-and-check methodology.

We study two different verification tasks, one on safety verification and another on proving program termination. Safety verification requires finding inductive invariants for loops as well as pre-post conditions for procedures, so that the given assertions in a program can be proved safe. In termination, the goal is to find a ranking function, as well as supporting invariants, to prove termination of loops. We curate a dataset of programs in the C language for these tasks and build an LLM-based toolchain, called LOOPY, for proof generation. The proofs are discharged by an off-the-shelf formal checker; in our implementation, we use Frama-C [14] because it directly supports C programs with invariant annotations.

LOOPY is based on two key aspects that help make effective use of LLM capabilities. The first is *prompt engineering* that encodes a set of instructions to describe the different tasks to an LLM. For instance, the prompt for program termination includes a definition of ranking functions in natural language. Ranking functions come in various forms, such as *lexicographic ranking functions* and *multi-phase ranking functions*; we demonstrate that LLMs are capable of producing such ranking functions when prompted with their definitions. We also introduce the concept of *nudging* where additional generic instructions are added in natural language to help the LLM generate the required proof artifact. For instance, the LLM is "encouraged" to use implications for dealing with conditional code, inferring bounds of loop variables, or producing invariants on unmodified parts of an array.

The second key aspect is the interplay between an LLM and a symbolic (formal) tool. We find that LLMs are better at generating ingredients of an inductive invariant than they are at generating the whole invariant. Consequently, LOOPY uses the HOUDINI algorithm [15] to weed out incorrect guesses and converge to a correct inductive subset from the set of LLM-generated guesses. HOUDINI only uses a linear number of calls to the checker (linear in the number of candidate invariants).

*Contributions:* We have curated a dataset of C programs for multiple different program verification tasks.[1] We also build and evaluate a tool called LOOPY for effectively leveraging LLM capabilities on these tasks. We present an evaluation

[1] https://github.com/microsoft/loop-invariant-gen-experiments

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_16

with multiple LLMS: GPT-4 [8], GPT-3.5 [16], and Code Llama [10], and compare the performance of LOOPY against a state-of-the-art symbolic baseline. Our results establish that LLMs have the potential of improving state-of-the-art in program verification. LOOPY is able to out-perform existing symbolic tools on several benchmarks.

## II. VERIFICATION TASKS AND DATASETS

We define a verification task as a C program along with a property of interest that must be established for the program. Our choice of the C programming language is based on the availability of the benchmarks as well as a formal checker (Frama-C [14]). Frama-C defines a language called ACSL [17] for writing annotations (assertions, invariants, ranking functions, etc.) as comments in C programs. We consider two kinds of verification tasks: Safety verification and Termination checking.

*a) Safety verification:* In this category of benchmarks, each of the C programs have embedded assertions. The goal is to come up with ACSL annotations that help Frama-C prove those assertions. We obtained these benchmarks from multiple sources, including Code2Inv [7], Accelerating Invariant Generation [18], Data-Driven CHC solver [19], Fluid Updates [20], Diffy [21], as well as the SV-COMP repository [22].

We perform basic filtering on these benchmarks. We discard programs that are known to be incorrect (i.e., the assertion does not hold). We also remove programs that are greater than 500 lines of code. This allows us to fit the entire program inside a single LLM query, helping us to focus on the code reasoning capabilities of LLMs. We then create three exclusive datasets, based on certain program features, as described below.

The first dataset consists of 469 programs that use only scalar types (either signed or unsigned) with a single main procedure with one loop. This category of benchmarks exercises basic mathematical reasoning, without bringing in concerns of modeling pointers, heap semantics, or quantified invariants. The second dataset consists of 31 programs with recursion, only scalar types, and no loops. These benchmarks have minimum 1, maximum 4, and average 1.4 non-main methods. The third dataset consists of 169 programs with a single method and at least one array or pointer. We are restricted by limitations of Frama-C to deal with such programs because it currently lacks support for dynamic memory allocation. We manually remove memory allocation from programs where it is not important. These programs have minimum 1, maximum 13, and an average of 4.4 loops per program.

*b) Termination checking:* The goal here is to infer a ranking function (also called a loop variant) for a loop that proves its termination. A ranking function is an integer-valued expression defined over program variables that is bounded below by 0 and strictly decreases in each iteration. We collect benchmarks from The Termination Competition [23] and from Shi et. al. [24]. We filter these programs, retaining programs that each consist of only scalar variables, single method, no assertions, and a single loop (that we believe is terminating). This set consists of 281 benchmarks.

We remove any comments in all the programs in our datasets because they could potentially provide hints to the LLMs.

*c) Summary of the results:* We show examples of programs in each of our datasets in Figure 1. All the comments in these examples are LOOPY-generated output (massaged slightly for conciseness), which in each case suffices to complete the corresponding verification task. Figure 2 summarizes LOOPY's performance across these datasets. The column Total is the total number of benchmarks in the dataset. The column "Vanilla LLMs" refers to the number of benchmarks that GPT-4 is able to solve with a basic prompt (and multiple completions, we detail these concepts in later sections). These numbers show the raw LLM performance on the corresponding tasks. Column LOOPY is the number of benchmarks solved by our LLM-based toolchain; a significant increase over raw LLM performance. The next two columns provide a comparison against a symbolic baseline. We show the performance of Ultimate Automizer [25], one of the tools that routinely wins medals in SV-COMP. The last column is the number of benchmarks that could be solved by either LOOPY or Ultimate, showcasing the potential of LLMs in improving the state-of-the-art.

## III. INDUCTIVE LOOP INVARIANT INFERENCE

This section considers the problem of inferring inductive loop invariants. Guided by empirical observations, we propose three techniques that can be used to augment LLMs for such tasks: (a) providing domain-specific instructions to the LLMs, (b) filtering incorrect LLM outputs with an adaptation of the Houdini algorithm, and (c) using LLMs to repair the incorrect invariants. We find that these three techniques significantly improve the ability of LLMs to infer loop invariants. While we use GPT-4 for most of our experiments, we also present a comparison with GPT-3.5-Turbo and CodeLlama-34b-Instruct [10] (an open source LLM) in III-G.

### A. The problem

Consider a C-like imperative language. Let $S$ denote statements written in this language. Hoare triples $\{P\}\ \ S\ \ \{Q\}$, where $P$ and $Q$ are logical propositions over program variables in some underlying logic, are assertions interpreted as: if $P$ holds before executing the statement $S$, then $Q$ holds after its completion [26] ($S$ may not terminate though). Inductive loop invariants, $I$, are logical summaries for loop statements in the language, used to prove the corresponding Hoare triples. For example, for $while$, following is an inference rule to derive a Hoare triple ($B$ denotes boolean expressions):

$$\frac{P \Rightarrow I \qquad \{I \wedge B\}\ \ S\ \ \{I\} \qquad I \wedge \neg B \Rightarrow Q}{\{P\}\ \ while\ B\ do\ S\ \ \{Q\}}$$

Inductive loop invariant $I$ is a logical proposition that (a) holds at the beginning of the loop ($P \Rightarrow I$), (b) is preserved by the loop body ($\{I \wedge B\}\ \ S\ \ \{I\}$), and (c) implies the postcondition of the loop ($I \wedge \neg B \Rightarrow Q$). We assume that the qualifier "inductive" is implicit when we talk about invariants in this paper, and hence drop it for convenience.

```c
void main() {
  int k = 100, b = 0;
  int i = 0, j = 0, n;
  // @invariant 0 <= n <= 2k
  // @invariant n % 2 == b
  // @invariant b == 0 ==> i == j
  // @invariant b == 1 ==> i == j - 1
  for (n = 0; n < 2*k; n++) {
    if (b) i++; else j++;
    b = !b;
  }
  assert (i == j);
}
```

