Hallucination-Resilient LLM-Driven Sound and Tunable Static Analysis

A Case of Higher-Order Control-Flow Analysis

Guannan Wei
Inria & ENS | PSL
Paris, France
Tufts University
Medford, MA, USA
guannan.wei@tufts.edu

Zhuo Zhang
Columbia University
New York, NY, USA
zz3474@columbia.edu

Caterina Urban
Inria & ENS | PSL
Paris, France
caterina.urban@inria.fr

Abstract

We argue that soundness remains essential for LLM-driven static analysis and discuss hallucination-resilient approaches in combining LLMs with static analysis that ensure soundness while improving precision. We propose to use LLMs as a way of meta-analysis and investigate this approach in higher-order control-flow analysis, building on the abstracting abstract machine framework and delegating abstract address allocation to an LLM. Our analyzer LLMAAM maintains soundness regardless of LLM behavior, while adaptively tuning analysis precision. We report promising preliminary results and outline broader opportunities for sound LLM-driven analysis.

CCS Concepts: **• Theory of computation → Program analysis**; **Abstraction**; **• Computing methodologies → Machine learning approaches**.

Keywords: static analysis, large language models, control-flow analysis

ACM Reference Format:
Guannan Wei, Zhuo Zhang, and Caterina Urban. 2025. Hallucination-Resilient LLM-Driven Sound and Tunable Static Analysis: A Case of Higher-Order Control-Flow Analysis. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Language Models and Programming Languages (LMPL '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3759425.3763378

1 Introduction

Witnessing the recent success (e.g., [1, 25]) of large language models (LLMs) in various code generation and comprehension tasks, it is promising to use LLMs to perform static analysis of programs, i.e., to predict the dynamic behavior of programs without executing them. A popular approach involves carefully crafting *prompts* alongside the program text to query LLMs about semantic properties of a program, such as data flow [36], call graph [31], bug detection [19], etc. Despite the promising yet controversial hypothesis that LLMs can "reason" the semantics of programs, they do not directly lead to semantically sound analyses. LLMs are prone to hallucinate [18], i.e., producing outputs that are factually incorrect or inconsistent with the actual program behaviors.

Even worse, unlike code generation tasks, where output programs can be validated using testing or other forms of specifications, there is no general way to automatically validate the results of static analysis produced by LLMs. This stems from a fundamental challenge: static analysis problems are inherently undecidable, and so too is the validation of an LLM's output in such cases. Without effective means to validate, the static analysis results conducted by LLMs induce both false positives and false negatives, thus are often untrustworthy.

In the static analysis literature, *soundness* has long been the gold standard for correctness: it states that the analysis will not miss any true behavior of the program, although often at the cost of introducing spurious results. Foundational frameworks like abstract interpretation [4, 5] have enabled the design of sound analyses with formal guarantees. Unfortunately, it is not clear how LLM-driven static analysis can benefit from these rigorous foundations, and it remains an open challenge how to effectively use LLMs for static analysis that yields sound, precise, and useful results.

In this paper, we argue that soundness *remains* essential for LLM-driven static analysis and advocate leveraging LLMs in ways that are *hallucination-resilient by design.* To guarantee soundness, LLMs must not be part of the trusted base – they cannot be relied upon to make correctness-critical decisions. Yet to deliver useful results, LLMs must still be entrusted with non-trivial tasks that influence the precision or scalability of the analysis. While reconciling these goals may seem challenging, we believe that there is an ample room to incorporate LLMs meaningfully without compromising soundness. For instance, LLMs could assist

ACM ISBN 979-8-4007-2148-9/25/10
https://doi.org/10.1145/3759425.3763378

with key heuristics in static analysis, such as variable packing [2, 30, 35], trace partitioning [24], synthesizing ranking functions [3, 23, 32], or guiding allocation-based polyvariance decisions [7] – all of which affect analysis precision but not soundness.

These use cases point to a promising direction: letting LLMs improve precision or scalability while leaving soundness in the hands of proven formal foundations. To make this approach effective, we propose that LLMs should reason not only about the target program, but about the underlying static analysis itself, drawing inspiration from the Cousot et al. [6]'s meta-abstract interpretation.

