SAM2INV: LLM-Driven Loop Invariant Synthesis via Dynamic Sampling and Iterative Refinement

Anonymous
Anonymous Institution
anonymous@example.com

Abstract

Automatically synthesizing loop invariants remains a central challenge in formal software verification. We present SAM2INV, an end-to-end system that leverages large language models (LLMs) to generate ACSL loop invariant annotations for C programs, enabling fully automated verification with Frama-C/WP. Our approach combines three key ideas: (1) a *smart dynamic sampling* strategy that executes the program under test with carefully tiered inputs to collect execution traces that expose loop behaviour; (2) *parallel diverse generation*, where multiple LLM calls with varied prompts and temperatures produce a rich candidate pool; and (3) a *Houdini-style iterative pruning* loop that eliminates failing candidates and feeds verification feedback back to the LLM for targeted repair. To facilitate large-scale evaluation, we further introduce a *probabilistic DSL* for synthesizing structured numeric loop programs with controllable complexity.

1 Introduction

Loop invariants are the cornerstone of deductive program verification. Given a loop annotated with an appropriate invariant, a verification condition generator such as Frama-C/WP [12] can reduce the correctness of the entire program to a set of first-order proof obligations that can be discharged automatically. In practice, however, *finding* the right invariant is the bottleneck: even expert users spend substantial effort crafting invariants that are simultaneously *inductive* (preserved by each iteration), *sufficient* (strong enough to imply the postcondition), and *syntactically admissible* (expressible in the annotation language).

Classical approaches to invariant synthesis fall into two broad camps. *Static* techniques such as abstract interpretation [5], template-based constraint solving [4], and Houdini-style iterative weakening [8] offer soundness guarantees but are limited to fixed abstract domains or templates. *Dynamic* techniques infer candidate invariants from execution traces using polynomial fitting [7], machine learning [18], or data-driven enumerative search [9], but must still close the gap between empirical observation and formal proof.

Recently, large language models (LLMs) have shown remarkable ability in code understanding and generation tasks. Several studies have explored using LLMs to generate loop invariants [2, 11, 16], demonstrating that LLMs can produce plausible candidates even for non-trivial programs. However, existing approaches typically rely on a single LLM call or a simple repair loop, and do not systematically exploit execution traces or diversified generation strategies.

In this paper we present SAM2INV (**Sam**pling **to Inv**ariant), an end-to-end system that combines the strengths of dynamic sampling, LLM-based generation, and formal verification into a tightly integrated pipeline. Our key contributions are:

(1) **Smart Dynamic Sampling.** We propose a tiered sampling strategy that executes the target program with carefully ordered inputs—from special boundary values through small, medium, and large integers—to collect execution traces that expose loop behaviour at different scales. The traces are formatted into structured prompts that guide the LLM toward discovering mathematical relationships among loop variables.

(2) **Parallel Diverse Generation.** Rather than relying on a single LLM query, we issue multiple parallel requests with varied prompt templates and elevated sampling temperatures. This produces a diverse candidate pool from which invariants are selected through trace-based filtering and Houdini-style pruning.

(3) **Iterative Verification–Repair Loop.** Candidates that survive initial filtering are verified with Frama-C/WP. Failing invariants are pruned in a Houdini fashion (guaranteed to terminate), and verification error messages are fed back to the LLM for targeted strengthening, closing the loop between generation and formal proof.

(4) **Probabilistic Loop Synthesis DSL.** To enable controlled, large-scale evaluation beyond existing benchmarks, we design a probabilistic domain-specific language (DSL) that generates structured numeric loop programs with tunable arithmetic complexity, nesting depth, and variable counts.

The remainder of this paper is organised as follows. Section 2 surveys related work. Section 3 gives a high-level overview of the SAM2INV pipeline. Section 4 details each component. Section 5 describes the loop synthesis DSL. Section 6 presents experimental evaluation. Section 7 concludes.

2 Related Work

Classical Invariant Synthesis. Abstract interpretation [5] computes fixed-point approximations over abstract domains (intervals, octagons, polyhedra) and has been widely adopted in industrial static analysers. Template-based approaches [4, 17] reduce invariant synthesis to constraint solving by fixing a parametric template and searching for satisfying coefficients. The Houdini algorithm [8] takes the dual approach: it starts with a large candidate set and iteratively removes conjuncts that are not inductive, converging to the strongest conjunction that is preserved by the loop. While sound, these

methods are inherently limited by the expressiveness of their abstract domains or template families.

