Can Large Language Models Reason about Program Invariants?

Kexin Pei [1 2] **David Bieber** [2] **Kensen Shi** [2] **Charles Sutton** [2] **Pengcheng Yin** [2]

Abstract

Identifying invariants is an important program analysis task with applications towards program understanding, bug finding, vulnerability analysis, and formal verification. Existing tools for identifying program invariants rely on dynamic analysis, requiring traces collected from multiple executions in order to produce reliable invariants. We study the application of large language models to invariant prediction, finding that models trained on source code and fine-tuned for invariant generation can perform invariant prediction as static rather than dynamic analysis. Using a scratchpad approach where invariants are predicted sequentially through a program gives the best performance, finding invariants statically of quality comparable to those obtained by a dynamic analysis tool with access to five program traces.

1. Introduction

Large language models (LLMs) have demonstrated tremendous promise and impressive capability in generating program source code (Chen et al., 2021; Austin et al., 2021; Li et al., 2022). As these techniques become more widespread, it is increasingly important to analyze and explain whether generated code meets its requirements. This requires making statements about *program semantics*, which often involve checking whether boolean predicates, called properties (in the sense used by Claessen & Hughes, 2000; Odena & Sutton, 2020, among many others), hold for all possible executions of the program, e.g., "*for all inputs such that* `obj != nil`*, does the program execute without raising an exception?*" (*cf.* Bieber et al., 2022). To this end, a line of work has explored whether neural networks can predict the results of program execution (Zaremba & Sutskever, 2014; Nye et al., 2021a; Bieber et al., 2020). So far, this problem has proved challenging even for simple programs.

In the programming languages literature, one of the most important insights is to reason at the level of *abstractions* of program states, e.g., the property "is `n >= 1` when line 12 executes?", rather than *concrete states*, such as "`n = 17` at line 12". This has been a fundamental insight from some of the earliest proposals to formalize program semantics (Hoare, 1969; Dijkstra, 1975). This move has computational advantages, because abstracting away details can simplify the analysis, but it is also representational, because the analysis task is often to check over all plausible inputs rather than specific concrete inputs.

If a program property is always true at a given program point, it is an *invariant*, which abstracts multiple program states by finding a common pattern that is easier to reason about. Identifying invariants is undecidable, so previous work has considered predicting them using machine learning (ML), such as predicting heap invariants (Brockschmidt et al., 2015; 2017), reranking invariants produced by dynamic analysis (Hellendoorn et al., 2019), combining learning and static analysis (Ryan et al., 2019; Si et al., 2018; 2020), and predicting specific types of invariants using classification and regression-based approaches (Sharma et al., 2012; 2013a; Garg et al., 2014; 2016; Liu et al., 2012; Sagdeo et al., 2011). Generally speaking, these approaches rely on using task-specific architectures and training sets, and often focus on specific types of invariants. However, given the success of pre-trained LLMs for program synthesis, it is natural to consider that these methods could be effective for predicting invariants as well. Thus, this paper considers the following concrete question: Can we train large language models to predict program invariants?

To tackle this problem, we create a dataset of Java programs with corresponding invariants generated from the well-known Daikon dynamic analyzer (Ernst et al., 2007). Daikon can generate a wide range of invariants based on its predefined templates, and we consider a subset of them that are particularly useful for downstream tasks like bug detection and test generation, such as arithmetic relationships, nullity, and value-set invariants (see Table 2). We fine-tune an LLM, pre-trained on source code, to take as input the source code of a program, and a target program point, and to output a list of invariants for the program state

[1]Columbia University [2]Google Research, Brain Team. Correspondence to: Kexin Pei <kpei@cs.columbia.edu>, David Bieber <dbieber@google.com>, Charles Sutton <charlessutton@google.com>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

```
1  import java.util.*;
2  public class test {
3   public static void main(String[] args)
4   {
5     Scanner in = new Scanner(System.in);
6     int t = in.nextInt();
7     while (t-- > 0) {
8       long n = in.nextLong();
9       long m = in.nextLong();
10      long p = m - (m % 4) + 1;
11      while (p <= m) {
12        if (n % 2 == 0) {
13          n -= p;
14        } else {
15          n += p;
16        }
17        p++;
18      }
19      System.out.println(n);
20    }
21  }
22 }
```

Listing 1: A source code file from our dataset (Solution s288 to problem 1607B). This program simulates a walk of a particle on the integer line.

at that point. Then, for new programs, the model can predict invariants from the source code, without needing to perform traditional static or dynamic analysis, both of which can be expensive.

In our experiments, we find that LLMs are effective at predicting invariants, achieving 86% precision and 86% recall. Remarkably, this is better than the existing dynamic analysis technique Daikon, if the budget allows a small number of test cases to be executed. We also consider a "scratchpad" style of approach (Nye et al., 2021a), in which we ask the model to generate properties for a list of program points all at once, rather than a single program point; this method can be viewed as a kind of abstract interpretation (Cousot & Cousot, 1977; Nye et al., 2021b). Even though this provides no additional information to the model at training or test time, intuitively, it encourages the model to reason through the program execution "step-by-step", and we find that this also improves performance. In the future, we hope that this work will prove useful for a variety of tasks in program analysis and synthesis, such as execution-based program synthesis (Zohar & Wolf, 2018; Ellis et al., 2019; Shi et al., 2022), program verification (Si et al., 2018; 2020) and test generation (Pacheco & Ernst, 2005; Anand et al., 2013).

2. Problem Statement

Given a source file S and a line of code ℓ in S, the task is to predict a list of invariants $P_{\ell 1} \ldots P_{\ell N}$, which are Boolean predicates of the program state that are always true whenever line ℓ is executed, for any input. We do not assume that S is a standalone file that can be executed on its own (although that will be the case in our experiments). Our focus will be on invariants that are associated with function entry, function exit, and the top of loop bodies (Section 2.1), because these types of invariants are particularly useful for downstream analysis tasks. We refer to those lines of code as *program points*, and we assume that ℓ is a line of code that corresponds to a program point.

Table 1: Target invariants for the source code in Listing 1.

Line	Type	Target Invariants
5	function entry	`args[] == []` `args has only one value`
8	loop body	`in has only one value` `t >= 0`

12	loop body	`m >= 1`
		`p <= m`
		`p >= 1`
		`n >= 1`

Listing 1 shows an example of a Java source file from the dataset that we use in our experiments. Table 1 shows a corresponding set of target invariants for this program. These invariants mean that every time Listing 1 is executed on a valid input, when executions reach the line in the leftmost column, all of the expressions on the rightmost column evaluate to true on the current program state. For example, in Listing 1, `p >= 1` is an invariant of the program (assuming that `m` must be positive for the input to be valid) at $\ell = 12$.

Difference with Code Completion In this paper, our approach to program invariant prediction goes beyond the scope of a traditional code completion task. The invariants we focus on do not merely involve completing existing statements in the program, or predicting assertion statements that might be written in the source code. Instead, we consider them as a distinct collection of logical predicates that consistently evaluate to true across all possible execution traces at specific program points.

Static or Dynamic Analysis While we aim to predict the predicates associated with program execution, our approach, during inference, is a type of *static analysis* that does not rely on any dynamic information. However, it diverges from traditional static analysis techniques in that it avoids the complex and often computationally expensive mechanisms that are typically employed in both research and production environments. These include control-flow and data-flow analyses, sophisticated algorithms that repeatedly iterate

over the code until they reach a fixed point (Flanagan & Leino, 2001), and formal verification tools like satisfiability modulo theories (SMT) solvers.

2.1. Invariant Types

The invariants that we predict can be divided into five high-level categories, based on which types of program points they apply to. These are: (1) object invariants, (2) class invariants, (3) function-entry invariants (also known as function preconditions), (4) function-exit invariants (also known as postconditions), and (5) loop invariants. *Object invariants* are those that are true of all class instances, from the perspective of a class user. That is, they are true at the entry and exit of all public methods to the object. *Class invariants* are similar but focus on the static fields of the class. *Function-entry invariants* capture the predicates and relationships of the function arguments and class fields when it is called. *Function-exit invariants* are properties of inputs and local variables that hold when the function finishes execution, including properties indicating whether the variables are changed by the function execution. Finally, *loop invariants* specify predicates on variables accessible at the beginning of each loop iteration, and are particularly important for program verification.

Additionally, we categorize invariants by the type of their Boolean expression; this is orthogonal to the classification based on program points. Under this classification, our work considers eighteen kinds of program invariants (Table 2), including invariants that check purity (`orig`) and nullity (`null`), value-set invariants (`val_set`) and arithmetic invariants (`comp` and `arith`). We choose these because they are natively supported by the dynamic analyzer we used to generate the dataset (see Section 3). Table 2 orders the invariant types based on a heuristic measure of complexity, which is based on the average length of the invariants and the number of variables involved in an invariant, based on our dataset.