A Case of Control-Flow Analysis. To illustrate the feasibility and benefits of our approach, we investigate using LLMs for control-flow analysis (CFA) [21, 26] of higher-order and imperative programs. We develop LLMAAM, an LLM-driven analyzer that produces a state transition graph over-approximating control- and value-flow. Our analyzer builds on the abstracting abstract machine (AAM) [12, 13] framework, a recipe for deriving sound analyses from abstract machine semantics. The AAM approach permits various degrees of precision (e.g. context sensitivity) by using different abstract address allocation strategies [7], which essentially controls when the analysis should preserve (e.g., by keeping values concretely) or discard information (e.g., by joining abstract values).

More importantly, any allocation strategy is sound by the construction of AAM as long as the addresses are chosen from a finite set (even non-deterministically). This is known as the *a posteriori* soundness theorem [7, 22], which shows that one can always reconstruct a map from the concrete state to the abstract state given the allocated abstract addresses *after* the analysis, which justifies the soundness.

LLMAAM's key insight is to exploit the *a posteriori* soundness theorem [22] and use an LLM as the *abstract address allocator*, which on-the-fly synthesizes the abstract addresses used in the analysis and entirely decides the abstraction of the control flow and the precision of the analysis. Meanwhile, the *a posteriori* soundness theorem ensures that our analysis is resilient to LLM hallucination: if the LLM is properly prompted, it could return a good abstract address leading to the desired precision (i.e. less false positives).

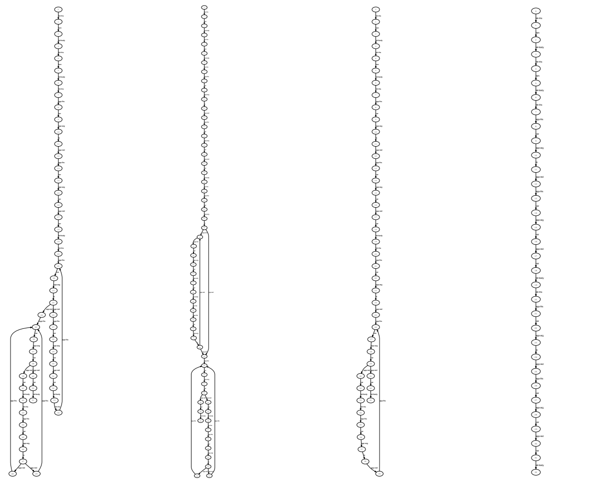
Unlike many prior work that prompts with the target program, LLMAAM uses LLMs to analyze the undergoing static analysis (instead of the target program), and asks the LLM to reason about the process of abstract interpretation (i.e. abstract states) by observing the analyzer implementation. Therefore, our approach is a meta-analysis using LLMs for abstract interpretation.

Contributions. (1) To soundly use LLMs in static analysis, we propose to use LLMs as a way of meta-analysis. Following this idea, we develop LLMAAM, a proof-of-concept LLM-driven abstract interpreter for higher-order programs, and present promising preliminary results on micro benchmarks. (2) We discuss several other opportunities in improving the precision of static analysis by leveraging LLMs in ways that are hallucination-resilient by design.

2 LLMAAM in Action

To see LLMAAM in action, we use a simple program shown in Figure 1, which illustrates potential spurious bindings and control-flow imprecision. The program defines identity functions `id` and `idid` (an η-expansion of `id`), applies `idid` twice, and returns one of the results.

```
let id = λz. z in
let idid = λw. id(w) in
let x = idid(1) in
let y = idid(2) in x
```



(a) 0CFA **(b)** 1CFA **(c)** 0CFA+P4F **(d)** 1CFA+P4F; or LLMAAM

Figure 1. A simple program and its visualized analysis results under different address allocation strategies: (d) is the optimal result (no spurious transitions), achievable either manually via 1CFA+P4F or automatically by LLMAAM. (a) and (b) use target expressions as continuation addresses; (c) and (d) use the P4F [8]. LLMAAM (d) uses an adaptive strategy.

In AAM, the degree of over-approximation (i.e. precision) is determined by the abstract address allocator. This allocator assigns abstract addresses to resources like bindings and continuations for storage on the abstract store. Reusing an address causes multiple resources to be merged, introducing over-approximation and reducing precision.

For example, in a context-insensitive analysis like 0CFA (Figure 1a), the variable `w` is always assigned the same abstract address $\alpha_{\mathtt{w}}$, regardless of the call site of `idid`. As a result, the argument values from both calls are merged in the store: $\alpha_{\mathtt{w}} \mapsto \{1, 2\}$, and this imprecision propagates to `z`. Adding context sensitivity (e.g., 1CFA shown in Figure 1b) helps distinguish bindings of `w`, yet imprecision remains due to conflated return flows (i.e., continuations).