(a) Loop invariant

```c
void main() {
  int x = unknown(), y = unknown(), z = unknown();
  // multi-phase ranking function
  // [z; y; x]
  while (x >= 0) {
    if (unknown()) x = x + y;
    else x = x + z;
    y = y + z; z = z - 1;
  }
}
```

(c) ranking function for loop termination

```c
void main()
{
  int N = unknown ();
  if (N <= 0) return;
  int i, a[N];
  // @invariant 0 <= i <= N;
  // @invariant \forall k. 0 <= k < i ==> a[k] == 3;
  // @assigns i, a[0..N-1];
  for (i = 0; i < N; i++) { a[i] = 3; }

  // @invariant 0 <= i <= N;
  // @invariant \forall j. i <= j < N ==>
  //                       a[j] == \at(a[j], Entry);
  // @invariant \forall j. 0 <= j < i ==>
  //                       a[j] == 1 || a[j] == 2;
  // @assigns i, a[0..N-1];
  for (i = 0; i < N; i++) {
    if (N % (i + 1) == 0) a[i] = a[i] - 1;
    else a[i] = a[i] - 2;
  }
  for (i = 0; i < N; i++) { assert (a[i] <= 2); }
}
```

(b) Loop invariant over arrays

Fig. 1: Example programs along with LOOPY-generated annotations. Each program verifies with Frama-C.

| Benchmark | Features | Total | Vanilla LLMs | LOOPY | Ultimate | LOOPY + Ultimate |
|---|---|---|---|---|---|---|
| Scalar loops | (1, 1) | 469 | 237 (51%) | 398 (85%) | 430 (92%) | 461 (98%) |
| Array loops | (1, $\geq$1) | 169 | 60 (36%) | 127 (75%) | 12 (7%) | 128 (75%) |
| Recursion | ($\geq$1, 0) | 31 | 14 (45%) | 16 (52%) | 20 (65%) | 23 (74%) |
| Termination | (1, 1) | 281 | 49 (17%) | 181 (64%) | 236 (84%) | 255 (91%) |

Fig. 2: Summary of LOOPY results. Features are (#methods, #loops).

Automatically synthesizing loop invariants is one of the classical problems in program verification. Our goal is to use and evaluate LLMs for this task. Interactions with LLMs happen via *prompts*. Prompts are textual instructions for LLMs to perform a task. LLMs respond to prompts with textual answers. LLMs may also be instructed to generate multiple responses (commonly called as *completions*) for one prompt. For our tasks, we design *prompt templates* that contain common instructions for LLMs to infer loop invariants, and template holes for the exact program. For each benchmark, we instantiate the template hole with the benchmark program.

### B. Basic algorithm

Figure 3 shows our basic algorithm for loop invariant inference using LLMs; in the subsequent sections, we will refine it with additional techniques. The algorithm takes as input a program $\mathcal{P}$, a prompt template $\mathcal{M}$, and the number of completions $\mathcal{N}_c$. It either returns Success $\mathcal{I}$, where $\mathcal{I}$ is a set of propositions such that $\bigwedge_{i \in \mathcal{I}} i$ is a loop invariant strong enough to prove the assertions in $\mathcal{P}$, or it returns Failure when it cannot infer a sufficiently strong loop invariant.

1: **procedure** INFERENCE($\mathcal{P}, \mathcal{M}, \mathcal{N}_c$)
2: **while** $0 < \mathcal{N}_c$ **do**
3: $\mathcal{I} \leftarrow \mathcal{L}(\mathcal{M}[\mathcal{P}])$
4: $b, \_, \_ \leftarrow \mathcal{O}(\mathcal{P}, \mathcal{I})$
5: **if** $b$ **then return** Success $\mathcal{I}$
6: **else** $\mathcal{N}_c \leftarrow \mathcal{N}_c - 1$
7: **return** Failure

Fig. 3: Algorithm for invariant inference using LLMs

To check the output of LLMs, the algorithm relies on an Oracle $\mathcal{O}$. The oracle takes as input the program $\mathcal{P}$ and the set $\mathcal{I}$. It returns as output a triple $(b, \mathcal{I}_S, \mathcal{I}_{NI})$, where:

1) $b$ is a boolean value, true if $\mathcal{P}$ verifies with the loop invariant $\bigwedge_{i \in \mathcal{I}} i$, false otherwise.
2) $\mathcal{I}_S \subseteq \mathcal{I}$ is the set of invariants that exhibit parsing errors; when $b$ is true, $\mathcal{I}_S$ is empty.
3) $\mathcal{I}_{NI} \subseteq \mathcal{I}$ is the set of invariants for which the oracle cannot establish the inductiveness property. This can happen for two reasons: (a) when the invariant does not

hold at the beginning of the loop, or (b) when the loop body does not maintain the invariant. When $b$ is true or when $\mathcal{I}_S$ is non-empty, $\mathcal{I}_{NI}$ is empty.

We assume that the oracle is sound, i.e., if it returns true, the assertions in $\mathcal{P}$ can be proven in the underlying logic using the loop invariant $\bigwedge_{i \in \mathcal{I}} i$. The basic algorithm does not use $\mathcal{I}_S$ and $\mathcal{I}_{NI}$.

Given these notations, the algorithm is a straightforward loop that prompts the LLM with the prompt template instantiated with the program ($\mathcal{M}[\mathcal{P}]$) until it either finds a loop invariant or runs out of the number of completions to try. Soundness of the algorithm follows from the soundness of the oracle $\mathcal{O}$.

For our experiments, we instantiate $\mathcal{O}$ with Frama-C, configured to use only the WP plugin for verifying ACSL annotations. Under these settings, Frama-C does not attempt to infer the invariants by itself; it is focused on verifying the correctness of supplied loop invariants as well as the assertions in the input program. Additionally, we configure the WP plugin to use Z3 [27], Alt-Ergo [28], and CVC4 [29] as the external provers, with a timeout of 3 seconds.

### C. Basic prompt

We evaluate the algorithm with a basic prompt template, $\mathcal{M}_0$, shown below. Here, the {{ code }} section is the template hole for the program. Notably, the template does not explain to LLM what loop invariants are or provide any detailed instructions for inferring them. It does, however, provide instructions to format the output in the ACSL syntax; this helps in automating the checking process.

```
Consider the following C program:
{{ code }}
Output the loop invariants for the loop in the program above. Output
all the loop invariants in one code block. E.g.,
/*@
    loop invariant i1;
    loop invariant i2;
*/
```

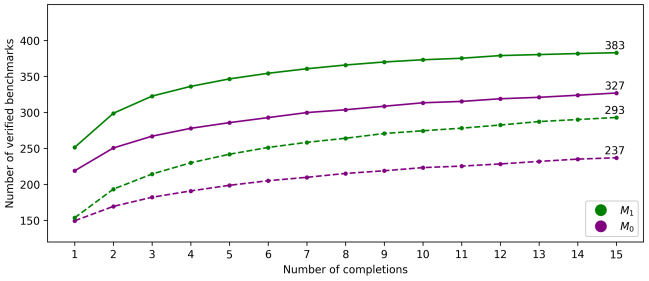


Fig. 4: Performance of LOOPY with and without Houdini (solid and dashed lines, resp.) for the two prompt templates

Figure 4 (dashed line for prompt $\mathcal{M}_0$) shows experimental results with GPT-4. It plots the success rate (number of verified benchmarks) as the number of completions is varied from 1 to 15. We account for the stochastic nature of LLMs in the standard way using the pass@k metric [30]: we first generate the maximum number of completions (15) and compute its success rate. Then, the success rate for $k < 15$ completions is obtained as the expectation over a random sample of size $k$ out of the 15 completions. As the figure shows, with 15 completions, GPT-4 is able to solve ∼50% (237/469) of the benchmarks. Multiple completions help, though there are diminishing returns after ∼8 completions. The experiment shows that even without any specialized instructions or techniques, GPT-4 is able to solve a non-trivial fraction of the benchmarks.