Dynamic Invariant Detection. Daikon [7] pioneered the idea of mining likely invariants from execution traces by instantiating templates over observed variable values. Subsequent work extended this idea with more expressive templates, statistical hypothesis testing, and machine-learning classifiers [9]. DIG [14] combines dynamic analysis with symbolic execution to discover non-linear polynomial invariants. These approaches are effective at generating plausible candidates but provide no formal guarantee; a separate verification step is required to confirm inductiveness.

LLM-Based Invariant Generation. The emergence of large language models has opened a new avenue for invariant synthesis. Chakraborty and Lahiri [2] demonstrate that GPT-4 can generate loop invariants for simple programs when given appropriate prompts and a verification feedback loop. Pei et al. [16] study the ability of code-trained LLMs to produce Hoare-style annotations, finding that models can often produce correct invariants but benefit significantly from iterative refinement. Kamath et al. [11] propose a "guess-and-check" framework that combines LLM generation with bounded model checking. Compared to these works, SAM2INV introduces three distinguishing features: (i) systematic use of execution traces from smart dynamic sampling to ground the LLM's generation; (ii) parallel diverse generation with multiple prompts and temperatures; and (iii) a Houdini-pruning integration that provides monotonic progress guarantees within the repair loop.

Program Synthesis with LLMs. More broadly, LLMs have been applied to a variety of synthesis tasks including code generation [3], test generation [13], and specification inference [6]. Our work shares the spirit of using LLMs as proposal generators within a verify-and-refine loop, but targets the specific domain of ACSL loop invariants and integrates domain-specific filtering (trace validation, ACSL syntax checking) to dramatically reduce the verification burden.

Benchmark Generation for Verification. Existing invariant synthesis benchmarks—such as the NLA (Non-Linear Arithmetic) suite from SV-COMP [1]—contain only a few dozen programs. To enable larger-scale evaluation, we introduce a probabilistic DSL for synthesizing numeric loop programs, drawing inspiration from grammar-based fuzzing [10] and probabilistic program synthesis [15].

3 System Overview

Figure 1 illustrates the end-to-end SAM2INV pipeline. The system takes as input a C function containing a `while` loop, a precondition (`requires`), and a postcondition (`assert`). It outputs the same function annotated with ACSL loop invariants and a `loop assigns` clause, such that Frama-C/WP can fully verify the postcondition.

The pipeline operates in three broad phases:

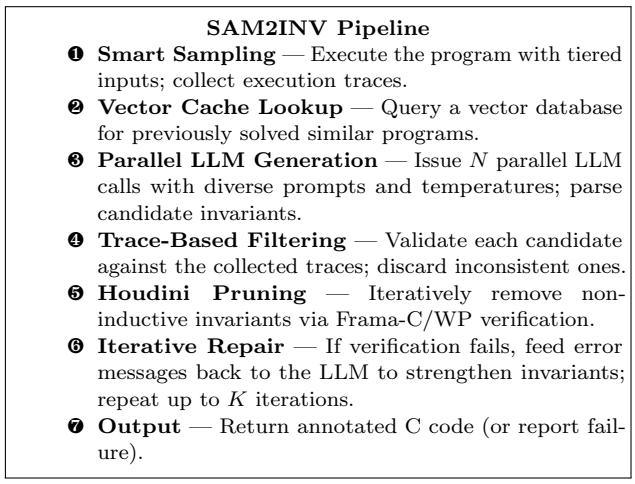

Figure 1: High-level pipeline of SAM2INV. Steps ❺–❻ form a feedback loop that is guaranteed to terminate.

Phase I: Trace Collection (Steps ❶–❷). The target program is compiled and executed under a set of carefully chosen inputs produced by the *smart sampler* (Section 4.1). Execution traces—recording variable values at every loop iteration—are collected and formatted into structured text. Before invoking the LLM, the system also queries a vector database (ChromaDB) to retrieve solutions for similar programs seen in prior runs, providing few-shot context.