Our method treats these different categories in the same way (Section 3); the only difference is in which program point is specified in the prompt. This, our method can be easily applied to invariants at other categories of program points or by other types of expressions. The categorizations are still useful because these different types of invariants will be useful for defining prompts (Section 3).

2.2. Training Data

Finally, as part of the problem, we assume that we have access to a training dataset of programs $\mathcal{S}$. Each program $S \in \mathcal{S}$ has a set of program points $\mathcal{L}$, and a set of *target invariants* $\mathcal{P}_\ell = P_{\ell 1} \ldots P_{\ell N}$ for each program point $\ell \in 1 \ldots |\mathcal{L}|$, which are deemed to be accurate. These target invariants can be generated using any existing static or dynamic analyses. Since this step is performed offline, more computational resources can be allocated for generating the target invariants compared to what would be feasible during test time deployment. The advantage of static analysis is that it is possible to generate invariants that are provably correct, although potentially with lower recall and higher computational cost. Dynamic analysis is conceptually simpler, because it simply executes the program, but has the challenges that many executions may be necessary to obtain accurate invariants. In this paper, we elect to use dynamic analysis based on Daikon (Ernst et al., 2007) (Section 3).

3. Method

We aim to train a language model to reason abstractly about program executions. Observing that program properties are themselves programs, our overall approach begins with a large language model pre-trained on source code, and fine-tunes it for invariant prediction. Motivated by abstract interpretation, we introduce as our main method a *scratchpad* approach to predicting invariants, first incrementally predicting invariants at intermediate program points before finally predicting invariants at the target program point. We also present a *direct* prediction approach where the model predicts the invariants at the target program point without first making predictions for intermediate program points.

3.1. Pre-trained Language Models of Code

We consider Transformer language models pre-trained on permissively licensed code obtained from public GitHub repositories. Our Transformer architecture follows Chowdhery et al. (2022), except that we do not use parallel decoding layers. We consider model scales of 430 million, 1 billion, and 5 billion parameters. The models are trained on 10.5, 21.0, and 96.5 billion tokens of source code, respectively.

3.2. Computing the Target Invariants

To build the samples for training and evaluation, we follow four main steps to preprocess the input programs and generate target invariants. Instrumentation (Step 1) ensures the program points of interest (Section 2.1) are annotated in the submission. By default, Daikon instruments function entry and exit only, so we add dummy function calls at the top of every loop body, which allows Daikon to generate invariants at these program points, e.g., `check_cf1_main_6` in Listing 2. Execution (Step 2) produces traces by running each program on hundreds of distinct program inputs. Daikon performs invariant generation (Step 3) on these traces to produce the target invariants for each program at every program point. Intuitively, Daikon can be understood as searching through a large set of expressions that represent invariants and removing ones that are contradicted by the execution traces, are redundant with other invariants, or that whose frequency is not different enough from chance. Finally, we

Table 2: 18 invariant types we considered in this paper.

Name	Description	Example
	Check whether variable is modified (postcondition)	
`orig`	Variable modified after function exits	`list[] != orig(list[])`
	Check on (derived) variables$^+$ in primitive types	
`null`	Whether variable is null/zero/empty	`n != null`
`one_val`	Whether variable always has only one value	`a has one value`
`comp`	Compare variables with (in)equalities	`i < 10`
`val_set`	Variable has value-set	`i one of {0,1,2}`
	Checks involving math operations	
`arith`	`comp` involving arithmetic operators	`i + j == 10`
`pow`	Variable is the power of another	`res is a power of 2`
`div`	Variable divides another	`people % i == 0`
`sqr`	Variable is the square of another	`max(b[]) == sum(b[])**2`
	Check array element-wise	
`elt_comp`	`comp` on each element	`l[] elements < 7`
`elt_val_set`	`val_set` for each element	`visit[] elements one of {0,1}`
`pairwise`	`comp` neighboring elements pair	`arr[] sorted by` $\leq$
	Check on sequence/array-level property	
`member`	Variable is a member of array	`p in people[j..i-1]`
`sub`	One variable/array is subset of another	`a[] is a subset of b[j..]`
`seqseq`	`comp` each element between arrays	`s[i..]` $\geq$ `t[0..count-1]`
`reverse`	Array is the reverse of another	`a[] is the reverse of c[i..]`
`agg_check`	Comparison on the aggregated values*	`sum(l[]) < 10`
	Conditional checks	
`cond`	Properties that rely on other properties	`(r == false) ==> (B.x` $\geq$ `0)`

$^+$ Derived variables include variable that does not exlicitly appear in the program, e.g., array element.

* Aggregation operation includes `size()`, `min()`, `max()`, `len()`

apply a template-based transformation (Step 4) to produce the input and target strings for each submission for use in fine-tuning. Section 3.3 describes the transformation step for the direct prediction and scratchpad prediction approaches. Additional details about each step are given in Appendix B.

Although the target invariants are based on hundreds of executions, they are still imperfect (Hellendoorn et al., 2019; Petersen; Polikarpova et al., 2009). Dynamic analysis can fail to identify invariants when it deems there to be sufficient probability that the invariant would not hold on additional executions. It is also limited to producing invariants in particular classes, and some valid invariants fall outside of these classes. In the other direction, dynamic analysis can produce false positive invariants when the executions available do not cover the full range of possible executions.

3.3. Input and Target Strings for Fine-tuning

Once the invariants are obtained for the training set programs, we form the input and target strings for fine-tuning.

Direct Prediction Under the direct prediction approach, the input string is a concatenation of source code S and target program point ℓ. The target string is a serialization of the target invariants ordered by invariant type as in Table 2. The target invariants themselves are produced as in Section 3.2. We describe the textual representation of a program point in Appendix B. This baseline represents the standard way to fine-tune a transformer language model for the task, yielding a model of $\mathcal{P}(P_\ell \mid S, \ell)$.

Scratchpad Prediction Under the scratchpad approach, the input string is a concatenation of the source code and a directive specifying the full sequence of program points to

predict invariants for. The target string lists each program point in the sequence followed by its target invariants, culminating in the target program point followed by the target program point's target invariants. For each program point, the target invariants are ordered by invariant type again using the order given in Table 2. As an implementation detail, we include the first program point as the final line of the input string, rather than as the first line of the target string. Appendix E shows examples of both the direct and scratchpad input string and target string formats. This approach models $\mathcal{P}(P_1, \ldots, P_\ell \mid S, 1, \ldots, \ell)$.

Nye et al. (2021a) introduce a scratchpad-based approach for encouraging Transformers to perform more complex operations, by training the model to produce intermediate tokens that are not counted as part of the final answer. Whereas the previous scratchpad work presented examples of training a model to step through concrete executions, in this work we extend the scratchpad approach to work at the abstract level of sequences of invariants.

This approach is also motivated by abstract interpretation (Cousot & Cousot, 1977). Abstract interpretation is a family of program analysis methods which reason at the level of *abstract domains*, which are sets of program states such as "all positive integers", rather than concrete states. Intuitively, the analysis steps through the program, updating the location of each variable in the abstract domain, rather than its concrete value. In our scratchpad prediction approach, we can interpret a list of invariants as a point in an abstract domain. When the LM predicts the invariants P_ℓ given those at $P_{\ell-1}$, this can be interpreted as learning to execute the program in an abstract program invariant space.

4. Experiments

To evaluate our proposed invariant generation methods, we perform a series of experiments on programs obtained from competitive programming contests (Section 4.1). Our primary baseline is Daikon, which generates the ground-truth for training and evaluation by executing the programs on hundreds of possible inputs. However, it is worth noting that real-world programs typically lack such an extensive test suite. Therefore, a crucial aspect of our investigation is to determine whether our learning-based approach can produce higher-quality invariants, especially when Daikon is limited to a small number of program executions.

First, we compare different variants of our language modeling approach for predicting invariants (Section 4.3). Among these, the best approach we find is the scratchpad method predicting invariants for one program point at a time, sequentially. Then, we present the main comparison, of the best LM approach to Daikon invariants (Section 4.4), finding that our language models produce invariants statically of quality comparable to those obtained by Daikon with a budget of up to five program executions.