### D. Prompt with domain-specific instructions

On manual inspection of the failure cases with $\mathcal{M}_0$, we observe that the model makes several low-level mistakes, such as using variables or functions that are not defined, using conditional statements in the invariants, etc. We also notice that the model misses certain common invariant expressions, such as bounding a variable with its minimum and maximum values or relations between variables themselves. Consider the loop invariant example in Figure 1(a). With $\mathcal{M}_0$, the LLM only outputs $0 \leq$ n$\leq 2*$k and n % 2 == b. While both these are valid invariants, they are not sufficient to prove the assertion, as they don't capture the relationship between i and j, which is conditional on the value of b.

To account for such failures, we design a prompt template $\mathcal{M}_1$ that provides more detailed instructions to the LLM. Specifically, it explains to the LLM, in natural language, what a loop invariant is, and provides some heuristics about how to come up with a loop invariant. The full prompt $\mathcal{M}_1$ is given below.

You are a helpful AI software assistant that reasons about how code behaves. Given a program, you can find loop invariants, which can then be used to verify some property in the program. Frama-C is a software verification tool for C programs. The input to Frama-C is a C program file with ACSL (ANSI/ISO C Specification Language) annotations. For the given program, find the necessary loop invariants of the while loop to help Frama-C verify the post-condition.
Instructions:

- Make a note of the pre-conditions or variable assignments in the program.
- Analyze the loop body and make a note of the loop condition.
- Output loop invariants that are true
  (i) before the loop execution,
  (ii) in every iteration of the loop and
  (iii) after the loop termination,
  such that the loop invariants imply the post condition.
- If a loop invariant is a conjunction, split it into its parts.
- Output all the loop invariants in one code block.
  For example:
  ```
  /*@
  loop invariant i1;
  loop invariant i2;
  */
  ```

Rules: **Do not use variables or functions that are not declared in the program.** **Do not make any assumptions about functions whose definitions are not given.** **All undefined variables contain garbage values. Do not use variables that have garbage values.** **Do not use keywords that are not supported in ACSL annotations for loops.** **Variables that are not explicitly initialized, could have garbage values. Do not make any assumptions about such values.** **Do not use the \at(x,

Pre) notation for any variable x.** **Do not use non-deterministic function calls.**

Consider the following C program:
```
{{ code }}
```
You are allowed to use implication to take care of the conditional nature of the code. Use implication (==>) instead of using if-then. For all variables, add conjunctions that bound the maximum and minimum values that they can take, if such bounds exist. If a variable is always equal to or smaller or larger than another variable, add a conjunction for their relation. If the assertion is guarded by a condition, use the guard condition in an implication. If certain variables are non-deterministic at the beginning or end of the loop, use an implication to make the invariant trivially true at that location. Output the loop invariants for the loop in the program above. Let's think step by step.

To evaluate $\mathcal{M}_1$, we repeat the same experiment as before with $\mathcal{M}_1$ and GPT-4; see dashed line for $\mathcal{M}_1$ in Figure 4. GPT-4 is able to solve 293 benchmarks, 23% more than $\mathcal{M}_0$, demonstrating the effectiveness of detailed prompt instructions. For the example in Figure 1(a), with $\mathcal{M}_1$, LLM outputs the final two invariants b $== 0 \Rightarrow$ i $==$ j and b $== 1 \Rightarrow$ i $==$ j $-$ 1, which are sufficient to verify the example.

### *E. Pruning incorrect invariants with Houdini*

In many cases, we observe that the LLM output contains the required invariants but they are mixed with other output expressions that are either syntactically invalid or are not valid loop invariants. Further, the required invariants may be spread across multiple completions. In both these cases, the basic algorithm fails.

The program below is one such example. A candidate loop invariant for this is $0 \leq$ x $<$ n. We observe that in most completions, LLM outputs $0 \leq$ x and x $\leq$ n as invariants, while there are some completions in which it outputs input $== 0 \Rightarrow$ x $<$ n and in some other it outputs input $\neq 0 \Rightarrow$ x $<$ n. All the completions together have the correct components, $0 \leq$ x, input $== 0 \Rightarrow$ x $<$ n, and input $\neq 0 \Rightarrow$ x $<$ n, but no single completion is correct in itself.

```
int n = unknown(); if (n <= 0) return;
int x = 0, input = unknown();
while (1) {
  if (input) { x = x + 1; if (x >= n) break; }
  input = unknown();
}
assert (x == n);
```

To handle such cases, we augment our basic algorithm with Houdini [15] to efficiently prune the incorrect outputs; Figure 5 shows the new algorithm. The algorithm maintains a set $\mathcal{I}_u$ of all the invariants output by the LLM across all the completions. If none of the completions succeed, the algorithm invokes the Houdini procedure with $\mathcal{I}_u$, and returns the result of the Houdini procedure.

The Houdini procedure tries to find a subset of $\mathcal{I}_u$ that is inductive and is sufficient to verify $\mathcal{P}$. While the number of possible subsets of $\mathcal{I}_u$ is exponential in the size of $\mathcal{I}_u$, it turns out that one can do this check with only a linear number of calls to the oracle (linear in the size of $\mathcal{I}_u$) [15]. Figure 6 shows

```
1: procedure INFERENCE(P, M, N_c)
2:     I_u ← ∅
3:     while N_c > 0 do
4:         I ← L(M[P])
5:         b, _, _ ← O(P, I)
6:         if b then return Success I
7:         else
8:             I_u ← I_u ∪ I
9:             N_c ← N_c − 1
10:    return HOUDINI(P, I_u)
```

Fig. 5: Inference algorithm with Houdini

```
1: procedure HOUDINI(P, I)
2:     while I ≠ ∅ do
3:         b, I_S, I_NI ← O(P, I)
4:         if b then return Success I
5:         if I_S ≠ ∅ then I ← I − I_S
6:         else
7:             if I_NI = ∅ then return Failure
8:             else I ← I − I_L
9:     return Failure
10:
```

Fig. 6: Houdini algorithm

an adaptation of the Houdini algorithm to our setting. The algorithm takes as input a program $\mathcal{P}$ and a set of candidate invariants $\mathcal{I}$. It either returns Success $\mathcal{I}_I$, where $\mathcal{I}_I \subseteq \mathcal{I}$ and $\bigwedge_{i \in \mathcal{I}_I} i$ is an inductive loop invariant strong enough to verify $\mathcal{P}$, or it returns a Failure if it cannot find such a subset. The algorithm repeatedly queries the oracle with its current set of candidate invariants $\mathcal{I}$. Recall that the output of oracle is a triple $(b, \mathcal{I}_S, \mathcal{I}_{NI})$, where $b$ is a boolean, $\mathcal{I}_S$ is the set of syntactically invalid candidates, and $\mathcal{I}_{NI}$ is the set of non-inductive candidates.