Phase II: Generation and Filtering (Steps ❸–❹). Multiple LLM instances are queried in parallel, each receiving a prompt assembled from the source code, the execution traces, a system prompt encoding ACSL rules, and (optionally) cached examples. Prompt diversity is achieved by varying the template and the sampling temperature. The resulting candidate invariants are first checked for ACSL syntax compliance, then validated against the execution traces: any invariant that is falsified by at least one observed state is immediately discarded.

Phase III: Verification and Repair (Steps ❺–❼). Surviving candidates are combined and verified by Frama-C/WP. If some invariants fail, the Houdini pruner removes the failing subset and re-verifies; this process is guaranteed to terminate because at least one invariant is removed per iteration. If the pruned set is still insufficient to prove the postcondition, the system enters an *iterative repair* loop: verification error messages are sent to the LLM, which proposes strengthened or additional invariants. The repair loop runs for at most K iterations (configurable; default K=10).

Running Example. Consider the following program that computes the cube of n:

```
1 /*@ requires a>=n && n==0; */
2 int main1(int a, int n){
3   int x, y, z;
```

```
  x=0; y=1; z=6;
  while(n <= a){
    n=n+1; x=x+y; y=y+z; z=z+6;
  }
  /*@ assert (n==a+1) && (y==3*n*n+3*n+1)
           && (x==n*n*n) && (z==6*n+6); */
}
```

The smart sampler executes this program with $a \in \{0, 1, \ldots, 10\}$ and collects traces such as:

iter	n	x	y
0	0	0	1
1	1	1	7
2	2	8	25

From these traces, the LLM discovers the relationships $x = n^3$, $y = 3n^2 + 3n + 1$, $z = 6n + 6$, and $n \leq a + 1$. After Houdini pruning confirms all four invariants are inductive, Frama-C/WP verifies the postcondition in a single pass.

4 Method

We now describe each component of SAM2INV in detail.

4.1 Smart Dynamic Sampling

The purpose of dynamic sampling is to collect execution traces that expose the mathematical relationships maintained by the loop. Naive random sampling often produces inputs that are either too simple (all zeros) or too large (causing integer overflow or time-outs). We therefore adopt a *tiered* sampling strategy that prioritises simple, informative values while gradually introducing diversity.

Definition 4.1 (Value Tiers). Given a parameter variable v with domain $[l, u]$, we define four value tiers:

$$\mathcal{T}_0 = \{0, 1, -1\} \cap [l, u] \quad \text{(special values)}$$
$$\mathcal{T}_1 = \{2, 3, \ldots, 10\} \cap [l, u] \quad \text{(small integers)}$$
$$\mathcal{T}_2 \subseteq \{11, \ldots, 50\} \cap [l, u] \quad \text{(medium, sampled)}$$
$$\mathcal{T}_3 \subseteq \{51, \ldots, 100\} \cap [l, u] \quad \text{(large, sampled)}$$

Sampling proceeds in phases:

(1) **Phase 1:** Generate the Cartesian product over $\mathcal{T}_0$ for all parameters. These "corner cases" are ideal for fitting low-degree polynomials.
(2) **Phase 2:** Mix $\mathcal{T}_0$ and $\mathcal{T}_1$ to add coverage of small positive integers.
(3) **Phase 3:** Include all tiers, prioritising combinations with lower total tier index.
(4) **Phase 4:** Fill remaining quota with biased random sampling (80% probability of choosing values ≤ 20).

Each input is executed, and the variable state at every loop iteration is recorded. To keep prompts within LLM context limits, we retain only the first and last k iterations per run (default k=3) and group traces into at most G groups (default G=10).

4.2 LLM-Based Invariant Generation

Prompt Design. Each LLM prompt is assembled from four components:

(1) A *system prompt* that defines the task (loop invariant synthesis for Frama-C/WP), lists allowed and forbidden ACSL constructs, and encodes the assertion-driven synthesis strategy (decompose postcondition conjuncts, add boundary invariants, preserve unmodified parameters).
(2) The *source code* of the target function.
(3) *Execution traces* formatted as structured variable–value tables (Section 4.1).
(4) (Optional) *Cached examples* retrieved from a ChromaDB vector store, providing few-shot demonstrations of similar solved programs.

Assertion-Driven Strategy. A key insight is that the postcondition assertion directly suggests invariant candidates:

(1) Each conjunct of the assertion (split on `&&`) is a candidate invariant.
(2) For a loop guard of the form `i < n`, the weakened form `i <= n` is an inductive boundary invariant.
(3) Function parameters not modified in the loop body are preserved: `v == \at(v, Pre)`.