4.1. Dataset

We evaluate our models on the Java submissions in the Code Contests dataset (Li et al., 2022), which consists of millions of submissions to about four thousand distinct programming challenges; the dataset provides upwards of 200 inputs for each problem. Following the procedure in Section 3.2 we instrument each submission. This allows us to use Daikon to execute each of the submissions, collecting traces to use for invariant generation. Using the available inputs, we obtain 200 traces for each program. Each program may contribute multiple examples to the dataset, at most one per program point. We split the examples by problem, such that no problem contributes examples to more than one dataset split, in order to prevent highly similar submissions from appearing both during training and evaluation. In total, the resulting *Code Contests Java Invariants* dataset includes 1,600,158 training, 86,346 validation, and 24,509 test examples.

4.2. Setup and Metrics

We fine-tune LMs using an initial learning rate of 0.001, and a cosine learning rate decay schedule for 20,000 steps on 64 TPU v4 cores. The batch size is 128. Unless otherwise specified, by default we fine-tune the model on a combined mixture of training examples in different prompt types: direct, scratchpad (Section 3.3), and oracle predictions (described below in Section 4.3). We use greedy decoding during inference. Therefore, there is no sampling strategy employed in the prediction.

We treat as ground truth the invariants with Daikon, obtained using two hundred inputs per program, reflecting the assumption that false positives are less likely with a larger number of inputs. To measure the similarity between the set of model-predicted invariants the ground truth, we report Jaccard similarity, precision, recall, and F_1 score at level of invariants. The evaluation sets for the different prompt types are created using the same set of underlying programs and target invariants.

4.3. Comparison of LM approaches

In our first set of experiments, we evaluate our proposed approaches (Section 3) to invariant generation. Besides the *direct* generation and step-by-step *scratchpad* generation (Section 3.3), we report results on an *oracle* approach that assumes access to the target invariants at the program points that precede the target program point ℓ, i.e., $\{1, P_1, \cdots, \ell - 1, P_{\ell-1}\}$. Therefore, its prediction task is $P(P_\ell \mid S, 1, P_1, \cdots, \ell - 1, P_{\ell-1}, \ell)$. Compared to the di-

rect and scratchpad method that predict invariants using only the static information of source code and program points, this oracle approach leverages additional information of program behavior before ℓ via dynamic analysis using Daikon.

Comparing Prompting Strategies Figure 1 summarizes the results obtained using different strategies of generating input strings for prompting (Section 3.3). We find that scratchpad prediction outperforms direct prediction. As discussed in Section 3.3, the scratchpad approach allows the model to reason about program behavior in a step-by-step fashion by predicting invariants for intermediate program points over the course of execution. Such contextual information is crucial for predicting invariants at target program points. This is inline with existing findings on program induction (Nye et al., 2021a), which show the advantage of predicting intermediate execution results compared to direct prediction of the final output.

Unsurprisingly, replacing model-predicted intermediate invariants in scratchpad with the target ones from Daikon (oracle prediction) further improves performance. In addition, our results align with typical scaling behavior of LLMs, with increased performance as the model scales (Kaplan et al., 2020).

Impact of Invariant Order As described in Section 3.2, we impose a canonical order on the set of invariants that correspond to a program point, with the intuition to place simpler invariants before more complex ones. Figure 2a compares predicting invariants following this order with the ablation that generates invariants using three randomly shuffled orders of invariant types. Interestingly, we do not find the choice of canonical ordering to be significant.

Mixing Prompting Strategies So far we have considered fine-tuning models using the combined training samples from three different prompt types, i.e., direct, scratchpad and oracle. Figure 2b shows how models trained with this combined training set compare to those with separated training sets on individual prompt types. Combined training appears helpful for the scratchpad approach, while separated training yields better results for direct and oracle prediction.

Recall that for combined training, we evaluate the same model on the same set of problems with different prompt types (Section 4.2). Intuitively, in the combined training setting, while the model has seen the same number of training instances from each prompt type, the target strings in the scratchpad examples ($P_1, \cdots, P_\ell$) are longer than those in the other two prompt types (P_ℓ, Section 3.3). Therefore the model is optimized on more target tokens from the scratchpad prompt strategy, which likely explains worse results on the other prompt types compared to separate training. The scratchpad approach likely also benefits from knowledge transfer when training together with the other prompt variants, since it shares similar input strings with direct prediction, while encapsulating the oracle prediction objective $P(P_\ell|P_1, \cdots, P_{\ell-1}, \dot{)}$ as a factor in its own objective.

4.4. Quality of Predicted Invariants

As our main measurement of the quality of the predicted invariants, we compare the invariants from the LM with those generated by Daikon when fewer inputs are available. This is a realistic comparison, as typically only a small number of inputs or test cases are available for a typical program during inference.

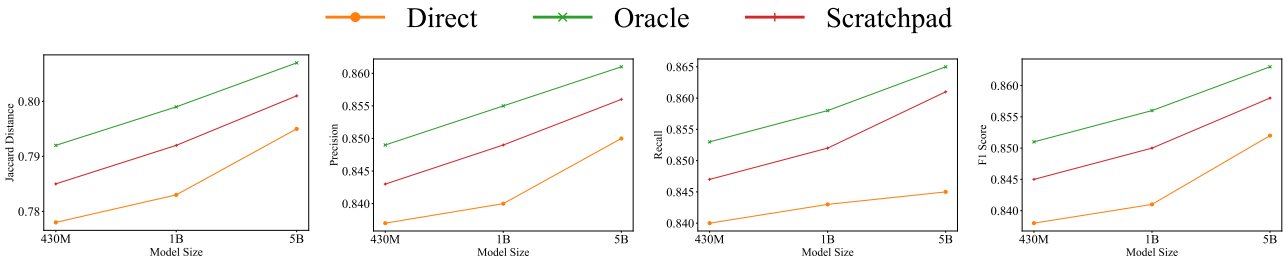
Figure 3 shows invariant generation performance of our language model and of Daikon as a function of the number of inputs (traces) available to the method; for our models, the number of traces is always zero. The performance of Daikon monotonically increases with the number of available traces, reaching its maximum performance (and precisely producing the target invariants) when the maximum number of traces are provided. Remarkably, when only a small number of traces are available (5 or fewer), our approach outperforms Daikon. Since dynamic analysis is always expensive, as it requires instrumenting the code and running it with tests, it is common that only zero or a small number of traces are practical to collect. This makes our approach, which is completely static during inference, appealing.

4.5. Prompting LLMs without Adaptation

In an early pilot study, we experimented with large language models (LLMs) that had not been specifically adapted for the task of invariant prediction. We observed that the performance of the LLM was subpar, which led us to decide on proceeding with further fine-tuning to enhance the model's capabilities.

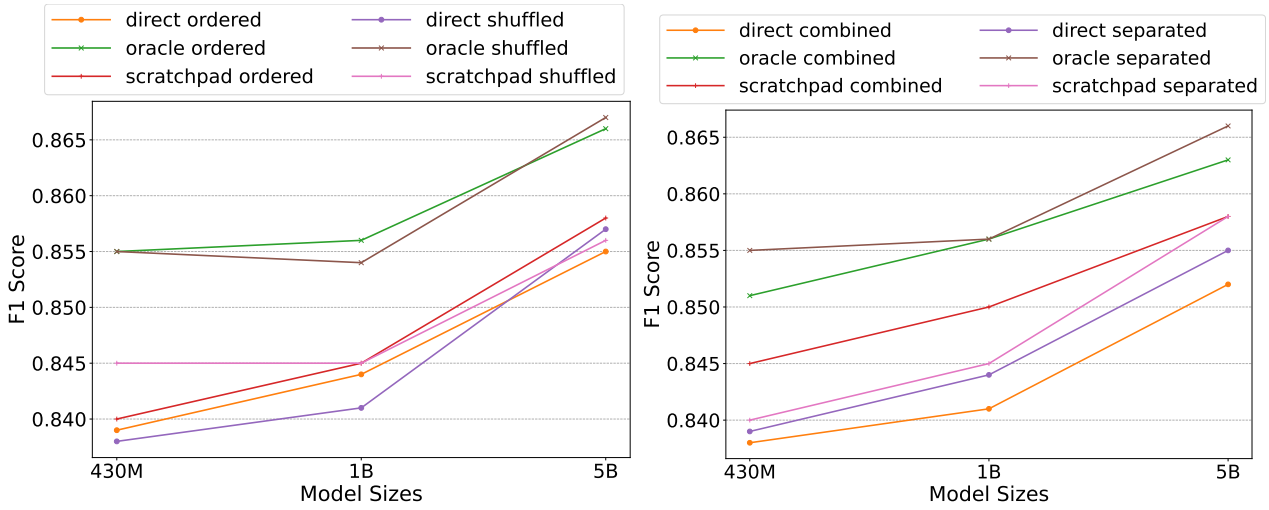
This section revisits this choice and directly prompt one of the most recent and larger LLMs, GPT-4, to assess whether an increase in model size could enhance the prediction of invariants even without any adaptation. We used two different sets of prompts: (1) all the example listings provided in the paper, including Listings 1 through 7, which contained several intriguing invariants, and (2) a randomly chosen set of 10 examples from the test set.