If the oracle returns true, the algorithm returns with Success $\mathcal{I}$. Otherwise, it removes one or more candidates from the set $\mathcal{I}$ and repeats the process. This pruning happens in one of two ways. If there are some candidate invariants that are syntactically invalid, they are pruned away. If there are no syntax errors, and the set $\mathcal{I}_{NI}$ is empty, the procedure returns Failure: this indicates the case when the current set $\mathcal{I}$ is a valid inductive invariant, but still not sufficient (i.e., strong enough) to verify the program. Otherwise the candidates in $\mathcal{I}_{NI}$ are pruned away and the loop repeats. The soundness of the Houdini algorithm follows directly from the soundness of the oracle. Houdini returns Success $\mathcal{I}$ only when the oracle verifies $\mathcal{P}$ with $\mathcal{I}$. Furthermore, the algorithm makes a linear number of calls to the oracle (linear in the size of candidate invariant set $\mathcal{I}$). In addition to soundness, Houdini guarantees to find the largest inductive subset of invariants [15].

*Evaluation:* Figure 4 (solid lines) show the impact of Houdini with both the prompt templates $\mathcal{M}_0$ and $\mathcal{M}_1$. With Houdini, the success rate for $k < 15$ is computed as an average over randomly sampling $k$ out of the 15 completions, taking their union, and running Houdini.

Houdini has a significant positive impact. With 15 completions and the $\mathcal{M}_1$ prompt, the use of Houdini increases the success rate by **30.7**% (from 293 to 383 solved benchmarks). As before, the prompt $\mathcal{M}_1$ does better than $\mathcal{M}_0$ (383 to

```
1: procedure INFERENCE(P, M, N_c, N_r)
2:     ...
3:     r ← HOUDINI(P, I_u)
4:     if r = Success _ then return r
5:     else return REPAIR(P, I_u, N_r)
```

Fig. 7: Inference algorithm with Repair

```
1: procedure REPAIR(P, I, N_r)
2:     _, I_S, I_NI ← O(P, I)
3:     while N_r > 0 do
4:         I ← L(M_r[P, I, I_S, I_NI])
5:         b, I_S, I_NI ← O(P, I)
6:         if b then return Success I
7:         else
8:             r ← HOUDINI(P, I)
9:             if r = Success _ then return r
10:            else N_r ← N_r − 1
11:    return Failure
```

Fig. 8: Repair algorithm

327 solved benchmarks). The results suggest that augmenting LLMs with symbolic techniques such as Houdini can increase the effectiveness of LLMs in solving such problems.

With the candidate invariants generated using prompt $\mathcal{M}_1$, we invoke Frama-C with timeout values higher than 3 seconds. We observed that the number of benchmarks verified by Frama-C remained the same with a timeout of 5 seconds and even 10 seconds.

### F. Using LLMs to repair incorrect invariants

We explore using LLMs to repair the incorrect invariants, guided by the error messages produced by the oracle. This is motivated by an observation that in some cases minor changes to the LLM output can give us the correct invariants.

We parameterize our inference algorithm with another parameter $\mathcal{N}_r$ that denotes the maximum number of repair retries that the algorithm can make (Figure 7). Instead of returning the result of Houdini, as in Figure 5, the revised algorithm checks whether Houdini succeeds. If it does, the algorithm returns the result. If Houdini fails, it invokes a repair procedure.

The Repair algorithm, shown in Figure 8, takes as input the program $\mathcal{P}$, the set of all the LLM output invariants $\mathcal{I}_u$ across all completions, and the number of repair retries $\mathcal{N}_r$. It uses a specialized prompt template $\mathcal{M}_r$, templated over $\mathcal{P}$, a set of invariants $\mathcal{I}$, and $\mathcal{I}_S$ and $\mathcal{I}_{NI}$, incorrect subsets of $\mathcal{I}$ as returned by the oracle. The prompt template provides instructions to the LLM to repair the incorrect invariants. We show a snippet of $\mathcal{M}_r$ below:

```
Frama–C returns the following message:
{{ error }}

If the error message indicates a syntax error in the loop annotation,
fix the line with the syntax error. To fix the non–inductive
invariants, try the following:

If an invariant is preserved but not established, add a clause to the
invariant to make it established (a clause that makes the invariant
hold before the loop begins).

If an invariant is established but not preserved, add a clause to the
invariant to make it preserved (a clause that makes the invariant
hold after the loop ends, assuming that it holds before the loop
begins).

If an invariant is neither established nor preserved, remove it or
replace it with a different inductive invariant. If none of the
above is possible, add a new loop invariant to strengthen the
existing invariants.
```

The repair algorithm first invokes the oracle to get the errors $\mathcal{I}_S$ and $\mathcal{I}_{NI}$. It then prompts the LLM using $\mathcal{M}_r$, instantiated with $\mathcal{P}, \mathcal{I}, \mathcal{I}_S$, and $\mathcal{I}_{NI}$, to repair $\mathcal{I}$; the output of LLM is a new set of candidate invariants. The algorithm then uses the oracle and the Houdini procedure to find a sufficiently strong inductive set of invariants within the new set. The process repeats until either the algorithm succeeds in finding such a set or it runs out of the retries budget $\mathcal{N}_r$. The soundness of Repair follows from the soundness of the oracle and Houdini—it returns Success only when either the oracle or Houdini returns Success.

*Evaluation:* To evaluate our inference algorithm with Repair, we need to provide the $\mathcal{N}_r$ parameter. To keep the LLM budget the same as before, we make $\mathcal{N}_c + \mathcal{N}_r = 15$ so that the use of Repair does not increase the number of LLM queries. Observing that without Repair, the number of verified benchmarks starts to plateau at around 8 completions (Figure 4), we set $\mathcal{N}_c = 8$ and $\mathcal{N}_r = 7$.

With the repair procedure, LOOPY is able to verify 15 more benchmarks than before, bringing the number of benchmarks verified to 398/469. An example where repair helps is as shown. Before repair, the candidate invariants are `y == 10 - (x - 1)` and `y < 10`, both of which capture the behavior of `x` and `y` after the first iteration of the loop. The invariants, however, do not hold at the beginning of the first iteration when `y` is unconstrained. With Repair algorithm, the invariants are repaired to `x == 1 ∨ y == 10 - (x - 1)` and `x == 1 ∨ y < 10`, allowing `y` to take any value before the first iteration. With these invariants, the program verifies.

```
void main()
{
  int x = 1;
  int y;
  while (x <= 10) {
    y = 10 - x;
    x = x + 1;
  }
  assert (y < 10);
}
```

### G. Comparing different LLMs

To compare different LLMs, we evalute our inference algorithm, with and without Houdini, on two other models: GPT-3.5-Turbo and CodeLlama-34b-Instruct [10]. For this experiment, we fix the number of completions to 15 and use the prompt template $\mathcal{M}_1$. Figure 9 shows the results, we also plot the previously shown results for GPT-4 for comparison.

