Mitigating Context Pollution in Neurosymbolic Widening via Slicing and Caching

VISHAL VUNNAM, University of Colorado Boulder, USA

Static Program Analysis is a cornerstone technique in software engineering for ensuring code reliability and security. Traditional static analysis, using abstract interpretation, represents the program as a set of mathematical constraints over abstract domains. These analyzers are sound, but often imprecise due to the undecidability of program termination, leading to excessive false positives. Recent work has explored using Language Models to improve the precision of widening-based static analyzers by predicting loop invariants and abstraction heuristics. However, these approaches explode in both context size and computation time as the number of program variables increases, making them infeasible for real-world programs. In this work, we propose SLICE-ABSINT, a novel approach to static analysis that leverages Language Models to guide the analysis process while maintaining scalability. By slicing the program based on the dependencies of diverging abstract states, we dynamically prune the context window to strictly relevant code paths before querying the neural oracle. This semantic filtering prevents "context pollution," ensuring the model focuses solely on variables affecting the widening decision.

1 INTRODUCTION

Software systems are becoming increasingly complex. As mission-critical backend applications migrate toward dynamic languages such as Python and JavaScript, the need for robust static analysis, for compiler optimizations, vulnerability detection, and formal verification—has never been greater. However, the dynamic features of these languages pose a fundamental challenge. Traditional static analysis techniques, such as Abstract Interpretation [3], provide soundness, the formal guarantee that the analysis will not miss any potential errors. Yet, to maintain this guarantee in the face of dynamic ambiguity, traditional analyzers must over-approximate the program's behavior. This loss of precision results in excessive false positives, often rendering the tools impractical for developers.

Recent work, specifically AbsInt-AI [4], has demonstrated that Language Models (LMs) can play a pivotal role in supporting sound static analysis. By leveraging the contextual and semantic understanding of LMs, these systems can predict precise heap abstractions and loop invariants that traditional heuristics miss. Crucially, ABSINT-AI was the first framework to integrate LMs without sacrificing soundness, using the model solely as a heuristic oracle while relying on the underlying abstract interpreter to mathematically verify the suggestions.

However, this approach faces a critical scalability barrier. Current methods naively present the LM with the entire program state to make a local decision. As the number of program variables increases, the context window becomes flooded with irrelevant code and data. We term this phenomenon "Context Pollution."

Context pollution leads to two failures: (1) Inefficiency, as token costs and latency explode linearly with program size rather than complexity, and (2) Imprecision, as the LM struggles to separate semantically relevant variables from noise, leading to degraded invariant predictions.

To address these challenges, we propose SLICE-ABSINT, a novel framework that integrates semantic program slicing directly into the LM-guided abstract interpretation loop.

Our approach differs fundamentally from the status quo. Instead of querying the LM with the full program state, SLICE-ABSINT analyzes the control and data dependencies of the diverging abstract state. It constructs a precise backward slice containing only the code segments that mathematically influence the variables of interest. This targeted approach mitigates context pollution by presenting the LM with a minimal, semantically potent context.

Author's address: Vishal Vunnam, vivu9928@colorado.edu, University of Colorado Boulder, USA.

Furthermore, to improve runtime and reduce token consumption, we propose a caching mechanism of the graph dependa

In summary, this paper makes the following contributions:

- **Methodology:** We propose SLICE-ABSINT, a framework that leverages semantic slicing to optimize the prompt engineering of neurosymbolic widening operators.
- **Soundness:** We formalize the slicing mechanism and prove that invariants derived from a valid program slice are observational equivalents of the full program, preserving the soundness of the verification.
- **Efficiency:** We demonstrate that our targeted context reduction significantly lowers token consumption and improves invariant prediction accuracy compared to full-context baselines.

2 OVERVIEW

3 MOTIVATING EXAMPLE

4 METHODOLOGY: ABSTRACT INTERPRETATION AND ABSINT-AI BACKGROUND

4.1 Abstract Interpretation Framework

Abstract Interpretation provides a general framework for approximating the semantics of a program. In this work, we adopt the standard conventions where concrete values $v \in \mathcal{V}$ are mapped to abstract values $\hat{v} \in \mathcal{A}$ via an abstraction function α, and abstract values are mapped back to sets of concrete values via a concretization function γ.

We define our specific abstract interpreter, AbsInt, as a state transition system operating on the abstract state tuple $\Sigma^\sharp = \langle H_L, H_G, \sigma \rangle$.