Parallel Diverse Generation. To maximise the chance of finding the correct invariant set, we issue N parallel LLM calls (default N=5) using a thread pool. Diversity is introduced along two axes:

- **Temperature.** We use $\tau \geq 1.0$ to encourage diverse completions.
- **Prompt variation.** Different calls may use different prompt templates or include/exclude cached examples.

All N responses are parsed, and the union of candidate invariants forms the initial candidate pool.

4.3 Trace-Based Filtering

Before invoking the expensive Frama-C verifier, we apply lightweight filters to eliminate clearly incorrect candidates.

Syntax Filter. Each candidate is checked against ACSL syntax rules:

- No quantifiers (`\forall`, `\exists`).
- No custom definitions (`predicate`, `logic`, `lemma`).
- No ternary operator (`? :`).
- `\at(v, Pre)` only on function parameters.
- No exponentiation (`^` is bitwise XOR in ACSL); use repeated multiplication instead.

Semantic Filter. Each candidate invariant is evaluated against the collected execution traces. A candidate that is *falsified* by any observed loop state is discarded. While this does not guarantee inductiveness (the traces are finite), it eliminates a large fraction of incorrect candidates cheaply.

4.4 Houdini Pruning

After filtering, the surviving candidate set is verified by Frama-C/WP. Typically, some candidates fail because they are not inductive (they hold at every observed state but are not preserved by one more iteration from an arbitrary state).

Algorithm: Houdini Pruning
Input: Candidate set C, verifier $\mathcal{V}$
Output: Maximal inductive subset $C^* \subseteq C$
$C^* \leftarrow C$
repeat
  $R \leftarrow \mathcal{V}(C^*)$ *// verify each invariant*
  $F \leftarrow \{c \in C^* \mid R(c) = \mathit{fail}\}$
  $C^* \leftarrow C^* \setminus F$
until $F = \emptyset$ **or** $C^* = \emptyset$
return C^*

Figure 2: Houdini-style pruning algorithm.

We apply the Houdini algorithm [8]:

Theorem 4.2 (Termination). *Algorithm* **??** *terminates in at most $|C|$ iterations, since each iteration removes at least one candidate.*

The Houdini pruner is implemented as a standalone module that depends only on Frama-C, not on the LLM. It can therefore be applied to *any* set of candidate invariants, regardless of their source.

4.5 Iterative Repair

If the Houdini-pruned set is non-empty but insufficient to prove the postcondition (i.e., Frama-C/WP reports Unknown for the assertion goal), the system enters an iterative repair loop:

(1) Extract the Frama-C error messages, identifying which proof obligations failed and why.
(2) Construct a *repair prompt* containing the current annotated code, the error messages, and instructions to strengthen or add invariants.
(3) Send the repair prompt to the LLM and parse the response.
(4) Apply the updated invariants and re-run Houdini pruning and verification.

This loop runs for at most K iterations (default K=10). Each iteration either makes progress (by adding or strengthening invariants) or exhausts the budget, at which point the system reports failure.

4.6 Generation Modes

SAM2INV supports three generation modes, selectable via configuration:

`code_only`**:** The LLM receives only the source code and system prompt. Execution traces are not used.
`fit_only`**:** The LLM receives only execution traces and is asked to discover mathematical relationships (the LLMFit module). The source code is not shown.
`hybrid` **(default):** The LLM first analyses traces via LLMFit to discover candidate relationships. These are injected as hints into the code-generation prompt, combining trace-based insight with code-level context.

Table 1: Hyperparameters of the LoopFactory DSL.

Symbol	Meaning
m	Max variables
p	Parameter (immutable) variables
n	Top-level loops
k	Max assignments per loop body
$D_{\max}$	Max nesting depth
π_{op}	Distribution over $\{+, -, \times, /, \text{mod}\}$
π_{cmp}	Distribution over $\{<, \leq, >, \geq, =, \neq\}$
π_{const}	Prob. of sampling a constant operand
π_{self}	Prob. of self-update ($v_i := v_i$ `op` x)
π_{nest}	Prob. of generating a nested sub-loop

The `hybrid` mode consistently outperforms the other two (Section 6), confirming that traces and code provide complementary signals.