GPT-4 shows superior performance compared to the other models, although GPT-3.5-Turbo is a close second with 370

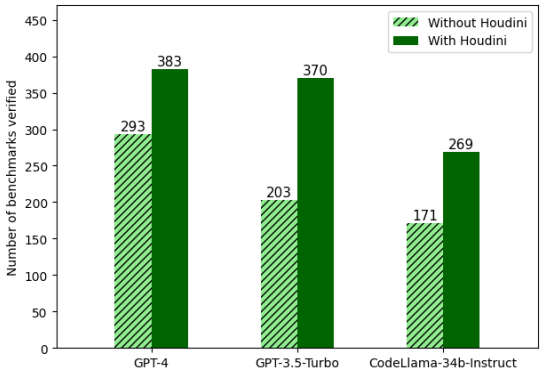


Fig. 9: LLMs comparison (with $\mathcal{M}_1$ and $\mathcal{N}_c = 15$)

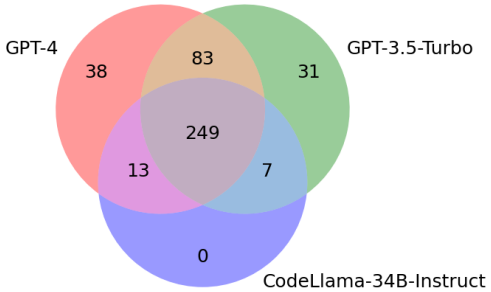


Fig. 10: Verified benchmarks

solved benchmarks when using Houdini. Interestingly, using Houdini helps other models "catch up" with GPT-4 by significantly increasing their success rates.

Figure 10 shows the intersection among the benchmarks verified using the different LLMs. GPT-4 has the most number of exclusively-solved benchmarks (38). GPT-3.5-Turbo is able to solve 31 benchmarks that GPT-4 could not. This experiment suggests that using multiple LLMs can help solve more benchmarks.

### *H. Qualitative analysis*

We manually analyzed the failure cases for LOOPY and observed that for 10 failures, LOOPY is able to produce a correct and sufficient loop invariant, but Frama-C fails to verify the program. This implies a success rate of **408/469** for LLMs, augmented with our techniques. Among the successfully verified benchmarks, LOOPY generated, on average, 4.2 invariants per benchmark. Each of these invariants had, on average, 1.8 variables, and 2.1 operators (boolean, relational, and arithmetic), indicating that a fair number of invariants were non-trivial.

*Analysis of failed benchmarks:* We analyzed the benchmarks that LOOPY was not able to solve. For the 10 benchmarks for which LOOPY produces the right invariant but Frama-C fails to verify the program, we believe that it should be possible to strengthen Frama-C (e.g., one failure was due to missing axiomatization of integer mod operation). For the remaining 61 benchmarks, we manually came up with *an*

```
void main() {
  int x = 0, y = 0, flag = 0;
  while (flag < 1) {
    if (y < 0) flag = 1;
    if (flag < 1) x = x + 1;
    if (x < 50) y = y + 1;
    else y = y - 1;
  }
  assert(y == -2 && x == 99)
}
```

```
int main() {
  int x = 0, y = 0, N;
  if (N < 0)
    return 1;

  while (1) {
    if (x <= N) y++;
    else if (x >= N + 1)
      y--;
    else return 1;

    if (y < 0) break;
    x++;
  }

  if (N >= 0)
    if (y == -1)
      if (x >= 2 * N + 3)
        assert(\false);

  return 1;
}
```

Fig. 11: Example (a) where LOOPY fails (left), and (b) where Ultimate fails but LOOPY succeeds (right).

invariant that makes the program verify with Frama-C. Based on these ground truth invariants, we do a subjective classification of the failures into 4 categories. The classification is not indicative of features that are beyond LLMs today; there are also benchmarks in each of these categories that LLMs are able to solve.

The first category of failures are benchmarks that require disjunctions in the invariant. These benchmarks can be described as either having loops with multiple *phases* (i.e., as the loop iterates, the code path taken inside the loop changes several times, based on some flags or other branches), or assertions that depend on whether the loop is executed at all (needing a disjunct to account for the case when the loop is not entered at all). The program shown in Figure 11(a) has a multi-phase loop. It starts with x and y both at 0; then both increase by 1 in each iteration until x reaches 50, after which x continues to increase by 1 while y starts to decrease. Describing these "phases" requires one clause for each phase, connected by disjunction. We classified 44/61 failures in this category.

The second category of failures are benchmarks whose ground truth invariants requires a clause with at least three variables. An example of an invariant in this category is the following: $(0 < p) \wedge (2*q + r \leq w) \wedge (p == r + 2*i)$. 5/61 failures fall in this category.

The third category of failures are benchmarks where more precise constraints were required, compared to what was generated by our algorithm. For instance, for one of the benchmarks, the algorithm inferred the invariant $(k == x + y + z) \wedge (x \leq y) \wedge (y == z)$, which turned out to be an inductive invariant, but insufficient to prove the assertion. Changing the second clause to x == y would make it work. There are 9/61 such failures. The fourth category, containing 3/61 failures,

requires reasoning about floating-point arithmetic. It was hard to us, even manually, to come up with their ground-truth invariants.

*Symbolic baseline:* We compare the performance of our LLM-based inference algorithm the Ultimate tool [25] on the 469 benchmarks. Ultimate has higher success rate than LOOPY with **430/469** benchmarks solved. However, we find that there are 31 benchmarks that Ultimate does not solve, but LOOPY can solve. There are 63 benchmarks that Ultimate solves but LOOPY does not. Ultimate and LOOPY combined can solve **461/469** benchmarks, hinting that combining symbolic tools with LLM-based techniques can improve the existing state-of-the-art. Figure 11(b) shows an example from these 31 benchmarks. The assertion in the program can be verified with the loop invariant (x $\leq$ N+1 $\Rightarrow$ y == x) $\wedge$ (x $>$ N+1 $\Rightarrow$ y == 2*(N+1) – x), which LOOPY infers but Ultimate does not.

To compare the run times of LOOPY and Ultimate, we randomly selected 50 benchmarks from our dataset and measured the average time taken by each tool to verify a benchmark. In the case of LOOPY, we generate 15 completions and check all of them with Frama-C. If all completions fail, then we run the Houdini loop. The average run time of Ultimate was 23.11s, and that of LOOPY was 186.05s (including the LLM inference time which was 119.86s, using an unoptimized LLM-inference stack). Although optimizations to the LOOPY implementation and the LLM inference stack could improve the run time, it is not in the scope of this work and we leave it as interesting avenues for future work.

### I. Loop invariant inference for programs with arrays

We next evaluate our loop invariant inference algorithm on 169 benchmarks that use arrays. We use the algorithm shown in Figure 5, i.e. with Houdini but no repair, with 8 completions ($\mathcal{N}_c = 8$). With the prompt template $\mathcal{M}_0$, our algorithm is able to solve **60/169** benchmarks, while using the template $\mathcal{M}_1$ increases this number to **102/169** benchmarks.

On manual inspection of the failures, we find that LLM sometimes misses clauses that are common in the invariants for loops that manipulate arrays. To help LLMs in such cases, we add some array-specific instructions to the prompt. Some sample instructions are shown below:

> For all the values and array ranges that do not change in the loop, add an invariant equating them to their value before the loop. Add a loop assigns clause listing all array ranges and variables assigned for every loop. When a loop assigns an array,
> –Invariants must specify state of all array elements after every iteration of the loop, even if they have not changed yet.
> –For the range of elements yet to be assigned by the loop, just equate them to their value before the loop. For nested loops, use the invariants of the inner loop as hints for the outer loop.