Since `idid` is called twice, a simple strategy using the target expression as the continuation address (e.g. the baseline in [8]) would conflate their return flows, known as the "call/return mismatch" [34] issue in the literature. Figure 1a and Figure 1b are the resulting over-approximated state graphs under this continuation allocation strategy but with different context sensitivities.

To obtain precise flow information for this program, the analyzer must distinguish *both* binding values of `w`/`z` *and* the return flows of these function calls of `id`/`idid`. To achieve the later, the pushdown-for-free (P4F) [8] strategy can be used, which augments the continuation address with the target environment (i.e., an mapping from variables to abstract addresses). Figure 1d illustrates the result of combining 1CFA with P4F, yielding a precise state graph without superfluous transitions, unlike other allocation strategies.

However, tuning precision in this way is non-trivial and increased context-sensitivity does not always lead to better precision. Precision tuning requires deep insight into the program's structure, even for a simple one shown in Figure 1. Our analyzer, LLMAAM, achieves this optimal precision automatically by delegating abstract address allocation to an LLM, which prompts the LLM with abstract states.[1]

Notably, the LLM's allocation strategy is adaptive: it does not uniformly follow 1CFA or P4F, but selectively applies these techniques at critical points in the analysis. Analysis designers can further tune the precision (e.g. selectively applied to certain functions) by adjusting the prompts. The soundness of this flexible, LLM-guided strategy is ensured by the a posteriori soundness theorem of AAM [22].

3 LLMAAM: Design and Implementation

Building on top of the AAM framework, LLMAAM abstractly simulates the execution of the target program as a nondeterministic abstract machine. This section sketches the excerpted abstract semantics, then explains how LLMs drive abstract address allocation. Much of the abstract semantics is standard, so we refer the reader to more pedagogical treatments [13] for full details.

3.1 Language

We explain our approach using a higher-order imperative intermediate language HoImp (Figure 2) based on the untyped call-by-value λ-calculus. In addition to core functional constructs, HoImp includes practical language features such as conditionals, while-loops, and mutable state.

In essence, HoImp models a subset of popular dynamically typed languages, such as Python or Scheme. While additional type information can improve precision, they are orthogonal to our core analysis. HoImp serves as an *intermediate language*, thus we assume a front-end translates source programs (e.g., Scheme) into HoImp. When interacting with the LLM, we reify the program's AST expressed in our meta-language (Scala, in this case) to strings, rather than the source code.

[1] Conversion with Gemini and GPT4o available at https://github.com/Kraks/llmaam/blob/main/examples

$$
\begin{aligned}
&x \in \mathsf{Var} \quad n \in \mathbb{N} \quad b \in \{\#\mathtt{t}, \#\mathtt{f}\} \quad op_1 \in \{-, \dots\} \quad op_2 \in \{+, \dots\} \\
&e \in \mathsf{Expr} ::= n \mid b \mid \lambda x.e \mid op_1\ e \mid e_1\ op_2\ e_2 \mid \mathtt{begin}\ e^+ \\
&\qquad \mid \mathtt{set!}\ x\ e \mid (e_1\ e_2) \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \\
&\qquad \mid \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \mid \mathtt{while}\ e_1\ \mathtt{do}\ e_2 \mid \dots
\end{aligned}
$$

Figure 2. Abstract syntax of HoImp (excerpt).