We performed zero-shot prompting, where the program was followed by a comment requesting the model to directly generate the invariants (Listing 8). The results of these prompts were far from those obtained by our much smaller (Section 3.1) and adapted/fine-tuned LMs (Table 4). In the case of (1), GPT-4 achieved a precision score of 0.44 and a recall score of 0.384. However, for (2), GPT-4 was unable to generate any of the invariants present in the ground truth.



(a) Jaccard distances (b) Precision (c) Recall (d) F1 scores

Figure 1: Our results on invariant prediction across different model sizes and prompting methods. Model sizes are 430M, 1B, and 5B parameters; prompting methods are direct, scratchpad, and oracle.



(a) Ordered vs. shuffled (b) Combined vs. separated

Figure 2: Invariant prediction performance comparing (left) target invariants ordered in the curriculum setting (Table 2) vs. randomly shuffled type orders, and (right) training the models on combined vs. separated prompt types.

While these results are not directly comparable to those reported in the paper due to the differences in the experimental setup, they nonetheless suggest that model adaptation, such as fine-tuning, remains a crucial component for this specific task of invariant prediction.

5. Case Studies

We present empirical results from our model and examine their implications for test generation.

5.1. Uncovering New Invariants

The target invariants obtained from Daikon are not always correct or exhaustive, as discussed in Section 3. Indeed, our models regularly produce invariants not found by Daikon that, upon manual inspection, we deem correct. This is perhaps surprising given that our models were trained only on supervision obtained from Daikon. Listing 5 in Appendix D shows an example: the invariant `test >= 1` is correctly predicted by our models but is not in the target invariants set. Similarly, Listing 4 shows that the models omit an incorrect target invariant `return == 7` and instead predict the correct invariant `return >= 1`, which is absent from the target set. The manual inspection is based on checking the text description of problem 1594-A (Listing 4 is one of its solutions), specifically the requirement: "The first line contains a single integer t ($1 \leq t \leq 10^4$) - the number of test cases." Here, t is the user-provided input that will be read by the statement `int test = input.nextInt()` in Listing 4.

The language models are not limited to considering the set of inputs used for executions. Instead, they generalize from the examples they were trained on, which collectively contain information about the set of reasonable inputs for a given function. By contrast, Daikon produces false positive invariants that overfit to any spurious regularities present in the

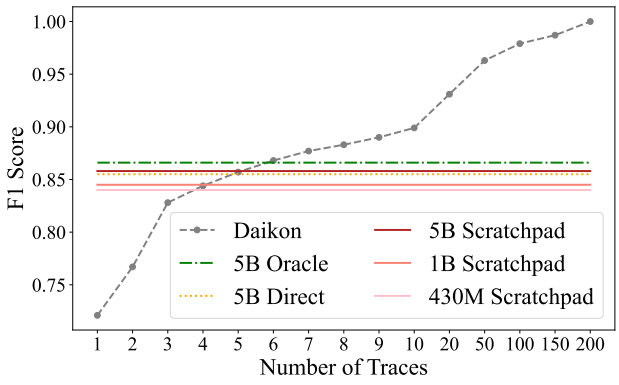


Figure 3: Daikon vs. ours with different prompt types and model sizes.

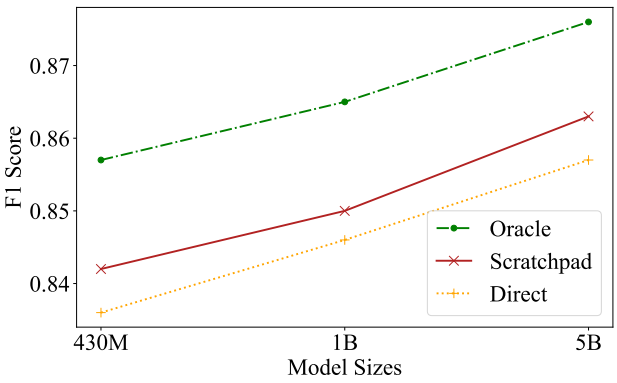


Figure 4: Invariant prediction by following program points in a *backward* manner.

set of inputs present for a program; this is why the Daikon invariants sometimes include overly constrained invariants like `return == 7` even when the invariants predicted by the language models do not.

5.2. Invariants Uncovered by Scaling the Model

Our results in Section 4.3 show improved performance by scaling the model. Here, we attempt to understand patterns in the predicted invariants from models at different size. Listings 2 and 3 in Appendix D present two examples. First, we observe that larger models are better at inferring non-trivial invariants derived from multiple steps of computation (e.g., `a one of` $\{-1, 1\}$ in Listing 2, `length` $>$ `nig` in Listing 3). This is in line with the scaling behavior of language models for other reasoning tasks such as program induction (Nye et al., 2021a) and solving math word problems (Austin et al., 2021). Second, larger models also tend to generalize better and emit fewer spurious invariants that only hold for certain I/O examples, such as the value-set invariants with `one of` predicates in Listing 3.

5.3. Test Generation

Invariants at the start of a function can act as constraints for test generation tools. Randoop is one such tool, typically using Daikon to generate invariants at the start of a function, and using these invariants as constraints to produce plausible inputs both to identify failures in the function and to generate regression tests (Pacheco et al., 2007). Our approach can replace Daikon in this pipeline. With our language model approach to invariant generation, one can generate invariants for the start of a function conditioning on its source code. One can also take a more directed approach; to generate invariants at the start of a function that are likely to lead to the execution of a particular program point ℓ, one could condition on invariants at each branch point on the path toward ℓ that, if satisfied, would lead to its execution.

To investigate the potential of our approach for test generation, we perform a final experiment focused on generating invariants at earlier points in a function. Specifically, we exclude the prediction of invariants for the final program point and introduce class invariants. This modification results in a revised dataset comprising 1,284,492 training examples, 61,431 validation examples, and 21,576 test examples. We then introduce variants of the scratchpad and oracle approaches by reversing the order of program points used. These variants aim to predict invariants starting from program points that occur later in the program, such as post-conditions and loop invariants, before considering those that appear earlier, such as pre-conditions.

To use backward invariant prediction for test generation, we observe that the invariants generated at the start of the program form a set of constraints over program inputs. Selecting a concrete map from each input variable to concrete values satisfying these constraints yields a concrete test. Figure 4 shows the results.

6. Related Work

Traditional approaches formulate program invariant inference as a search problem, guided by heuristics (Galeotti et al., 2015; Sharma & Aiken, 2016). Guess-And-Check (Sharma et al., 2013b) employs a data-driven approach to come up with invariant candidates in a form of conjunctions of linear equalities by analyzing traces. NumInv (Nguyen et al., 2017) extends Guess-And-Check to produce polynomials beyond equalities such as octagonal inequalities, and queries the symbolic execution engine (KLEE, Cadar et al., 2008) for counterexamples to guide its search. LoopInvGen (Padhi et al., 2017) produces arbitrary invariant candidates by composing atomic invariants based on program synthesis, which are guided by counterexamples produced by the solver.

Learning program invariants has been increasingly studied. For example, by observing the runtime behavior of the program, Daikon (Ernst et al., 2007) learns candidate invariants from its predefined invariant templates that the traces do not violate. In contrast, Code2Inv (Si et al., 2020) uses reinforcement learning with graph neural networks to guide the search for candidate program invariants. CLN2Inv (Ryan et al., 2019) constructs candidate invariant templates (with learnable parameters) based on static analysis. It then constructs a neural network architecture that explicitly represents the invariant templates with relaxed logic to smooth the conjunction, disjunction, negation, etc., which enables learning the parameters of the invariants in a differentiable manner. Hellendoorn et al. (2019) present a deep learning approach to predict whether an invariant proposed by Daikon is correct, which can be used as a reranking score for invariants produced by Daikon. Generally speaking, a limitation of these methods is that they can incur substantial overhead due to dynamic tracing (Ernst et al., 2007) or expensive search procedures (Si et al., 2020).

Several learning-based approaches have been proposed for specific classes of invariants. One line of work uses learning specifically of algebraic invariants (Sharma et al., 2012; 2013a;c; Garg et al., 2014; 2016), such as `arith` in Table 2. Although (as can be seen from the examples) many of the invariants produced by our models are algebraic, LLMs are additionally able to predict much more general invariants. Brockschmidt et al. (2015; 2017) present a neural network approach for predicting heap invariants in separation logic, which can be filtered using static analysis. Invariant mining techniques have also been explored for design verification of hardware (Liu et al., 2012; Sagdeo et al., 2011); these approaches dynamically collect hardware states and apply learning methods such as decision trees and linear regression to generate specific classes of invariants.