The instructions are mostly about capturing the precise state of the arrays in the invariants. With these instructions the number of solved benchmarks increases to **127/169**. Interestingly, these 127 are not a superset of the 102 solved with just $\mathcal{M}_1$ alone; Figure 12 shows a Venn diagram of the three sets: benchmarks solved with $\mathcal{M}_0$, with $\mathcal{M}_1$, and with $\mathcal{M}_1$ and instructions. Consider the array loop invariant example from

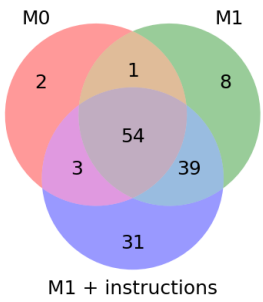


Fig. 12: Performance of Loopy instantiated with different prompts for arrays, $\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_1$ + Instructions

Figure 1. The first for loop initializes the array a (of length N) s.t. a[i] = 3. The second loop, depending on whether N % (i + 1) is 0, either assigns a[i] = a[i] – 1 or a[i] = a[i] – 2. The third loop asserts that a[i] $\leq$ 2. With $\mathcal{M}_1$, the LLM fails to come up with the invariant \forall j. i <= j < N ==> a[j] == \at(a[j], Entry) for the second loop, stating that the values of array elements a[i] onwards are what they were at the beginning of the loop. This invariant is crucial for verifying the assert, and so, the program fails to verify. However, once we instruct the LLM to capture all array elements in the invariant, LLM outputs this clause, and the program verifies.

For the 42 failure cases, there are 9 benchmarks where LOOPY infers the correct and sufficient loop invariants, but Frama-C fails to verify the program. Further, there are some programs where the loop invariants inferred by LOOPY are close to the required invariants. In one of the benchmarks, for example, the loops iterate from 1 to N, but LOOPY infers invariants where the index variable ranges from 0 to N. We believe such cases may be handled by using Repair (Figure 8); we leave this for future work.

## IV. PROGRAM TERMINATION

*The problem:* A ranking function is used to prove termination of a loop. In its simplest form, a ranking function $V$ is an expression involving the variables used in the loop with the following two properties: (a) at the beginning of every loop iteration, the value of $V$ is $\geq 0$, and (b) the value of $V$ strictly decreases with each loop iteration. Thus, the value of $V$ at the beginning of an iteration provides an upper bound on the number of remaining loop iterations. A ranking function is sometimes also called a variant. There is a rich literature on algorithms for synthesizing ranking functions using abstract intepretation [31] constraint solving, model checking [32], [33] and more recently using custom trained neural networks [34]. We evaluate LLM capabilities to add to this body of work.

Beyond a simple expression, there are other common forms of ranking functions, lexicographic ranking functions and multi-phase ranking functions [35], [32]. A lexicographic ranking function is a ordered list $[V_i]$, where (a) at the beginning of a loop iteration, for all $i$, the value of $V_i$ is $\geq 0$, and (b) in every loop iteration, there exists $j$ s.t. the value of $V_j$ strictly decreases and $\forall k.\ k < j$, the value of $V_k$ remains the

| Prompts used | No. of benchmarks verified |
|---|---|
| $M_2$ | 133 |
| $M_2$, $M_3$ | 170 |
| $M_2$, $M_3$, $M_4$ | 181 |

Fig. 13: Results for ranking function inference

```
1: procedure VariantInference(P, M, N_V, N_I)
2:     while 0 < N_V do
3:         V ← L(M[P])
4:         I ← ∅
5:         n ← N_I
6:         while 0 < n do
7:             I ← I ∪ L(M_I[V, P])
8:             n ← n − 1
9:         r ← Houdini(P, I)
10:        if r = Failure then N_V ← N_V − 1
11:        else
12:            Success I ← r
13:            r ← O_T(P, V, I)
14:            if r then return Success (V, I)
15:            else N_V ← N_V − 1
16:    return Failure
```

Fig. 14: Ranking function inference

same. A multi-phase ranking function [36] is a special case of lexicographic ranking function. It is an ordered list $[V_i]$, where when the loop execution starts, first $V_1$ decreases until it becomes non-positive, then $V_2$ decreases until it becomes non-positive, and so on. There are other forms of ranking functions and well-founded relations [37], [38], [39], [40], [36], [35], but we restrict focus to only the ones described above.

To prove that a ranking function is valid for a loop, one might need additional loop invariants to establish key properties about the loop. The problem, therefore, is to infer both a ranking function as well as the supporting invariants that are needed to prove its correctness.

We assume access to an oracle $\mathcal{O}_\mathcal{T}$ that takes as input a program $\mathcal{P}$ (with a single loop), a (candidate) ranking function $\mathcal{V}$, and an inductive loop invariant $\mathcal{I}$. It returns a boolean value where true implies that $\mathcal{V}$ can be proven to be a valid ranking function using $\mathcal{I}$. Frama-C provides such an interface, and we use it as our oracle.

*Ranking function inference algorithm:* Figure 14 shows our ranking function inference algorithm. It takes as input a program $\mathcal{P}$ with a single loop, a prompt template $\mathcal{M}$, and two number of completions parameters $\mathcal{N}_V$ and $\mathcal{N}_I$. It returns either Success $(\mathcal{V}, \mathcal{I})$, where $\mathcal{V}$ and $\mathcal{I}$ are ranking function and inductive loop invariant for the loop in $\mathcal{P}$ respectively, or Failure otherwise.

The algorithm instantiates the prompt template $\mathcal{M}$ with $\mathcal{P}$, and queries the LLM to infer a candidate ranking function $\mathcal{V}$. To be able to invoke the oracle to check the LLM output, we need an inductive loop invariant as well. The algorithm infers it using the techniques developed in the last section. Specifically, it instantiates a prompt template $\mathcal{M}_I$ with the candidate ranking function $\mathcal{V}$ and the program $\mathcal{P}$, and prompts the LLM. Template $\mathcal{M}_I$ instructs the LLM to infer an inductive loop invariant required for proving a given ranking function. The algorithm collects the LLM output loop invariants for $\mathcal{N}_I$ completions, and invokes the Houdini algorithm (Figure 6). If Houdini succeeds, the algorithm invokes the oracle $\mathcal{O}_\mathcal{T}$ with $\mathcal{V}$ and $\mathcal{I}$ (the output of Houdini). If the oracle returns true, the algorithm returns Success $(\mathcal{V}, \mathcal{I})$. Whereas if either Houdini or the oracle fails, the algorithm repeats until it succeeds or it exhausts the maximum number of retries $\mathcal{N}_V$. The soundness of the algorithm follows from the soundness of Houdini (Figure 6) and the oracle.

*Evaluation:* We evaluate our ranking function inference algorithm on 281 benchmarks, with one method and one loop each. We set the parameters $\mathcal{N}_V$ and $\mathcal{N}_I$ to 5 each, and as mentioned before, use Frama-C as the oracle. We show that by adding increasingly domain-specific instructions to the prompt template $\mathcal{M}$, we can solve more benchmarks.

The prompt template $\mathcal{M}_I$ contains instructions for the LLM to infer inductive loop invariants for a given ranking function. The prompt mentions that the LLM should infer an invariant that implies the ranking function decreases with every iteration. It also contains the loop invariant instructions similar to those in the $M_1$ prompt from the previous section. The full prompt is available in the public repository[1].