$$
\begin{aligned}
\Sigma \in \widehat{\mathsf{State}} &\triangleq (\mathsf{Expr} + \widehat{\mathsf{Value}}) \times \widehat{\mathsf{Env}} \times \widehat{\mathsf{BStore}} \times \widehat{\mathsf{KStore}} \\
&\quad \times \widehat{\mathsf{Kont}} \times \widehat{\mathsf{Time}} \\
\rho \in \widehat{\mathsf{Env}} &\triangleq \mathsf{Var} \to \widehat{\mathsf{BAddr}} \\
\sigma_v \in \widehat{\mathsf{BStore}} &\triangleq \widehat{\mathsf{BAddr}} \to \mathcal{P}\left(\widehat{\mathsf{Value}}\right) \\
\sigma_\kappa \in \widehat{\mathsf{KStore}} &\triangleq \widehat{\mathsf{KAddr}} \to \mathcal{P}\left(\widehat{\mathsf{Kont}}\right) \\
t \in \widehat{\mathsf{Time}} &\triangleq \mathsf{Expr}^* \\
\alpha_v \in \widehat{\mathsf{BAddr}} &\triangleq \mathsf{Var} \times \widehat{\mathsf{Instm}}^* \\
\alpha_\kappa \in \widehat{\mathsf{KAddr}} &\triangleq \widehat{\mathsf{Instm}}^* \\
i \in \widehat{\mathsf{Instm}} &\triangleq \mathsf{Expr} + \widehat{\mathsf{Env}} + \widehat{\mathsf{BStore}} + \widehat{\mathsf{Time}} \\
v \in \widehat{\mathsf{Value}} &\triangleq \top_{\mathsf{num/bool}} \mid (\lambda x.e \times \widehat{\mathsf{Env}}) \\
k \in \widehat{\mathsf{Kont}} &::= \mathsf{KHalt} \mid \mathsf{KLet}(x, e, \rho, \alpha_\kappa) \\
&\quad \mid \mathsf{KArg}(e, \rho, \alpha_\kappa) \mid \mathsf{KFun}(\lambda x.e, \rho, \alpha_\kappa) \mid \dots
\end{aligned}
$$

Figure 3. Definition of the abstract state space.

$$
\begin{aligned}
tick &: \widehat{\mathsf{State}} \to \widehat{\mathsf{Time}} \\
alloc_v &: \widehat{\mathsf{State}} \times \mathsf{Var} \times \widehat{\mathsf{Time}} \to \widehat{\mathsf{BAddr}} \\
alloc_\kappa &: \widehat{\mathsf{State}} \times \mathsf{Expr} \times \widehat{\mathsf{Env}} \times \widehat{\mathsf{BStore}} \times \widehat{\mathsf{Time}} \to \widehat{\mathsf{KAddr}}
\end{aligned}
$$

Figure 4. LLM-tuned functions in LLMAAM.

3.2 Abstract Semantics

Abstract State. Figure 3 shows the definition of abstract states Σ. An abstract state consists of (1) the control string or abstract value, (2) the abstract environment for local bindings, (3) the abstract store for heap-allocated binding values, (4) the continuation store for heap-allocated continuations, (5) the current continuation, and (6) a bounded time-stamp (i.e. history of computation).

An environment is a mapping from variables to abstract binding addresses. We divide the abstract store into two parts: (1) *binding store* $\widehat{\mathsf{BStore}}$ and (2) *continuation store* $\widehat{\mathsf{KStore}}$. Binding stores map abstract addresses to set-lattices of abstract values, which are either the top of numbers/booleans or a closure. We aggressively abstract over primitive values