To our knowledge, none of the previous work has considered large pre-trained language models. Therefore, our approach does not need dynamic analysis during inference, and can generalize across various types of invariants. Moreover, our model is efficient, taking only a single inference pass using the language models without iterative search. Additionally, our models can learn how invariants evolve across program points, which is a staple of traditional static analyses, but has been difficult to incorporate into learning approaches.

7. Conclusion

We have presented a method for predicting invariants of programs using large language models (LLMs). The method fine-tunes LLMs pre-trained on code to map from a source code file and a program point of interest to a list of predicted invariants. We find that LLMs are effective, achieving 86% precision and recall, with better performance than Daikon if the set of available tests is limited. Future work includes extending the set of invariants, and learning specific invariants that are useful for particular downstream tasks such as program synthesis, verification, and test generation.

Acknowledgements

We would like to express our sincere appreciation to Vincent Hellendoorn, Henryk Michalewski, Varun Godbole, Marc Brockschmidt, Dan Zheng, Hanjun Dai, Petros Maniatis, and the members of the Learning for Code team at Google for their invaluable feedback to this research paper. Additionally, we extend our gratitude to the anonymous reviewers for their thoughtful and constructive comments, which have significantly enhanced the quality of this work.

References

Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., Harman, M., Harrold, M. J., McMinn, P., Bertolino, A., Jenny Li, J., and Zhu, H. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, 86(8):1978–2001, August 2013.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. Program synthesis with large language models. August 2021.

Bieber, D., Sutton, C., Larochelle, H., and Tarlow, D. Learning to execute programs with instruction pointer attention graph neural networks. *Advances in Neural Information Processing Systems*, 33:8626–8637, 2020.

Bieber, D., Goel, R., Zheng, D., Larochelle, H., and Tarlow, D. Static prediction of runtime errors by learning to execute programs with external resource descriptions. March 2022.

Brockschmidt, M., Chen, Y., Cook, B., Kohli, P., and Tarlow, D. Learning to decipher the heap for program verification. In *ICML Workshop on Constructive Machine Learning (CML)*, 2015.

Brockschmidt, M., Chen, Y., Kohli, P., Krishna, S., and Tarlow, D. Learning shape analysis. In *Static Analysis Symposium (SAS)*, 2017.

Cadar, C., Dunbar, D., Engler, D. R., et al. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pp. 209–224, 2008.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H.,

Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W., Nichol, A., Babuschkin, I., Balaji, S., Jain, S., Carr, A., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code. July 2021.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

Claessen, K. and Hughes, J. QuickCheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pp. 268–279, New York, NY, USA, September 2000. Association for Computing Machinery.

Cousot, P. and Cousot, R. Abstract interpretation. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '77*, New York, New York, USA, 1977. ACM Press.

Dijkstra, E. W. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8): 453–457, 1975.

Ellis, K., Nye, M. I., Pu, Y., Sosa, F., Tenenbaum, J. B., and Solar-Lezama, A. Write, execute, assess: Program synthesis with a REPL. *Advances in Neural Information Processing Systems {(NeurIPS)}*, 2019.

Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

Flanagan, C. and Leino, K. R. M. Houdini, an annotation assistant for esc/java. In *FME 2001: Formal Methods for Increasing Software Productivity: International Symposium of Formal Methods Europe Berlin, Germany, March 12–16, 2001 Proceedings*, pp. 500–517. Springer, 2001.

Galeotti, J. P., Furia, C. A., May, E., Fraser, G., and Zeller, A. Inferring loop invariants by mutation, dynamic analysis, and static checking. *IEEE Transactions on Software Engineering*, 41(10):1019–1037, 2015.

Garg, P., Löding, C., Madhusudan, P., and Neider, D. ICE: A robust framework for learning invariants. In *Computer Aided Verification*, Lecture notes in computer science, pp. 69–87. Springer International Publishing, Cham, 2014.

Garg, P., Neider, D., Madhusudan, P., and Roth, D. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pp. 499–512, New York, NY, USA, January 2016. Association for Computing Machinery.

Hellendoorn, V. J., Devanbu, P. T., Polozov, O., and Marron, M. Are my invariants valid? a learning approach. March 2019.

Hoare, C. A. R. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

Kaplan, J., McCandlish, S., Henighan, T. J., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. Scaling laws for neural language models. *ArXiv*, abs/2001.08361, 2020.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., Hubert, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D., Sutherland Robson, E., Kohli, P., de Freitas, N., Kavukcuoglu, K., and Vinyals, O. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

Liu, L., Sheridan, D., Tuohy, W., and Vasudevan, S. A technique for test coverage closure using GoldMine. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 31(5): 790–803, May 2012.

Nguyen, T., Antonopoulos, T., Ruef, A., and Hicks, M. Counterexample-guided approach to finding numerical invariants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 605–615, 2017.

Nye, M., Andreassen, A. J., Gur-Ari, G., Michalewski, H., Austin, J., Bieber, D., Dohan, D., Lewkowycz, A., Bosma, M., Luan, D., et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021a.

Nye, M., Pu, Y., Bowers, M., Andreas, J., Tenenbaum, J. B., and Solar-Lezama, A. Representing partial programs with blended abstract semantics. In *International Conference on Learning Representations (ICLR)*, 2021b.

Odena, A. and Sutton, C. Learning to represent programs with property signatures. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rylHspEKPr.

Pacheco, C. and Ernst, M. D. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 - Object-Oriented Programming*, pp. 504–527. Springer Berlin Heidelberg, 2005.

Pacheco, C., Lahiri, S. K., Ernst, M. D., and Ball, T. Feedback-directed random test generation. In *ICSE 2007, Proceedings of the 29th International Conference on Software Engineering*, pp. 75–84, Minneapolis, MN, USA, May 2007.

Padhi, S., Sharma, R., and Millstein, T. Loopinvgen: A loop invariant generator based on precondition inference. *arXiv preprint arXiv:1707.02029*, 2017.

Petersen, M. K. A. An evaluation of daikon: A dynamic invariant detector.

Polikarpova, N., Ciupa, I., and Meyer, B. A comparative study of programmer-written and automatically inferred contracts. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pp. 93–104, New York, NY, USA, July 2009. Association for Computing Machinery.

Ryan, G., Wong, J., Yao, J., Gu, R., and Jana, S. Cln2inv: learning loop invariants with continuous logic networks. *arXiv preprint arXiv:1909.11542*, 2019.

Sagdeo, P., Athavale, V., Kowshik, S., and Vasudevan, S. PRECIS: Inferring invariants using program path guided clustering. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pp. 532–535, November 2011.

Sharma, R. and Aiken, A. From invariant checking to invariant inference using randomized search. *Formal Methods in System Design*, 48(3):235–256, 2016.

Sharma, R., Nori, A. V., and Aiken, A. Interpolants as classifiers. In *International Conference on Computer Aided Verification*, 2012.

Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., and Nori, A. V. A data driven approach for algebraic loop invariants. In *Programming Languages and Systems*, pp. 574–592. Springer Berlin Heidelberg, 2013a.

Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., and Nori, A. V. A data driven approach for algebraic loop invariants. In *European Symposium on Programming*, pp. 574–592. Springer, 2013b.

Sharma, R., Gupta, S., Hariharan, B., Aiken, A., and Nori, A. V. Verification as learning geometric concepts. In *Static Analysis*, Lecture notes in computer science, pp. 388–411. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013c.

Shi, K., Dai, H., Ellis, K., and Sutton, C. CrossBeam: Learning to search in Bottom-Up program synthesis. In *International Conference on Learning Representations (ICLR)*, 2022.

Si, X., Dai, H., Raghothaman, M., Naik, M., and Song, L. Learning loop invariants for program verification. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pp. 7762–7773, Red Hook, NY, USA, December 2018. Curran Associates Inc.

Si, X., Naik, A., Dai, H., Naik, M., and Song, L. Code2Inv: A deep learning framework for program verification. In *Computer Aided Verification*, Lecture notes in computer science, pp. 151–164. Springer International Publishing, Cham, 2020.

Tabachnyk, M. and Nikolov, S. Ml-enhanced code completion improves developer productivity. 2022. URL https://ai.googleblog.com/2022/07/ml-enhanced-code-completion-improves.html.

Zaremba, W. and Sutskever, I. Learning to execute. 2014.

Ziegler, A., Kalliamvakou, E., Li, X. A., Rice, A., Rifkin, D., Simister, S., Sittampalam, G., and Aftandilian, E. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, pp. 21–29, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392730. doi: 10.1145/3520312.3534864. URL https://doi.org/10.1145/3520312.3534864.