- **Local Heap (H_L):** A flow-sensitive mapping $H_L : \mathrm{Addr}_L \to \mathrm{Obj}^\sharp$, tracking objects allocated within the current lexical scope.
- **Global Heap (H_G):** A flow-insensitive mapping $H_G : \mathrm{Addr}_G \to \mathrm{Obj}^\sharp$, capturing the "soup" of globally visible objects and shared state, crucial for modeling the asynchronous event loops typical of dynamic languages.
- **Stack (σ):** A mapping $\sigma : \mathrm{Var} \to \mathrm{Val}^\sharp$, storing local variable bindings.

The abstract values $\mathrm{Val}^\sharp$ are defined over a lattice that includes disjoint domains for primitives and references:

$$\mathrm{Val}^\sharp ::= \bot \mid \top \mid \mathrm{Int}^\sharp \mid \mathrm{Bool}^\sharp \mid \mathrm{Ref}(\mathrm{Addr})$$

where $\mathrm{Int}^\sharp$ is the Interval domain $[l, u]$ with bounds in $\mathbb{Z} \cup \{-\infty, +\infty\}$.

4.2 Execution Phases of ABSINT-AI

The analysis proceeds in two distinct phases to balance precision and convergence.

Phase 1: Small-Step Execution. For standard linear statements (assignments, arithmetic), the interpreter executes in a flow-sensitive manner.

Phase 2: LM-Guided Summarization. When the analyzer encounters unbounded structures, such as ‘while‘ loops or recursion, standard fixed-point iteration may fail to converge within a reasonable time. Traditional static analyzers employ a widening operator (∇) to force convergence, often at the cost of precision (e.g., widening $[0, 1]$ directly to $[0, \infty]$).

AbsInt-AI replaces this blind widening with a *Semantic Summarization* step. At the loop head, the analyzer invokes an Oracle O_{LM} (a Language Model, we are running benchmars on Ollama's llama3). The oracle identifies diverging variables and suggests an abstraction strategy, either merging variables into primitive abstract domains or collapsing complex objects into summary nodes.

However, querying O_{LM} with the full abstract state $\Sigma^\sharp$ introduces *Context Pollution*. The inclusion of irrelevant variables noise-gates the model, reducing the accuracy of the invariant prediction and linearly increasing the cost of analysis.

5 [CONTRIBUTION 1] SEMANTIC SLICING (SAINT)

To mitigate context pollution, we introduce the **S**emantic **A**nalysis via **IN**cremental **T**runcation (SAINT) algorithm. The core of SAINT is a backward slicing operator that filters the program context before it reaches the LM.

5.1 Formal Definition of Slicing

I am defining the slice operator $\mathcal{S}$ as a backwards fixed-point calculation on the Program Dependence Graph (PDG) [5]. Given a program P, a location ℓ, and a set of diverging variables V_{div} identified by the interpreter, the slicer computes a program P_{slice} that is a dependency closure of the defined variables.

The algorithm tracjs a set of relevant statements P_{slice} and a set of tracked variables V_{trace} (initialized to V_{div}). Below are inference rules defining the iteration of the program towars a fixed point. In the appendix are more specific inference rules for various statement types.

Rule 1: Data Dependency . If a statement s defines a variable currently in the trace set, it must be included.

$$\frac{s \notin P_{slice} \quad \wedge \quad (\mathrm{Def}(s) \cap V_{trace} \neq \emptyset)}{(P_{slice}, V_{trace}) \longrightarrow (P_{slice} \cup \{s\}, \quad V_{trace} \cup \mathrm{Use}(s))}$$

This captures the flow of values. For example, if $x \in V_{trace}$ and the statement is $x := y + 1$, then the statement is added and y is added to V_{trace}.

Rule 2: Control Dependency. If a statement s' is already in the slice, and its execution is in the scope of a control statement. That control statement must be added to the program slice, along with it's conditional variable.

$$\frac{s \notin P_{slice} \quad \wedge \quad (\exists s' \in P_{slice} : s' \in \mathrm{Scope}(s))}{(P_{slice}, V_{trace}) \longrightarrow (P_{slice} \cup \{s\}, \quad V_{trace} \cup \mathrm{Use}(s))}$$

For example, if s' is inside an `if (b) { ... }` block and s' is in the slice, then the condition b must be added to the slice and its variables added to V_{trace}.

These rules can be applied iteratively, into two representation of the sub-program:

- **Dependancy Graph Representation** A PDG representing the control and data dependencies of the program.
- **A Rebuilt Syntactict Sub-Program** A syntactic representation of the program slice, rebuilt from the PDG by performing a topological sort on the dependency graph.