$$s@\langle\, x, \rho, \sigma_v, \sigma_\kappa, \kappa, t \,\rangle \longrightarrow \langle\, v, \rho, \sigma_v, \sigma_\kappa, \kappa, \mathit{tick}(s) \,\rangle$$
$$\text{where } v \in \sigma_v(\rho(x)) \qquad [\textsc{var}]$$
$$s@\langle\, n, \rho, \sigma_v, \sigma_\kappa, \kappa, t \,\rangle \longrightarrow \langle\, \top_{\mathsf{num}}, \rho, \sigma_v, \sigma_\kappa, \kappa, \mathit{tick}(s) \,\rangle$$
$$\text{where } n \in \mathbb{N} \qquad [\textsc{lit-num}]$$
$$s@\langle\, \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2, \rho, \sigma_v, \sigma_\kappa, \kappa, t \,\rangle \longrightarrow$$
$$\langle\, e_1, \rho, \sigma_v, \sigma'_\kappa, \mathsf{KLet}(x, \rho, e_2, \alpha_\kappa), t' \,\rangle$$
$$\text{where } t' = \mathit{tick}(s) \quad \alpha_\kappa = \mathit{alloc}_\kappa(s,\ e_1,\ \rho,\ \sigma_v,\ t')$$
$$\sigma'_\kappa = \sigma_\kappa \sqcup \{\ \alpha_\kappa \mapsto \kappa\ \} \qquad [\textsc{let-rhs}]$$
$$s@\langle\, v, \rho, \sigma_v, \sigma_\kappa, \mathsf{KLet}(x, \rho, e, \alpha_\kappa), t \,\rangle \longrightarrow \langle\, e, \rho', \sigma'_v, \sigma_\kappa, \kappa, t \,\rangle$$
$$\text{where } \alpha_v = \mathit{alloc}_v(s,\ x,\ t) \qquad \rho' = \rho[x \mapsto \alpha_v]$$
$$\sigma'_v = \sigma_v \sqcup \{\ \alpha_v \mapsto v\ \} \quad \kappa \in \sigma_\kappa(\alpha_\kappa) \qquad [\textsc{let-body}]$$
$$s@\langle\, (e_1\, e_2), \rho, \sigma_v, \sigma_\kappa, \kappa, t \,\rangle \longrightarrow$$
$$\langle\, e_1, \rho, \sigma_v, \sigma'_\kappa, \mathsf{KArg}(e_2, \rho, \alpha_\kappa), t' \,\rangle$$
$$\text{where } t' = \mathit{tick}(s) \quad \alpha_\kappa = \mathit{alloc}_\kappa(s,\ e_1,\ \rho,\ \sigma_v,\ t')$$
$$\sigma'_\kappa = \sigma_\kappa \sqcup \{\ \alpha_\kappa \mapsto \kappa\ \} \qquad [\textsc{app-lhs}]$$
$$s@\langle\, (\lambda x.\, e_1,\ \rho_1), \_, \sigma_v, \sigma_\kappa, \mathsf{KArg}(e_2, \rho_2, \alpha'_\kappa), t \,\rangle \longrightarrow$$
$$\langle\, e_2, \rho_2, \sigma_v, \sigma_\kappa, \mathsf{KFun}(\lambda x.e_1, \rho_1, \alpha_\kappa), t \,\rangle \qquad [\textsc{app-arg}]$$
$$s@\langle\, v, \_, \sigma_v, \sigma_\kappa, \mathsf{KFun}(\lambda x.e, \rho, \alpha_\kappa), t \,\rangle \longrightarrow \langle\, e, \rho', \sigma'_v, \sigma_\kappa, \kappa, t \,\rangle$$
$$\text{where } \alpha_v = \mathit{alloc}_v(s,\ x,\ t) \qquad \rho' = \rho[x \mapsto \alpha_v]$$
$$\sigma'_v = \sigma_v \sqcup \{\ \alpha_v \mapsto v\ \} \quad \kappa \in \sigma_\kappa(\alpha_\kappa) \qquad [\textsc{app-beta}]$$

Figure 5. Selected abstract transition rules. Note that rules VAR, LET-BODY, and APP-BETA are nondeterministic.

(e.g., numbers); other choices such as interval with suitable widening strategies are possible. Continuations are defined in the standard way. Both stores are point-wise lattices, thus $\sigma_1 \sqcup \sigma_2 = \lambda\alpha.\sigma_1(\alpha) \cup \sigma_2(\alpha)$.

A binding address $\widehat{\mathsf{BAddr}}$ must consist of a variable, and it may also contain additional instrumentations $\widehat{\mathsf{Instm}}$. A continuation address is more flexible in that it is just instrumented by a sequence of $\widehat{\mathsf{Instm}}$. Instrumentation is the key device to control the precision of the analysis, which determines how the abstract state-space partitions the concrete state-space. In addition to expressions, environments, and stores, time-stamps can be used to instrument addresses with the (partial) computation history.

Transition. Figure 5 shows selected rules of the abstract transition relation $\Sigma \to \mathcal{P}(\Sigma)$. In these rules, we write $\rho[x \mapsto \alpha]$ to denote a new environment extending ρ with a binding of variable x to address α. We write $\sigma \sqcup \{\ \alpha \mapsto v\ \}$ to denote join-update, i.e., the new store has $\alpha \mapsto \sigma(\alpha) \cup \{\ v\ \}$. Additionally, $\sigma(\alpha) = \varnothing$ if α is undefined in σ.

If the first component of state is a complex expression (e.g., APP-LHS), we step to the successor state with the subexpression according the evaluation order, a new continuation and continuation store, and the $\widehat{\mathit{tick}}$-ed time-stamp (which can instrument further addresses). If the state's first component is already a syntactic value (e.g. LIT-NUM), we steps to a successor state with its corresponding abstract value (e.g. $\top_{\mathsf{num}}$). If the first component is already an abstract value (e.g. APP-ARG and APP-BETA), we inspect the continuation to decide the successor states, which can be nondeterministic due to dereference of the continuation address. The analysis is performed by collecting all reachable states until reaching a fixpoint. Since all components of Σ are finite, the state space is finite and the analysis terminates.