Zohar, A. and Wolf, L. Automatic program synthesis of long programs with a learned garbage collector. September 2018.

A. Broader Impact

As machine learning advances, developer tools have come to greater rely upon imperfect predictions from learned systems. In aggregate, these advances improve developer productivity and save time (Tabachnyk & Nikolov, 2022; Ziegler et al., 2022). However, incorrect predictions from machine learning developer tools can also slow developers down, possibly leading developers to introduce bugs or otherwise make mistakes during development. It is important for tool makers to take a human-centric approach when designing developer tools that rely on predictions that may be wrong, providing adequate signals for developers to make informed choices when using these tools. Similarly, it is incumbent upon developers using these tools to think critically about the outputs they produce, rather than trusting the tools to behave correctly in all situations. Chen et al. (2021) and

Chowdhery et al. (2022) further discuss the broader impact of using large language models for program synthesis.

B. Dataset Processing Pipeline

We provide implementation details about each step of the dataset processing pipeline here.

Step 1: Instrumentation. This step consists of source code transformations to the Java submissions that are necessary in order to use Daikon to identify invariants in the programs. As demonstrated in Appendix E, this step inserts dummy method calls in the program to support tracing the variables of interest at certain program point. By default, Daikon supports generating invariants for class and object fields, and the arguments and class fields at method entry and exit (see Section 2.1), but omits loop invariants, an important class of invariants of the program. We address this issue by directly instrumenting the source code. Specifically, we add dummy function calls with empty body at the loop entry, mitigating this Daikon limitation. The signature of the dummy function call conforms to the following format: `public static void check_<class name>_<method>_<loop line number>(<accessible variables>)`. The `<loop line number>` indicates the beginning line (zero-based) of the loop in the source code file. The `<accessible variables>` here includes all variables, e.g., class fields, function arguments, and local variables, as long as their definition *reaches* the loop entry from all *control flow paths* (Section 2).

For example, in Listing 8, `check_test_main_9(p, m, n);` marks a program point with loop invariants on the variables `p`, `m`, and `n`, but variables `in` and `t` are not included.

In order to ensure the instrumented submission compiles, we also add the function definition at the end of the class. We fix the dummy function to be always a `public static` method of the class without return values. Finally, we compile the instrumented submission with Daikon so that Daikon can trace its execution.

Step 2: Execution. Execution takes place in two phases. We first use Daikon's DynComp binary, which uses dynamic analysis to obtain variable type information; this prevents variables of different types from being compared in the assertions we will generate in Step 3. We next execute each instrumented submission to collect traces on all available inputs using Daikon's Chicory binary. The resulting traces contain the values of all variables for all runs of each submission. If a submission's execution exceeds a 60 second timeout limit, we drop it from consideration.

Step 3: Invariant generation. Once the traces are collected we use Daikon to generate invariants. We adjust the Daikon settings such that the generated invariants are non-trivial and diverse. The resulting invariant types are listed in Table 2.

Step 4: Data transformation. Finally, for each variant of our method, we convert the programs and their generated invariants into a format suitable for model training. We describe here the text transformations required to produce the inputs and targets for each of our approaches.

For the *direct* method, the transformation to produce the input is to append the comment text `// predict invariants for <program point>` following the submission source code. The target is produced by listing the invariants at that program point, ordered according to the invariant type order given in Table 2. We give an example in Listings 8 and 9 of Appendix E.

In the *scratchpad* method, we construct the input by appending the comment text `// predict invariants for <comma-separated program point list>` on one line, followed by `<first program point in list>` on the subsequent line. The target consists in alternating the list of invariants at the named program point, followed by the next program point in the list. This concludes with naming the target program point, followed by the ordered list of invariants at the target program point. We give an example in Listings 10 and 11 of Appendix E.

Finally in the *oracle* approach, the inputs and targets together match those of the scratchpad approach, but additional invariant information is provided in the inputs rather than predicted in the targets. Specifically, the inputs contain alternately each program point from the program point list and its invariants, up to and including the target program point, but not its invariants. The target is solely the invariants for the target program point.

C. Complete Experimental Results

Daikon's results. Table 3 shows Daikon's performance when varying the number of traces. We measure Jaccard distance (d_J), precision (P), recall (R), and F1 score (F1). Not surprisingly, Daikon's performance increases monotonically with the number of traces. In addition, the performance gains obtained by increasing the number of traces diminish when there are already enough traces. It is worth noting that dynamic analysis (executing the program and logging the intermediate execution states) is generally more expensive than static analysis. In our case, Daikon requires compiling the code first and tracing the execution twice (see Section B), in which each procedure takes up to 60 seconds (as a timeout) to finish. In contrast, our model takes only the static code as input without incurring any overhead from dynamic analysis during inference.

Table 3: Daikon results when running on different number of traces.

# Traces	d_J	P	R	F1
1	.659	.743	.700	.721
2	.703	.794	.741	.767
3	.783	.834	.823	.828
4	.801	.849	.839	.844
5	.814	.863	.851	.857
6	.828	.873	.863	.868
7	.839	.883	.871	.877
8	.847	.889	.878	.883
9	.856	.895	.886	.890
10	.865	.904	.895	.899
20	.906	.934	.929	.931
50	.947	.962	.964	.963
100	.969	.978	.981	.979
150	.981	.986	.989	.987
200	1.00	1.00	1.00	1.00

Our results. Table 4 includes the complete results of the models on the test set (Section 4.1). We have described the results in Section 4 from different perspectives. Note that in the case of direct prediction, the forward and backward approaches are exactly the same since their predictions do not condition on preceding or succeeding invariants. The results differ because we omit predicting class invariants in the training set of forward prediction. The reason is that that we do not have any preceding invariants before predicting the class invariants. The backward prediction, however, will have succeeding invariants already provided or predicted before predicting class invariants. Therefore, we consider class invariants in the backward prediction task.

D. Selected Examples

We present model predictions alongside the target invariants for a number of examples in the dataset in Listing 2 - 7. The results are discussed in Section 4 and Section 5. Each example is presented as the input string used in the scratchpad method. Metrics are computed with respect to the target invariants.

Table 4: Our results categorized by the finetuning strategies, the prediction types, the order of invariants, the model sizes, and the prediction directions.

Finetune Strategies	Prompt Types	Invariant Orders	Size	Forward Analysis				Backward Analysis			
				d_J	P	R	F1	d_J	P	R	F1
Separated	Direct	Ordered	430M	.780	.837	.842	.839	.773	.829	.823	.826
			1B	.785	.842	.847	.844	.780	.834	.829	.831
			5B	.800	.854	.857	.855	.807	.855	.853	.854
		Shuffled	430M	.779	.836	.841	.838	.774	.830	.822	.826
			1B	.781	.838	.844	.841	.786	.839	.833	.836
			5B	.801	.854	.861	.857	.806	.854	.852	.853
	Oracle	Ordered	430M	.797	.853	.857	.855	.808	.858	.853	.855
			1B	.799	.856	.857	.856	.814	.862	.861	.861
			5B	.812	.864	.869	.866	.831	.877	.874	.875
		Shuffled	430M	.797	.854	.856	.855	.810	.860	.856	.858
			1B	.794	.853	.855	.854	.808	.858	.853	.855
			5B	.812	.866	.869	.867	.833	.879	.876	.877
	Scratchpad	Ordered	430M	.778	.839	.842	.840	.785	.839	.834	.836
			1B	.785	.843	.847	.845	.794	.845	.841	.843
			5B	.801	.857	.860	.858	.817	.865	.862	.863
		Shuffled	430M	.786	.843	.847	.845	.787	.839	.836	.837
			1B	.785	.844	.846	.845	.791	.843	.838	.840
			5B	.798	.854	.859	.856	.815	.863	.861	.862
Combined	Direct	Ordered	430M	.778	.837	.840	.838	.788	.840	.833	.836
			1B	.783	.840	.843	.841	.799	.849	.843	.846
			5B	.795	.850	.854	.852	.811	.859	.856	.857
		Shuffled	430M	.778	.837	.840	.838	.786	.839	.833	.836
			1B	.783	.842	.842	.842	.796	.847	.839	.843
			5B	.801	.855	.858	.856	.808	.856	.852	.854
	Oracle	Ordered	430M	.792	.849	.853	.851	.810	.861	.854	.857
			1B	.799	.855	.858	.856	.819	.868	.862	.865
			5B	.807	.861	.865	.863	.832	.879	.874	.876
		Shuffled	430M	.794	.851	.854	.852	.808	.858	.853	.855
			1B	.796	.854	.854	.854	.816	.866	.860	.863
			5B	.812	.865	.869	.867	.830	.877	.872	.874
	Scratchpad	Ordered	430M	.785	.843	.847	.845	.792	.846	.838	.842
			1B	.792	.849	.852	.850	.801	.854	.847	.850
			5B	.801	.856	.861	.858	.816	.866	.861	.863
		Shuffled	430M	.785	.844	.847	.845	.791	.843	.838	.840
			1B	.790	.850	.849	.849	.802	.854	.846	.850
			5B	.807	.861	.865	.863	.815	.864	.860	.862

```
import java.util.*;
public class cf1 {
  public static void main(String[] args) {
    int t;
    Scanner input = new Scanner(System.in);
    t = input.nextInt();
    while (t != 0) {

      check_cf1_main_6(input);
      int n, m, rb, cb, rd, cd;
      n = input.nextInt(); // rows
      m = input.nextInt(); // columns
      rb = input.nextInt();
      cb = input.nextInt();
      rd = input.nextInt();
      cd = input.nextInt();
      int count = 0, a = 1, b = 1;
      while (true) {

        check_cf1_main_15(a, b, count);
        if (rb == rd || cb == cd) {
          break;
        }
        if (rb + a > n || rb + a < 1) {
          a *= -1;
        }
        if (cb + b > m || cb + b < 1) {
          b *= -1;
        }
        rb += a;
        cb += b;
        count++;
      }
      t--;
      System.out.println(count);
    }
  }

  public static void check_cf1_main_15(int a, int b, int count){}

  public static void check_cf1_main_6(Scanner input){}
}

// predict invariants for cf1.main.ENTER(String[] args),
    cf1.check_cf1_main_6.ENTER(Scanner input), cf1.check_cf1_main_15.ENTER(int a, int b,
    int count)
cf1.main.ENTER(String[] args)
```