We first evaluate a basic prompt template for inferring the loop ranking function. The prompt template explains to the LLM what a ranking function is, but it does not instruct the LLM to infer a specific kind of ranking function (lexicographic or multi-phase, for instance). The complete prompt text is available in the public repository[1]. The result of using this prompt template is shown in Table 13 as the prompt $M_2$. As can be seen, with this prompt, our algorithm is able to solve **133/281** benchmarks.

On a closer inspection of the failures, we find that while the algorithm could solve simple cases of ranking functions (e.g., when it is a single expression), it did not do so well on benchmarks that require lexicographic or multi-phase ranking functions. We, next, add instructions for inferring lexicographic ranking functions.

> A lexicographic ranking function is a sequence of expressions with the property that each expression must be positive for the loop to execute. For example, if (e1, e2, e3) is a lexicographic ranking function, then with each loop iteration, either e1 is positive and decreases, or e1 remains the same and e2 is positive and decreases or e1 and e2 remain the same and e3 is positive and decreases. Find a lexicographic ranking function for the loop in the following program.

With this prompt, the algorithm is able to solve **37** more benchmarks (prompt $M_3$ in Table 13). Finally, we try similar instructions for the multi-phase ranking functions, and solve **11** more benchmarks (prompt $M_4$ in Table 13), taking the total verified benchmarks to **181/281**. The example shown in Figure 1(c) is one of the benchmarks that fails with the basic prompt, but with the multi-phase prompt, our algorithm infers [z; y; x] as a multi-phase loop ranking function.

*Symbolic baseline:* Ultimate solves **236/281** benchmarks. There are **162/281** that both our algorithm and Ultimate solve,

```
// @requires n >= 0;
// @ensures \result == n % 2;
int isOdd(int n) {
    if (n == 0)  return 0;
    else if (n == 1) return 1;
    else return isEven(n - 1);
}
// @requires n >= 0;
// @ensures \result == 1 - n % 2;
int isEven(int n) {
    if (n == 0) return 1;
    else if (n == 1) return 0;
    else return isOdd(n - 1);
}
int main() {
    int n = unknown_int();
    if (n < 0) return 0;
    int result = isOdd(n);
    assert(result >=0 && result != n % 2);
}
```

Fig. 15: Example recursive program along with LOOPY-generated annotations. The annotated program verifies with Frama-C.

**19/281** that only our algorithm solves, and **74/281** that only Ultimate solves. Thus our algorithm and Ultimate combined solve **255/281**.

## V. RECURSIVE PROGRAMS

While the primary focus of this work has been on dealing with the complexity of loops, we also explore the ability of LLMs to deal with recursive programs. Specifically, programs where the methods are (mutually-) recursive; Figure 15 shows an example. The task here is to infer the pre- and postconditions for the methods in the program, such that Frama-C is able to verify the assertions in the program. The prompt we used for this task is available in the public repository[1]. With a total 31 benchmarks, LOOPY is able to successfully verify $16/31$programs with 8 completions. Ultimate is able to verify 20/31 of these benchmarks. Figure 15 also shows the pre- and postconditions inferred by LOOPY.

## VI. RELATED WORK

*LLMs for invariant generation:* Pei et al. [41] study this problem by building dataset of programs and corresponding invariants and then fine-tune a pre-trained LLM on this dataset. Our approach does not rely on fine-tuning and directly evaluates the capabilities of foundational models. Furthermore, Pei et al. do not focus on generating *inductive* invariants that are necessary for establishing a formal proof of correctness.

Lemur [42] presents a proof calculus and an algorithm to use an LLM to generate and repair invariants. Lemur uses a symbolic verifier to check for inductiveness and generate counterexamples. They use a chaining approach to iteratively strengthen, repair or backtrack on proposed invariants. This necessitates a proof of soundness for single method with loops, and may require further extensions for an interprocedural setting. It is unclear if Lemur would find an inductive invariant even if the LLM proposes all of its ingredients, since Lemur is sensitive to the order in which invariants are proposed. Integrating Houdini with Lemur could be a promising direction for future work. Further, their approach does not apply to proving termination, and even for invariants, it has been evaluated on a much smaller set of benchmarks. Lemur is publicly available but we were unable to run it. On the benchmarks of the Lemur paper, LOOPY and Lemur perform comparably when given the same budget of LLM queries. Among the 133 Code2Inv benchmarks, LOOPY solves 103 benchmarks while Lemur solves 107 benchmarks. Among Lemur's 50 SV-COMP benchmarks, both tools solve 26 benchmarks each.

Yao et al. [43] leverage LLMs to semi-automate proofs for Rust programs in the context of the Verus program verifier. However, they do not consider loop termination or working across multiple methods. Furthermore, their evaluation is not fully automated for the benchmark and only compared against a purely manual baseline. Chakraborty et al. [44] build iRank, a custom model for ranking candidate invariants. iRank is orthogonal and complementary to our work; we can use it as a heuristic for decreasing the number of calls to the oracle by only checking highly-ranked invariant candidates.

*LLMs for proof assistants:* LLMs have been used to automate proof synthesis in interactive proof assistants [45], [46]. Leandojo [46], for instance, fine-tunes a retrieval model for lemma selection and a generative model for proof generation for the Lean theorem prover. Given the general purpose nature of these proof assistants, these approaches are not tailored for automatic program verification that we target in this paper. Our approach also does not require fine-tuning a model.

### A. Threats to validity

A potential concern while working with LLMs is the problem of data contamination, which happens when the benchmarks used for evaluation were already a part of the training data used for the models. In this case, the models can overfit, which affects their ability to generalize to newer benchmarks.

There is no ideal way to completely remove contamination while working with industrial models, or even for open-source ones, given the scope of training data that they consume. We compensate, to the best of our ability, by considering as many benchmarks as we could try, and increasing the diversity of tasks as well as program and invariant features (arrays, ranking functions, etc.). The Stack [47], which is a public code corpus used to train open source models like StarCoder [48] and DeepSeek-Coder-V2 [49], does not contain the SVCOMP and Code2Inv benchmarks. We are also not aware of any other data source where these programs appear alongside their invariants.

Another concern is about the reproducibility of our results. Closed models, such as GPT-4, can get updated any time, which can affect the numbers reported in this paper. Our toolchain is parametric on the choice of LLMs and we do use an open-source model (CodeLlama) to compensate for this concern. LLMs are also stochastic, implying that they can return different responses for the same query. Using multiple completions helps compensate for this stochasticity.

## References


[1] S. Padhi, R. Sharma, and T. D. Millstein, "Data-driven precondition inference with learned features," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, 2016, pp. 42–56. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908099

[2] M. Brockschmidt, Y. Chen, P. Kohli, S. Krishna, and D. Tarlow, "Learning shape analysis," in *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*, ser. Lecture Notes in Computer Science, F. Ranzato, Ed., vol. 10422. Springer, 2017, pp. 66–87. [Online]. Available: https://doi.org/10.1007/978-3-319-66706-5_4

[3] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, R. Bodík and R. Majumdar, Eds. ACM, 2016, pp. 499–512. [Online]. Available: https://doi.org/10.1145/2837614.2837664

[4] J. Yao, G. Ryan, J. Wong, S. Jana, and R. Gu, "Learning nonlinear loop invariants with gated continuous logic networks," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 106–120.

[5] G. Ryan, J. Wong, J. Yao, R. Gu, and S. Jana, "Cln2inv: Learning loop invariants with continuous logic networks," in *International Conference on Learning Representations*, 2020.