Revisiting the *a Posteriori* Soundness Theorem. During the transition, the analysis uses several precision parameters (Figure 4), which produce addresses $\widehat{\mathsf{BAddr}}$ and $\widehat{\mathsf{KAddr}}$. Some instantiations of these parameters include various forms of context/object-sensitivity [7, 27] and pushdown-for-free (P4F) [8]. In LLMAAM, these parameters are tuned dynamically by an LLM rather than predetermined, with prompting details discussed in Section 3.3.

AAM permits *a posteriori* soundness: the analysis is sound regardless of the chosen abstract addresses, as long as they are drawn from a finite set. Note that if the allocator always returns fresh addresses, the "analysis" recovers *concrete* execution (thus may not be computable). Informally, the theorem guarantees that after analysis, one can always reconstruct an abstraction map that justifies the chosen addresses, serving as a witness of soundness. In essence, the allocator defines a partition of the concrete address space. For details, see Might and Manolios [22] and Gilray et al. [7].

3.3 Prompts

Having established that LLMs can be used soundly in our analysis, the next question is: how should we prompt them? Instead of directly prompting with the target program, LLMAAM prompts the LLM to reason about the ongoing analysis itself in an online manner.

The system prompt defines the analysis task, including data structures from Figure 3, the logic of address allocation, and instructions for reasoning. We also define a query protocol (Figure 4), where each query is a JSON object containing the current abstract state $\widehat{\Sigma}$, a query type (`BAddr`, `KAddr`, or `Tick`), and additional context.

The LLM responds with a JSON object containing its reasoning and result. For `BAddr` and `KAddr`, it returns booleans indicating which fields to use for instrumentation; for `Tick`, it returns a number `k` specifying how many call-contexts to retain. LLMAAM parses the response and constructs abstract and continuation addresses accordingly. The full prompt can be found in our implementation [9].

This, however, is just one prompting strategy. We discuss a few other enhanced strategies: (1) Including transition rules in the prompt could help the LLM reason more systematically about the analysis process. (2) Additional user-provided prompts could tune the precision, such as guiding the LLMs

Input to the LLM:

```
{
  "state": <State Σ̂ reified as string>,
  "query-type": "BAddr" or "KAddr" or "Tick",
  "variable": String,             // BAddr only
  "time": Time,
  "source-expression": Expr,      // KAddr only
  "target-expression": Expr,      // KAddr only
  "target-environment": Env,      // KAddr only
  "target-binding-store": BStore, // KAddr only
}
```

Expected output from LLM:

```
{
  "reason": string
  "query-type": "BAddr" or "KAddr" or "Tick",,
  "variable": String,             // BAddr only
  "time": Bool,
  "source-expression": Bool,      // KAddr only
  "target-expression": Bool,      // KAddr only
  "target-environment": Bool,     // KAddr only
  "target-binding-store": Bool,   // KAddr only
  "k": Int represented as String, // Tick only
}
```

Figure 6. Input/output protocol for LLM interaction.

for better precision on specific functions or program points. (3) Furthermore, since abstract states are discrete, we can reify not only the current state but also bounded past and future states for retrospective or prospective meta-analysis. A systematic evaluation of these variant prompting in our framework is an interesting direction for future work.

3.4 Implementation and Preliminary Evaluation

We implement the prototype analyzer LLMAAM [9] in Scala using `LangChain4j`. We also have a front-end that can parse Scheme into our IR.

Table 1. Preliminary evaluation; best results highlighted. AAC follows the formulation given in Sec. 3.4 of [8].

	idid		kcfa2		mj09		loop2	
	#N	#E	#N	#E	#N	#E	#N	#E
0CFA+P4F	42	41	307	306	303	302	110	108
0CFA+AAC	33	32	203	202	**179**	**175**	110	108
0CFA+SRC	42	41	246	245	647	644	149	146
1CFA+P4F	33	32	TO	TO	1437	1407	108	106
1CFA+AAC	33	32	336	324	331	323	108	106
1CFA+SRC	42	41	12189	12117	6849	6834	108	106
	33	**32**	**133**	**131**	199	202	**108**	**106**
llmaam	33	32	153	150	268	259	108	106
	33	32	272	265	281	273	108	106

To assess the feasibility and benefits of our approach, we conduct a preliminary evaluation (Table 1) on several micro-benchmarks and compare with other typical deterministic allocation strategies. All the following results do not use store widening. Full-scale evaluation on realistic benchmarks is left for future work. Due to the inherent randomness of LLMs, we report 3 results of LLMAAM using GPT-4o-mini. TO indicates a 10-minute timeout.