Listing 2: Scratchpad input string for solution s310 to problem 1623A. As the model size increases, additional invariants are uncovered. The invariants predicted by the scratchpad approach at all model sizes are shown in the table.

Invariants	Target	430M	1B	5B
`count >= 0`	✓	✓	✓	✓
`a >= 1`	✗	✓	✗	✗
`a one of { -1, 1 }`	✓	✗	✓	✓
`a != 0`	✓	✗	✓	✓
`b one of { -1, 1 }`	✓	✗	✓	✓
`b != 0`	✓	✗	✓	✓
`b >= 1`	✗	✓	✗	✗
Jaccard	1.00	0.14	1.00	1.00
Precision	1.00	0.33	1.00	1.00
Recall	1.00	0.20	1.00	1.00
F1 Score	1.00	0.25	1.00	1.00

E. Prompt Formats

Here we demonstrate an example of the full input and target strings used by the direct and scratchpad approaches. We use the example given in Listing 1 as a running example. The input string and target string for the direct method are given in Listing 8 and Listing 9 respectively. The input string and target string for the scratchpad method are given in Listing 10 and Listing 11.

```java
import java.io.*;
import java.util.*;

public class absentremainder {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    StringBuilder b = new StringBuilder();
    int numCases = Integer.parseInt(in.readLine());
    for (int i = 0; i < numCases; i++) {
      check_absentremainder_main_9(i, numCases, in, b);
      ArrayList<Integer> list = new ArrayList<>();
      int length = Integer.parseInt(in.readLine());
      StringTokenizer tokenizer = new StringTokenizer(in.readLine());
      for (int j = 0; j < length; j++) {
        list.add(Integer.parseInt(tokenizer.nextToken()));
      }
      int nig = 0;
      for (int j = 0; j < length; j++) {
        check_absentremainder_main_18(j, length, list, nig);
        if (list.get(j) < list.get(nig)) {
          nig = j;
        }
      }
      int counter = 0;
      for (int j = 0; j < length; j++) {
        check_absentremainder_main_24(j, length, nig, counter, b, list);
        if (j != nig && counter < length / 2) {
          b.append(list.get(j) + " " + list.get(nig) + "\n");
          counter++;
        } else if (counter >= length / 2) {
          break;
        }
      }
    }
    System.out.print(b);
    in.close();
  }
  public static void check_absentremainder_main_24(int j, int length, int nig, int
      counter, StringBuilder b, ArrayList<Integer> list){}
  public static void check_absentremainder_main_18(int j, int length, ArrayList<Integer>
      list, int nig){}
  public static void check_absentremainder_main_9(int i, int numCases, BufferedReader in,
      StringBuilder b){}
}
// predict invariants for absentremainder.main.ENTER(String[] args),
    absentremainder.check_absentremainder_main_9.ENTER(int i, int numCases,
    BufferedReader in, StringBuilder b),
    absentremainder.check_absentremainder_main_18.ENTER(int j, int length,
    ArrayList<Integer> list, int nig)
absentremainder.main.ENTER(String[] args)
```

Listing 3: Scratchpad input string for solution s300 to problem 1613B. As the model size increases, additional invariants are uncovered. The invariants predicted by the scratchpad approach at all model sizes are shown in the table.

Invariants	Target	430M	1B	5B
`j >= 0`	✓	✓	✓	✓
`length > nig`	✓	✗	✗	✓
`j < length`	✓	✓	✓	✓
`nig one of { 0, 1, 2 }`	✗	✓	✗	✗
`j >= nig`	✓	✗	✓	✓
`length one of { 2, 6 }`	✗	✗	✓	✗
`nig >= 0`	✓	✗	✓	✓
Jaccard	1.00	0.33	0.67	1.00
Precision	1.00	0.67	0.80	1.00
Recall	1.00	0.40	0.80	1.00
F1 Score	1.00	0.50	0.80	1.00

```java
import java.io.*;
import java.util.StringTokenizer;

public class Main {
  public static class FastInput {
    private BufferedReader br;
    private StringTokenizer st;
    public FastInput() {
      br = new BufferedReader(new InputStreamReader(System.in));
    }
    public String next() {
      while (st == null || !st.hasMoreTokens()) {
        try {
          st = new StringTokenizer(br.readLine());
        } catch (IOException obj) {
          System.out.println(obj);
        }
      }
      return st.nextToken();
    }
    public int nextInt(){
        return Integer.parseInt(next());
    }
    ...
  }
  // public static final int MOD = 1000000007;
  public static void main(String[] args) throws IOException {
    FastInput input = new FastInput();
    FastOutput out = new FastOutput();
    int test = input.nextInt();
    while (test-- > 0) {
      check_Main_main_78(test, input, out);
      long n = input.nextLong();
      if ((n & 1) == 0) {
        long last = n;
        long fast = (n - 1) * (-1);
        out.println(fast + " " + last);
      } else {
        long fast = n / 2;
        long last = (n / 2) + 1;
        out.println(fast + " " + last);
      }
      out.flush();
    }
    out.close();
  }
  public static void check_Main_main_78(int test, FastInput input, FastOutput out){}
}
// predict invariants for Main$FastInput.Object(), Main$FastInput.nextInt.EXIT()
Main$FastInput.Object()
```

Listing 4: Source code for solution s290 to problem 1594A

Invariants	Target	Direct	Scratchpad	Oracle
return == 7	✓	✗	✗	✗
this.st has only one value	✓	✓	✓	✓
this.br == orig(this.br)	✓	✓	✓	✓
return >= 1	✗	✓	✓	✓
Jaccard	1.00	0.50	0.50	0.50
Precision	1.00	0.67	0.67	0.67
Recall	1.00	0.67	0.67	0.67
F1 Score	1.00	0.67	0.67	0.67

```java
import java.io.*;
import java.util.*;

public class C {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int t = sc.nextInt();
    while (t-- > 0) {

      check_C_main_7(t, sc);
      int n = sc.nextInt();
      int k = sc.nextInt();
      if (k == 1) {
        System.out.println(n - 1);
      } else {
        int test = n;
        int mul = 1;
        test -= 2;
        int chk = 0;
        while (test > 0) {

          check_C_main_17(test, mul, k, chk);
          mul *= 2;
          if (mul <= k) {
            test -= mul;
          } else {
            test -= k;
          }
          chk++;
        }
        System.out.println(chk + 1);
      }
    }
  }

  public static void check_C_main_17(int test, int mul, int k, int chk){}

  public static void check_C_main_7(int t, Scanner sc){}
}

// predict invariants for C.main.ENTER(String[] args), C.check_C_main_7.ENTER(int t,
    Scanner sc), C.check_C_main_17.ENTER(int test, int mul, int k, int chk)
C.main.ENTER(String[] args)
```