4.7 Vector Cache

To amortise effort across runs, SAM2INV maintains a ChromaDB vector store that indexes solved programs by their code embeddings. When a new program arrives, the system queries the store for the k most similar programs (by cosine similarity) and includes their invariant annotations in the prompt as few-shot examples. After a successful verification, the new program–invariant pair is added to the store. This simple caching mechanism provides two benefits: (1) it reduces LLM query cost for programs similar to previously solved ones, and (2) it provides the LLM with concrete, verified examples that improve generation accuracy.

5 Probabilistic Loop Synthesis

Existing benchmarks for loop invariant synthesis (e.g., the NLA suite) contain only a few dozen programs, making it difficult to evaluate invariant synthesis tools at scale or to study the effect of program complexity on synthesis success. To address this, we design LoopFactory, a probabilistic domain-specific language (DSL) for generating structured numeric loop programs with controllable complexity.

5.1 Hyperparameters

The generator is parameterised by:

$$\theta = (m,\ p,\ n,\ k,\ D_{\max},\ \pi_{\text{op}},\ \pi_{\text{cmp}},\ \pi_{\text{const}},\ \pi_{\text{self}},\ \pi_{\text{nest}})$$

where m is the maximum number of variables, p the number of (immutable) parameter variables, n the number of top-level loops, k the maximum assignments per loop body, $D_{\max}$ the maximum nesting depth, and the remaining symbols are probability distributions governing operator choice, constant injection, self-update assignment, and loop nesting (Table 1).

5.2 DSL Syntax

Expressions. Arithmetic expressions are of the form:

$$e \ ::= \ v \ \mid \ c \ \mid \ v \ \texttt{op} \ x$$

where $v \in \mathcal{V}$ is a variable, $c \in \mathbb{Z}$ is an integer constant, $x \in \mathcal{V} \cup \mathbb{Z}$, and $\texttt{op} \in \{+, -, \times, /, \text{mod}\}$. If $\texttt{op} = \text{mod}$, the right operand must be a positive integer constant.

Boolean Guards. Loop guards are comparisons e_1 $\texttt{cmp}$ e_2 with $\texttt{cmp} \in \{<, \leq, >, \geq, =, \neq\}$.

Assignments. An assignment $v := e$ requires $v \in \mathcal{L}$, where $\mathcal{L} = \mathcal{V} \setminus \mathcal{P}$ is the set of writable (non-parameter) variables.

5.3 Program Structure

A generated program has the form:

$$\textbf{Prog} \ ::= \ \textbf{Init} \ ; \ \textbf{LoopForest}$$

The **Init** block assigns every writable variable exactly once, using expressions that depend only on parameter variables, ensuring a well-defined initial state.

The **LoopForest** is a sequence of loop trees $\mathcal{F} = [T_1, \ldots, T_n]$, where each tree node

$$T = \langle\, b,\ S,\ \mathcal{C} \,\rangle$$

consists of a guard b, an assignment list S with $|S| \leq k$, and a (possibly empty) list of child loops $\mathcal{C}$. Nesting depth is bounded by $D_{\max}$.

5.4 Semantic Loop Templates

To ensure that generated loops exhibit mathematically interesting behaviour (rather than trivial or divergent patterns), the DSL includes twelve *semantic templates* that encode common loop idioms:

(1) Cubic growth recursion (x, y, z coupled updates)
(2) Geometric recursion ($x = x \cdot z + 1$)
(3) Cumulative sum / dot product
(4) Power-sum accumulation
(5) Quotient–remainder counting
(6) Binary multiplication style
(7) Linear dual-variable shifting
(8) Linear decrement to zero
(9) Modulo bucket counting
(10) Euclidean-style subtraction
(11) Nested triangular sums
(12) Nested affine updates

Each template defines a fixed update pattern but allows the generator to sample concrete operators and constants from the probabilistic model, yielding a family of programs per template.

5.5 Generative Distribution

The probability of a complete program factorises as:

$$\Pr(\textbf{Prog}) = \prod_{v \in \mathcal{L}} \Pr(v := e_v) \ \cdot \ \prod_{T \in \mathcal{F}} \Pr(T) \qquad (1)$$

where, recursively,

$$\Pr(T) = \Pr(b) \cdot \prod_{s \in S} \Pr(s) \cdot \prod_{T' \in \mathcal{C}} \Pr(T').$$

This factorisation ensures that the generator defines a proper probability distribution over the space of well-formed loop programs, enabling statistical analysis of benchmark difficulty and ablation over individual hyperparameters.