Since all results are over-approximations, we use the number of states (#N) and edges (#E) as a proxy for precision. Few numbers indicate less spurious result. We observe that on 3 out of 4 benchmarks, LLMAAM outperforms other strategies in terms of both states and edges. The other benchmark is close to the best deterministic strategy.

4 Conclusion and Future Opportunities

We have demonstrated LLMAAM, an abstract interpreter for higher-order control-flow analysis utilizing LLMs as the abstract address allocator. We plan to further extend LLMAAM to handle realistic languages and large-scale analysis.

The core insight behind our approach, i.e., using LLMs as a meta-analysis to improve static analysis in hallucination-resilient ways, extends beyond CFA and LLMAAM. Below, we discuss several related works and promising directions where LLMs could play a meaningful role in improving precision or scalability, while maintaining soundness through established frameworks.

Control-Flow Sensitivity. Similar to allocation-based polyvariance [7, 22] used in LLMAAM, control-flow widening [11] and trace partitioning [10, 20, 24] formulate control-flow sensitivity in different ways. As where to apply partition or widening the control is often a heuristic decision, they can be synthesized by an LLM too. Similar to LLMAAM, Wang et al. [37] explored using LLMs to decide heap-objects abstraction for loops, which we believe can be subsumed by a more systematic approach such as Gilray et al. [7].

Termination Analysis. Termination analysis often relying on ranking functions and widening heuristics. Designing piecewise-defined ranking functions is a notoriously difficult task, and existing techniques rely on human-crafted templates or search heuristics. As evidenced by PROTON [23], LLMs could aid in synthesizing such ranking functions or guiding widening strategies [32, 33] given that the underlying analysis is resilient to unsound widening heuristics.

Numeric Analysis. Variable packing [2, 35] or variable partitioning [28, 30] have been used to improve the precision and scalability of numeric abstract domains. Both of them require heuristic decisions about which variables to group or separate, which we believe can be suggested by an LLM either offline (i.e. before the analysis) or online. Prior to the LLM-era, reinforcement learning has been used in numerical abstract domains too [29].

Pointer Analysis. Pointer analyses have embraced data-driven techniques to improve precision and scalability [14–17]. These works already demonstrate the benefit of learning-based heuristics, and it would be interesting to see how LLMs can further improve pointer analyses.

Acknowledgments

The authors would like to thank Chengpeng Wang for insightful discussions. This work was partially supported by the SAIF project, funded by the "France 2030" government investment plan managed by the French National Research Agency, under the reference ANR-23-PEIA-0006.

References

[1] 2024. Amazon CodeWhisperer. https://aws.amazon.com/codewhisperer/.

[2] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *PLDI*. ACM, 196–207.

[3] Nathanaël Courant and Caterina Urban. 2017. Precise Widening Operators for Proving Termination by Abstract Interpretation. In *TACAS (Lecture Notes in Computer Science, Vol. 10205)*. Springer, 136–152.

[4] Patrick Cousot. 2021. *Principles of abstract interpretation.* MIT Press.

[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 *POPL*. ACM, 238–252.

[6] Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. 2019. A^2I: abstract^2 interpretation. *Proc. ACM Program. Lang.* 3, POPL (2019), 42:1–42:31.

[7] Thomas Gilray, Michael D. Adams, and Matthew Might. 2016. Allocation characterizes polyvariance: a unified methodology for polyvariant control-flow analysis. In *ICFP*. ACM, 407–420.

[8] Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016. Pushdown control-flow analysis for free. In *POPL*. ACM, 691–704.

[9] Guannan Wei, Zhuo Zhang, and Caterina Urban. 2025. LLMAAM Analyzer. https://github.com/Kraks/llmaam.

[10] Maria Handjieva and Stanislav Tzolovski. 1998. Refining Static Analyses by Trace-Based Partitioning Using Control Flow. In *SAS (Lecture Notes in Computer Science, Vol. 1503)*. Springer, 200–214.

[11] Ben Hardekopf, Ben Wiedermann, Berkeley R. Churchill, and Vineeth Kashyap. 2014. Widening for Control-Flow. In *VMCAI (Lecture Notes in Computer Science, Vol. 8318)*. Springer, 472–491.

[12] David Van Horn and Matthew Might. 2010. Abstracting abstract machines. In *ICFP*. ACM, 51–62.