Listing 5: Source code for solution s186 to problem 1606B

INVARIANTS	TARGET	DIRECT	SCRATCHPAD	ORACLE
`mul >= 1`	✓	✓	✓	✓
`test >= 1`	✗	✓	✓	✓
`mul is a power of 2`	✓	✓	✓	✓
`chk >= 0`	✓	✓	✓	✓
`k >= 2`	✓	✓	✓	✓
JACCARD	1.00	0.80	0.80	0.80
PRECISION	1.00	0.80	0.80	0.80
RECALL	1.00	1.00	1.00	1.00
F1 SCORE	1.00	0.89	0.89	0.89

```java
import java.util.*;
public class Grass {
  public static long helper(long x0, long n) {
    if (Math.abs(x0) % 2 != 0) {
      if (n % 2 == 0) {
        if (n % 4 == 0) {
          return x0;
        } else {
          return x0 - 1;
        }
      } else {
        if ((n - 1) % 4 == 0) {
          return x0 + n;
        } else
          return x0 - n - 1;
      }
    } else {
      if (n % 2 == 0) {
        if (n % 4 == 0) {
          return x0;
        } else
          return x0 + 1;
      } else {
        if ((n - 1) % 4 == 0) {
          return x0 - n;
        } else
          return x0 + n + 1;
      }
    }
  }
  public static void main(String[] args) {
    Scanner sr = new Scanner(System.in);
    long tc = sr.nextLong();
    while (tc-- > 0) {
      check_Grass_main_33(tc, sr);
      long x0 = sr.nextLong(), n = sr.nextLong();
      long val = helper(x0, n);
      System.out.println(val);
    }
  }
  public static void check_Grass_main_33(long tc, Scanner sr){}
}
// predict invariants for Grass.helper.ENTER(long x0, long n), Grass.helper.EXIT7(long
    x0, long n)
Grass.helper.ENTER(long x0, long n)
```

Listing 6: Source code for solution s98 to problem 1607B

Invariants	Target	Direct	Scratchpad	Oracle
return != orig(x0)	✗	✓	✗	✗
return one of { -1, 1, 3 }	✓	✗	✗	✗
return % orig(n) == 0	✗	✓	✗	✗
return == orig(x0)	✓	✗	✓	✓
return >= orig(x0)	✗	✓	✗	✗
return != 0	✓	✗	✗	✗
orig(x0) != orig(n)	✗	✓	✓	✓
return != orig(n)	✓	✗	✗	✗
return > orig(n)	✗	✓	✗	✗
orig(n) % return == 0	✓	✗	✗	✗
orig(n) one of { 0, 4 }	✓	✗	✗	✗
Jaccard	1.00	0.00	0.14	0.14
Precision	1.00	0.00	0.50	0.50
Recall	1.00	0.00	0.17	0.17
F1 Score	1.00	0.00	0.25	0.25

```java
import java.util.*;

public class cf762 {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    cf762 ob = new cf762();
    int n = sc.nextInt();
    for (int i = 0; i < n; i++) {
      System.out.println(square(sc.next()));
    }
  }

  static String rev(String s) {
    String reverse = "";
    for (int i = s.length() - 1; i >= 0; i--) {
      reverse += s.charAt(i);
    }
    return (reverse);
  }

  static String square(String s) {
    if (s.length() % 2 != 0) {
      return ("NO");
    } else {
      int spl = s.length() / 2;
      int z = 0, x = s.length() - 1;
      String str1 = "", str2 = "";
      while (z <= spl && x >= spl) {
        check_cf762_square_27(z, spl, x, str1, str2);
        str1 += s.charAt(z);
        str2 += s.charAt(x);
        z++;
        x--;
      }
      System.out.println("Str1 = " + str1);
      System.out.println("Str2 = " + rev(str2));
      str2 = rev(str2);
      if (str1.equals(str2)) {
        return ("YES");
      } else
        return ("NO");
    }
  }
  static void check_cf762_square_27(int z, int spl, int x, String str1, String str2){}
}
// predict invariants for cf762.Object(), cf762.square.EXIT23(String s),
    cf762.check_cf762_square_27.ENTER(int z, int spl, int x, String str1, String str2)
cf762.Object()
```

Listing 7: Source code for solution s234 to problem 1619A

Invariants	Target	Direct	Scratchpad	Oracle
`spl <= x`	✓	✓	✓	✓
`z one of { 0, 1 }`	✗	✓	✗	✓
`z <= x`	✗	✓	✓	✓
`z < spl`	✓	✗	✗	✗
`z <= spl`	✗	✓	✓	✓
`z one of { 0, 1, 2 }`	✗	✗	✓	✗
`z < x`	✓	✗	✗	✗
`z - 2 * spl + x + 1 == 0`	✓	✗	✗	✗
`z >= 0`	✓	✗	✗	✗
`spl >= 1`	✓	✗	✗	✗
`spl one of { 1, 2, 3 }`	✗	✓	✓	✓
`x >= 1`	✓	✗	✗	✗
Jaccard	1.00	0.09	0.09	0.09
Precision	1.00	0.20	0.20	0.20
Recall	1.00	0.14	0.14	0.14
F1 Score	1.00	0.17	0.17	0.17

```java
import java.util.*;
public class test {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    int t = in.nextInt();
    while (t-- > 0) {

      check_test_main_5(t, in);
      long n = in.nextLong();
      long m = in.nextLong();
      long p = m - (m % 4) + 1;
      while (p <= m) {

        check_test_main_9(p, m, n);
        if (n % 2 == 0) {
          n -= p;
        } else {
          n += p;
        }
        p++;
      }
      System.out.println(n);
    }
  }

  public static void check_test_main_9(long p, long m, long n){}

  public static void check_test_main_5(int t, Scanner in){}
}
// predict invariants for test.check_test_main_9.ENTER(long p, long m, long n)
```

Listing 8: Input string for the direct approach for solution s288 to problem 1607B. In the direct approach, the model predicts invariants only for the target program point, not any of the other program points. The other program points are still annotated in the source, as shown.

```
test.check_test_main_9.ENTER(long p, long m, long n)
+++++
orig
=====
null
=====
one_val
=====
comp
m >= 1
p <= m
p >= 1
=====
val_set
=====
arithmetic
=====
power
=====
divides
=====
square
=====
elt
=====
elt_oneof
=====
eltwise
=====
member
=====
sub
=====
seqseq
=====
reverse
=====
agg
=====
conditional
-----
```

Listing 9: Target string for the direct approach for solution s288 to problem 1607B. All invariant types are listed in the order given in Table 2, even those invariant types which have no invariants at the named program point.

```
import java.util.*;
public class test {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    int t = in.nextInt();
    while (t-- > 0) {

      check_test_main_5(t, in);
      long n = in.nextLong();
      long m = in.nextLong();
      long p = m - (m % 4) + 1;
      while (p <= m) {

        check_test_main_9(p, m, n);
        if (n % 2 == 0) {
          n -= p;
        } else {
          n += p;
        }
        p++;
      }
      System.out.println(n);
    }
  }

  public static void check_test_main_9(long p, long m, long n){}

  public static void check_test_main_5(int t, Scanner in){}
}
// predict invariants for test.main.ENTER(String[] args),
    test.check_test_main_5.ENTER(int t, Scanner in), test.check_test_main_9.ENTER(long p,
    long m, long n)
test.main.ENTER(String[] args)
```

Listing 10: Input string for the scratchpad approach for solution s288 to problem 1607B. After the annotated source code, the directive shows the order of program points to predict invariants for, ending with the target program point. The input string concludes with the first program point to generate invariants for.

```
+++++
orig
=====
null
args[] == []
=====
one_val
args has only one value
=====
comp
=====
val_set
=====
arithmetic
=====
power
=====
divides
=====
square
=====
elt
=====
elt_oneof
=====
eltwise
=====
member
=====
sub
=====
seqseq
=====
reverse
=====
agg
=====
conditional
-----
test.check_test_main_5.ENTER(int t,
    Scanner in)
+++++
orig
=====
null
=====
one_val
in has only one value
=====
comp
t >= 0
=====
val_set
=====
arithmetic
=====
power
=====
divides
=====
square
=====
elt
=====
elt_oneof
=====
eltwise
=====
member
=====
sub
=====
seqseq
=====
reverse
=====
agg
=====
conditional
-----
test.check_test_main_9.ENTER(long p, long
    m, long n)
+++++
orig
=====
null
=====
one_val
=====
comp
m >= 1
p <= m
p >= 1
=====
val_set
=====
arithmetic
=====
power
=====
divides
=====
square
=====
elt
=====
elt_oneof
=====
eltwise
=====
member
=====
sub
=====
seqseq
=====
reverse
=====
agg
=====
conditional
-----
```

Listing 11: Target string for the scratchpad approach for solution s288 to problem 1607B. All invariant types are listed, even those without invariants. Invariants are given for program points in the order given in the input string.