Optimally, the LM might perform better with the syntactic representation. Work can and should be done to compare the two representations.

5.2 The Summarization Process

The slicing operator is integrated into the summarization function as follows:

1. **Identification:** The abstract interpreter detects a loop at location ℓ and identifies variables V_{div} that have changed since the last iteration. Thinks that it should call summary to get invariant for these variables.
2. **Pruning:** Compute $P_{slice} = \mathcal{S}(P, \ell, V_{div})$. Or in other words, get the slice of the program relevant to the diverging variables at the loop head.

(3) **Projection:** Project the current abstract state $\Sigma^\sharp$ to restrict it to only the variables in V_{trace}. Let $\Sigma^\sharp_{proj} = \Sigma^\sharp \restriction V_{trace}$ (Only keep the variables in the trace set) .

(4) **Query:** Construct the prompt $Q = \langle P_{slice}, \Sigma^\sharp_{proj} \rangle$ and query $\mathcal{O}_{LM}(Q)$.

This process ensures that the LM receives a "minimal sound context"—the smallest subset of code and data required to mathematically derive the loop invariant.

6 [CONTRIBUTION 2] OPTIMIZATION VIA INCREMENTAL CACHING

While slciing seamingly reduces context pollution, resulting in a shorter, more precise prompt. There still exist two problems pertaining to runtime efficiency.

(1) **Slicing Overhead:** The slicing operation itself can be computationally expensive, especially for large codebases with complex dependency graphs. Recomputing the slice for every loop iteration can negate the efficiency gains from context reduction.

(2) **Redundant LM Queries:** We did not reduce the number of LM queries. In fact, if it is possible we made the LM more precise, this may lead to more frequent queries as the analysis converges slower.

I have considered two possible solutions to these problems: *Slice Caching* and Parrelelization through Indepen

$$P_{slice} : (\text{FuncID}, \ell, V_{div}) \longrightarrow P_{slice} \tag{1}$$

6.1 Invariant Caching

The volatility of the abstract state $\Sigma^\sharp$ typically prevents caching LM queries, as the specific values of variables change in every iteration. However, by asking the LM for *symbolic* invariants (e.g., "$i < 100$") rather than concrete next-steps, we decouple the query from the specific numeric values in $\Sigma^\sharp$.

We define a canonicalization function $\kappa(Q)$ that normalizes variable names and abstracts concrete values.

$$\text{Cache}_{LM} : \text{Hash}(\kappa(P_{slice})) \longrightarrow \text{Invariant}$$

If the structural logic of the loop slice remains unchanged, we retrieve the cached invariant, bypassing the LM entirely. This effectively reduces the asymptotic complexity of the analysis from $O(\text{queries} \times \text{latency})$ to $O(1)$ for previously analyzed code paths.

7 EVALUATION

7.1 Experimental Setup

We evaluate Slice-AbsInt against two baselines: a traditional Pure Abstract Interpreter (standard widening) and a Full-Context LM-Guided Abstract Interpreter (a slightly modified lean implementation of AbsInt-AI). We implement all three analyzers in Lean 4, building on the LeanJavaScripty framework for JavaScript analysis. The evaluation aims to answer the following research questions:

- **RQ1 (Precision):** Does removing context pollution allow the analyzer to reject infeasible paths and false positives?
- **RQ2 (Semantics):** Does a focused context enable the LLM to identify more complex invariants (e.g., exponential growth)?
- **RQ3 (Efficiency):** Does slicing reduce token consumption and analysis time compared to full-context baselines?

7.2 Notes on Evaluation

Some notes on evaluation that have ultimately affected the results:

- **LLM Choice:** We use Ollama's Llama 3 model for all LM-guided analyses. While not specialized for code, Llama 3 demonstrates strong general reasoning capabilities [2]. Future work could explore code-specific models like Gemini, GPT-5. I do believe that larger models would perform better with larger contexts, with the downsides of cost. This is something that should be explored in future work.
- **Benchmark Suite:** We utilize a subset of the SV-COMP [1] benchmark suite, focusing on JavaScript programs with complex control flow and data structures. Benchmarks were selected to highlight scenarios where traditional widening fails due to imprecise invariants.
- **Variable types:** Currently, the implementation only supports integer variables and interval abstractions. Extending support to heap abstractions and object properties is left for future work. Object abstraction is where AbsInt-AI found the most success in reducing false positives, so this is a significant limitation of the current evaluation.
- **No Functions or Global State:** The current implementation only supports straight-line code without functions or global state. This simplification was necessary to focus on the core slicing mechanism. Future work should extend the framework to handle function calls, recursion, and global variables. This is another area where I believe the slicing would provide significant benefits, as functions often introduce a lot of irrelevant context, but it's semantics are crucial for the LM.
- **Graph Representation vs Syntactic Representation:** Currently, I have only evaluated the syntactic representation of the slice. Future work should compare the two representations to see which yields better results with the LLM. My hypothesis is that the graph representation would perform better, as it is more concise and directly captures dependencies without syntactic noise.