[6] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song, "Learning loop invariants for program verification," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[7] X. Si, A. Naik, H. Dai, M. Naik, and L. Song, "Code2inv: A deep learning framework for program verification," *Computer Aided Verification*, vol. 12225, pp. 151 – 164, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:211027794

[8] OpenAI, "GPT-4 technical report," https://doi.org/10.48550/arXiv.2303.08774, 2023.

[9] R. Anil, A. M. Dai, O. Firat, M. Johnson, D. Lepikhin, A. Passos, and et al., "Palm 2 technical report," *CoRR*, vol. abs/2305.10403, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.10403

[10] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, and et al., "Llama 2: Open foundation and fine-tuned chat models," *CoRR*, vol. abs/2307.09288, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2307.09288

[11] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, and et al., "Training language models to follow instructions with human feedback," in *NeurIPS*, 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html

[12] GitHub, "Github copilot," https://github.com/features/copilot, 2022.

[13] Amazon, "Amazon codewhisperer," https://aws.amazon.com/codewhisperer/, 2023.

[14] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," *Formal Aspects Comput.*, vol. 27, no. 3, pp. 573–609, 2015. [Online]. Available: https://doi.org/10.1007/s00165-014-0326-7

[15] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for esc/java," in *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, ser. FME '01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 500–517.

[16] OpenAI, "GPT-3.5," https://platform.openai.com/docs/models/gpt-3-5, 2023.

[17] J. Signoles, B. Desloges, and K. Vorobyov, *E-ACSL User Manual*. [Online]. Available: http://frama-c.com/download/e-acsl/e-acsl-manual.pdf

[18] K. Madhukar, B. Wachter, D. Kroening, M. Lewis, and M. Srivas, "Accelerating invariant generation," in *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '15. Austin, Texas: FMCAD Inc, 2015, p. 105–111.

[19] H. Zhu, S. Magill, and S. Jagannathan, "A data-driven chc solver," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 707–721. [Online]. Available: https://doi.org/10.1145/3192366.3192416

[20] I. Dillig, T. Dillig, and A. Aiken, "Fluid updates: Beyond strong vs. weak updates," in *Programming Languages and Systems*, A. D. Gordon, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 246–266.

[21] S. Chakraborty, A. Gupta, and D. Unadkat, "Diffy: Inductive reasoning of array programs using difference invariants," in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds. Cham: Springer International Publishing, 2021, pp. 911–935.

[22] D. Beyer, "Competition on software verification and witness validation: Sv-comp 2023," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Nature Switzerland, 2023, pp. 495–522.

[23] The Termination Competition, "The Termination Problem Database," https://github.com/TermCOMP/TPDB, 2023.

[24] X. Shi, X. Xie, Y. Li, Y. Zhang, S. Chen, and X. Li, "Large-scale analysis of non-termination bugs in real-world oss projects," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 256–268. [Online]. Available: https://doi.org/10.1145/3540250.3549129

[25] M. Heizmann, J. Hoenicke, and A. Podelski, "Refinement of trace abstraction," in *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, ser. Lecture Notes in Computer Science, J. Palsberg and Z. Su, Eds., vol. 5673. Springer, 2009, pp. 69–85.

[26] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[27] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[28] S. Conchon, A. Coquereau, M. Iguernlala, and A. Mebsout, "Alt-Ergo 2.2," in *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, Oxford, United Kingdom, Jul. 2018. [Online]. Available: https://inria.hal.science/hal-01960203

[29] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 171–177. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_14

[30] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[31] P. Cousot and R. Cousot, "An abstract interpretation framework for termination," in *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, J. Field and M. Hicks, Eds. ACM, 2012, pp. 245–258. [Online]. Available: https://doi.org/10.1145/2103656.2103687

[32] C. Urban, A. Gurfinkel, and T. Kahsai, "Synthesizing ranking functions from bits and pieces," in *Tools and Algorithms for the Construction and Analysis of Systems*, M. Chechik and J.-F. Raskin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 54–70.

[33] M. Brockschmidt, B. Cook, and C. Fuhs, "Better termination proving through cooperation," in *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds., vol. 8044. Springer, 2013, pp. 413–429. [Online]. Available: https://doi.org/10.1007/978-3-642-39799-8_28

[34] M. Giacobbe, D. Kroening, and J. Parsert, "Neural termination analysis," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 633–645. [Online]. Available: https://doi.org/10.1145/3540250.3549120

[35] J. Leike and M. Heizmann, "Ranking Templates for Linear Loops," *Logical Methods in Computer Science*, vol. Volume 11, Issue 1, Mar. 2015. [Online]. Available: http://lmcs.episciences.org/797

[36] A. M. Ben-Amram and S. Genaim, "On multiphase-linear ranking functions," *CoRR*, vol. abs/1703.07547, 2017. [Online]. Available: http://arxiv.org/abs/1703.07547

[37] M. Colón and H. Sipma, "Synthesis of linear ranking functions," in *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, ser. Lecture Notes in Computer Science, T. Margaria and W. Yi, Eds., vol. 2031. Springer, 2001, pp. 67–81. [Online]. Available: https://doi.org/10.1007/3-540-45319-9_6

[38] A. R. Bradley, Z. Manna, and H. B. Sipma, "Linear ranking with reachability," in *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, ser. Lecture Notes in Computer Science, K. Etessami and S. K. Rajamani, Eds., vol. 3576. Springer, 2005, pp. 491–504. [Online]. Available: https://doi.org/10.1007/11513988_48

[39] ——, "The polyranking principle," in *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*, ser. Lecture Notes in Computer Science, L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, Eds., vol. 3580. Springer, 2005, pp. 1349–1361. [Online]. Available: https://doi.org/10.1007/11523468_109

[40] A. Podelski and A. Rybalchenko, "A complete method for the synthesis of linear ranking functions," in *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*, ser. Lecture Notes in Computer Science, B. Steffen and G. Levi, Eds., vol. 2937. Springer, 2004, pp. 239–251. [Online]. Available: https://doi.org/10.1007/978-3-540-24622-0_20

[41] K. Pei, D. Bieber, K. Shi, C. Sutton, and P. Yin, "Can large language models reason about program invariants?" in *Proceedings of the 40th International Conference on Machine Learning*, ser. ICML'23, 2023.

[42] H. Wu, C. Barrett, and N. Narodytska, "Lemur: Integrating large language models in automated program verification," 2023.

[43] J. Yao, Z. Zhou, W. Chen, and W. Cui, "Leveraging large language models for automated proof synthesis in rust," 2023.

[44] S. Chakraborty, S. K. Lahiri, S. Fakhoury, M. Musuvathi, A. Lal, A. Rastogi, A. Senthilnathan, R. Sharma, and N. Swamy, "Ranking llm-generated loop invariants for program verification," in *Findings of The 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP-findings 2023)*, 2023.

[45] E. First, M. N. Rabe, T. Ringer, and Y. Brun, "Baldur: Whole-proof generation and repair with large language models," 2023.

[46] K. Yang, A. M. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. Prenger, and A. Anandkumar, "Leandojo: Theorem proving with retrieval-augmented language models," 2023.

[47] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder 2 and the stack v2: The next generation," 2024.

[48] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder: may the source be with you!" 2023. [Online]. Available: https://arxiv.org/abs/2305.06161

[49] Q. Zhu, D. Guo, Z. Shao, D. Yang, P. Wang, R. Xu, Y. Wu, Y. Li, H. Gao, S. Ma *et al.*, "Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence," *arXiv preprint arXiv:2406.11931*, 2024.