Expected Properties. Under the Bernoulli nesting model, the expected nesting depth is:

$$\mathbb{E}[\text{depth}] = \sum_{d=0}^{D_{\max}} d \cdot (1 - \pi_{\text{nest}})\, \pi_{\text{nest}}^{d}$$

and the expected number of loop nodes is $\sum_{d=0}^{D_{\max}} \pi_{\text{nest}}^{d}$. These closed-form expressions allow practitioners to tune π_{nest} and $D_{\max}$ to obtain a desired distribution of benchmark difficulty.

6 Experiments

7 Conclusion

We have presented SAM2INV, an end-to-end system for automatic loop invariant synthesis that tightly integrates large language models with dynamic sampling and formal verification. The three pillars of our approach—smart tiered sampling, parallel diverse generation, and Houdini-style iterative pruning—address complementary weaknesses of prior LLM-based methods: sampling grounds the LLM in concrete program behaviour, diversity increases the probability of generating the correct invariant, and Houdini pruning provides monotonic progress guarantees within the verification loop.

Our probabilistic loop synthesis DSL, LoopFactory, enables controlled large-scale evaluation beyond the small existing benchmarks, and its factorised generative distribution allows principled ablation over program complexity dimensions.

Several directions remain for future work. First, extending the system to handle pointer-manipulating programs and separation-logic invariants (via Coq/VST integration, which our codebase already partially supports) would broaden the applicability of the approach. Second, replacing the prompt-based LLM interaction with fine-tuned models trained on verified invariant datasets may improve both accuracy and efficiency. Third, integrating symbolic techniques—such as abstract interpretation for computing initial bounds or interpolation for strengthening failing invariants—could provide a hybrid symbolic–neural architecture with stronger guarantees.

References

[1] Dirk Beyer. 2023. Competition on Software Verification (SV-COMP). https://sv-comp.sosy-lab.org/.
[2] Saikat Chakraborty, Shuvendu K. Lahiri, Sarah Fakhoury, Madanlal Musuvathi, Akash Lal, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2024. Towards Neural Synthesis for SMT-Assisted Proof-Oriented Programming. In *Proc. Int. Conf. on Learning Representations (ICLR)*.
[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*.
[4] Michael A. Colón, Sriram Sankaranarayanan, and Henny B. Sipma. 2003. Linear Invariant Generation Using Non-linear Constraint Solving. In *Proc. Int. Conf. on Computer Aided Verification (CAV)*. Springer, 420–432.

[5] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. 4th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages (POPL)*. 238–252.

[6] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. Can Large Language Models Write Specifications?. In *Proc. ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (FSE)*.

[7] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon System for Dynamic Detection of Likely Invariants. In *Science of Computer Programming*, Vol. 69. 35–45.

[8] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *Proc. Int. Symp. on Formal Methods Europe (FME)*. Springer, 500–517.

[9] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning Invariants Using Decision Trees and Implication Counterexamples. In *Proc. 43rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*. 499–512.

[10] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based Whitebox Fuzzing. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. 206–215.

[11] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. 2023. Finding Inductive Loop Invariants using Large Language Models. *arXiv preprint arXiv:2311.07948*.

[12] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A Software Analysis Perspective. In *Proc. Int. Conf. on Software Engineering and Formal Methods (SEFM)*. Springer, 233–247.

[13] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *Proc. 45th Int. Conf. on Software Engineering (ICSE)*. 919–931.

[14] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using Dynamic Analysis to Discover Polynomial and Array Invariants. In *Proc. 34th Int. Conf. on Software Engineering (ICSE)*. 1076–1086.

[15] Aditya V. Nori, Sherjil Ozair, Sriram K. Rajamani, and Deepak Viber. 2015. Efficient Synthesis of Probabilistic Programs. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. 208–217.

[16] Kexin Pei, David Biber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can Large Language Models Reason about Program Invariants?. In *Proc. Int. Conf. on Machine Learning (ICML)*.

[17] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. 2004. Non-linear Loop Invariant Generation Using Gröbner Bases. In *Proc. 31st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*. 318–329.

[18] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*. 7751–7762.