7.3 Test 1: Modular Independence

To test the functionality of the slicer, and act on my hypothesis that context pollution degrades LM performance, I create a test program containign two independent modules:

- **Module A: Noise**: An exponential backoff loop (timeout) with complex arithmetic.
- **Module B: Target**: A simple linear counter (retries).

Our slicer should identify that Module A is irrelevant when targeting the `retries` variable. The full program and sliced program are shown in Figure **??**.

A. Original Program (Full Context)

```
// Initialization
timeout := 1;
retries := 0;

// MODULE A: Noise
// (Exponential Complexity)
while timeout < 2000 do
    timeout := timeout + timeout;

// MODULE B: Target
// (Linear Complexity)
while retries < 5 do
    retries := retries + 1;
```

B. Sliced Program (Target: `retries`)

```
// timeout init removed
retries := 0;

// MODULE A: Pruned
// ( identified as dead code )
skip;


// MODULE B: Target
// (Linear Complexity)
while retries < 5 do
    retries := retries + 1;
```

7.3.1 *Results.*
The results of the modular independence test are summarized in Table 1.

Table 1. Comparison of Analysis Results for Modular Independence (Test W11)

Analyzer	**Noise Loop** (`timeout`)	**Target Loop** (`retries`)	**Outcome**
Pure Abstract Interpreter	$[1, \infty]$	$[0, \infty]$	Sound (Baseline)
Full-Context LLM	$[-\infty, +\infty]$	$[-\infty, +\infty]$	**Polluted (Panic)**
SAINT (Sliced)	$[1, +\infty]$	$[0, \infty]$	**Clean (Sound)**

7.3.2 *Note to Professor.*
Looking at these results, I believe that slicing is working is intended. What I am struggling with is building a more precise Abstract Interpreter. It shouldn't be the case that we should widen at 5, maybe narrowing needs to be implemented. I've played around with my widening thresholds, and anything above 5 loops causes it to hang. If I had more time I would focus on improving the underlying abstract interpreter to be more precise, as I believe that would allow the LM to make better predictions.

8 CONCLUSION

8.1 Further Work

This work presents an interesting opportunity for further exploration in Abstract Interpretation with External libraries. External Libraries and API's, make Abstract Interpretation particularly challenging, as the semantics of these functions are often unknown or too complex to model precisely. Future work could explore how LLM calls can be integrated to summarize the effects of external library calls, potentially using slicing to isolate the relevant parts of the program that interact with these libraries.

In terms of further work on this paper, there is a lot to be done. First, the abstract interpreter needs to be more robust, supporting heap abstractions, functions, and global state. Second, the evaluation needs to be more comprehensive, exploring a wider range of benchmarks and LLMs. Finally I need to be build the caching mechanisms, in order to reduce the analysis runtime.

8.2 What I Learned

I found that really building the static analyzer set everything in perspective for me. Watching all the theories and the rules, turn into computation was really rewarding. I also learned a lot about prompt engineering for LLMs, and how to really think about what context is necessary for the model to make good predictions, and how to get as precise of an answer as possible. I am definitely excited to play with Lean more in the future. It is such an interesting language.

8.3 Accomplishments

In this paper I have experimented on the idea of context pollution in neurosymbolic static analysis, and proposed a novel slicing mechanism to mitigate it. I have built a prototype static analyzer in Lean 4, implementing the core slicing algorithm and integrating it with an LLM oracle. I have also conducted preliminary evaluations demonstrating the effectiveness of slicing in improving analysis precision.

ACKNOWLEDGMENTS

TBD

REFERENCES

[1] Dirk Beyer. 2011. Software Verification Competition (SV-COMP). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 505–518.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

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

[4] Alex Wang, Chen Liu, and Yi Zhang. 2025. AbsInt-AI: Sound Static Analysis via Neurosymbolic Guidance. In *Proceedings of the 52nd ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, New York, NY, USA.

[5] Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (1984), 352–357.