[13] David Van Horn and Matthew Might. 2012. Systematic abstraction of abstract machines. *J. Funct. Program.* 22, 4-5 (2012), 705–746.

[14] Minseok Jeon, Sehun Jeong, Sung Deok Cha, and Hakjoo Oh. 2019. A Machine-Learning Algorithm with Disjunctive Model for Data-Driven Program Analysis. *ACM Trans. Program. Lang. Syst.* 41, 2 (2019), 13:1–13:41.

[15] Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and scalable points-to analysis via data-driven context tunneling. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 140:1–140:29.

[16] Minseok Jeon, Myungho Lee, and Hakjoo Oh. 2020. Learning graph-based heuristics for pointer analysis without handcrafting application-specific features. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 179:1–179:30.

[17] Sehun Jeong, Minseok Jeon, Sung Deok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 100:1–100:28.

[18] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Yejin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.* 55, 12 (2023), 248:1–248:38. https://doi.org/10.1145/3571730

[19] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 474–499.

[20] Laurent Mauborgne and Xavier Rival. 2005. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *ESOP (Lecture Notes in Computer Science, Vol. 3444)*. Springer, 5–20.

[21] Jan Midtgaard. 2012. Control-flow analysis of functional programs. *ACM Comput. Surv.* 44, 3 (2012), 10:1–10:33.

[22] Matthew Might and Panagiotis Manolios. 2009. A Posteriori Soundness for Non-deterministic Abstract Interpretations. In *VMCAI (Lecture Notes in Computer Science, Vol. 5403)*. Springer, 260–274.

[23] Diganta Mukhopadhyay, Ravindra Metta, Hrishikesh Karmarkar, and Kumar Madhukar. 2025. PROTON 2.1: Synthesizing Ranking Functions via fine-tuned locally Hosted LLM (Competition Contribution). In *TACAS (3) (Lecture Notes in Computer Science, Vol. 15698)*. Springer, 242–247.

[24] Xavier Rival and Laurent Mauborgne. 2007. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.* 29, 5 (2007), 26.

[25] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023). https://doi.org/10.48550/ARXIV.2308.12950 arXiv:2308.12950

[26] Olin Shivers. 1988. Control-Flow Analysis in Scheme. In *PLDI*. ACM, 164–174.

[27] Olin Shivers. 1991. The Semantics of Scheme Control-Flow Analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'91, Yale University, New Haven, Connecticut, USA, June 17-19, 1991*, Charles Consel and Olivier Danvy (Eds.). ACM, 190–198. https://doi.org/10.1145/115865.115884

[28] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2017. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 46–59. https://doi.org/10.1145/3009837.3009885

[29] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2018. Fast Numerical Program Analysis with Reinforcement Learning. In *CAV (1) (Lecture Notes in Computer Science, Vol. 10981)*. Springer, 211–229.

[30] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2018. A practical construction for decomposing numerical abstract domains. *Proc. ACM Program. Lang.* 2, POPL (2018), 55:1–55:28.

[31] Chia-Yi Su and Collin McMillan. 2025. Do Code LLMs Do Static Analysis? arXiv:2505.12118 [cs.SE] https://arxiv.org/abs/2505.12118

[32] Caterina Urban. 2013. The Abstract Domain of Segmented Ranking Functions. In *SAS (Lecture Notes in Computer Science, Vol. 7935)*. Springer, 43–62.

[33] Caterina Urban and Antoine Miné. 2014. A Decision Tree Abstract Domain for Proving Conditional Termination. In *SAS (Lecture Notes in Computer Science, Vol. 8723)*. Springer, 302–318.

[34] Dimitrios Vardoulakis and Olin Shivers. 2010. CFA2: A Context-Free Approach to Control-Flow Analysis. In *ESOP (Lecture Notes in Computer Science, Vol. 6012)*. Springer, 570–589.

[35] Arnaud Venet and Guillaume P. Brat. 2004. Precise and efficient static array bound checking for large embedded C programs. In *PLDI*. ACM, 231–242.

[36] Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, Xiaoheng Xie, and Xiangyu Zhang. 2024. LLMDFA: Analyzing Dataflow in Code with Large Language Models. In *NeurIPS*.

[37] Michael Wang, Kexin Pei, and Armando Solar-Lezama. [n. d.]. ABSINT-AI: Language Models for Abstract Interpretation. In *ICLR 2025 Workshop: VerifAI: AI Verification in the Wild*.

Received 2025-07-06; accepted 2025-08-08