ABSINT-AI: AGENTIC HEAP ABSTRACTIONS FOR ABSTRACT INTERPRETATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Static program analysis is a foundational technique in software engineering for reasoning about program behavior. Traditional static analysis algorithms model programs as logical systems with well-defined semantics, but rely on uniform, hard-coded heap abstractions. This limits their precision and flexibility, especially in dynamic languages like JavaScript, where heap structures are heterogeneous and difficult to analyze statically. In this work, we introduce ABSINT-AI, a language-model-guided static analysis framework that augments abstract interpretation with adaptive, per-object heap abstractions for Javascript. This enables the analysis to leverage high-level cues, such as naming conventions and access patterns, without requiring brittle, hand-engineered heuristics. Importantly, the LM agent operates within a bounded interface and never directly manipulates program state, preserving the soundness guarantees of abstract interpretation. To evaluate our approach, we focus on a soundness-critical task: determining whether object property accesses may result in undefined or null dereferences. This task directly models a common requirement in compiler optimizations, where proving that an access is safe enables the removal of dynamic checks or simplifies code motion. On this task, ABSINT-AI reduces false positives by up to 34% compared to traditional static analyses with fixed heap abstractions, while preserving formal guarantees. Our ablations show that the LM's ability to interact agentically with the analysis environment is crucial, outperforming non-agentic LM predictions by 25%.

1 INTRODUCTION

As dynamic languages like JavaScript find their way into more backend applications with strong performance requirements, there has been a growing interest in compiling them down to more optimal forms (ang; Serrano, 2022; Chandra et al., 2016). An important obstacle for these approaches is the difficulty of performing sound static program analysis on these languages due to their dynamic behavior and extensive use of complex heap allocated data (Feldthaus et al., 2013; Antal et al., 2023; Sridharan et al., 2012). This is a problem because sound analysis is an essential element of compiler optimization (Hind, 2001; Schneck, 1973). Soundness ensures that the analysis captures all possible runtime behaviors of the program; without it, compilers cannot guarantee the safety of specific transformations.

A key challenge in sound and scalable static analysis for JavaScript is reasoning about the heap. JavaScript's dynamic object model allows programs to construct and mutate objects with unpredictable shapes, runtime-dependent fields, and implicit behavior tied to values stored within fields. Consider a typical loop that allocates multiple heterogeneous objects: some are short-lived wrappers, others are stable configuration records, and others may exhibit role-dependent behaviors encoded in field values. Traditional static analyses typically rely on uniform abstraction strategies, and often result in excessive over-approximation and imprecision. Constructing precise yet scalable heap abstractions is a major challenge for JavaScript due to its lack of static types and its permissive object model, and it remains a major bottleneck for static analysis frameworks.

In this paper, we introduce ABSINT-AI, an agentic framework that assists static analysis by performing heap abstractions. Our approach preserves the strong guarantees provided by traditional static analysis techniques while addressing some of their major limitations. Static analysis techniques analyze programs by treating them as sets of logical statements with well-defined semantics (Cousot & Cousot, 1977). This type of analysis can provide guarantees of soundness, but these methods leave

out a lot of information, such as variable names, comments, general programming design patterns, and background knowledge. LMs on the other hand, are able to take advantage of this information very well, but lack the robustness of traditional static analysis. For example, changing variable names has been shown to have a drastic impact on model performance (Zeng et al., 2022; Srikant et al., 2021). ABSINT-AI combines the best of both worlds by using LMs to provide background information to a static analyzer without losing soundness guarantees.

The key design choice in ABSINT-AI is that it preserves the formal soundness guarantee of symbolic program analysis by constraining the LM to only choose from a pre-determined set of *sound abstraction strategies* and decide *where* to apply abstractions. As a result, ABSINT-AI bounds the (inevitable) LM errors to only increased false positives (due to the aggressive abstraction decision) or slow down the convergence of the analysis (reduce to the precise but expensive analysis) without compromising the soundness.

Specifically, ABSINT-AI consists of a custom static analysis pipeline that invokes an agentic LM framework at key decision points - most notably before fixpoint computations in unbounded loops, where the choice of abstraction heavily influences convergence and precision. At each such point, the agent inspects the current analysis state, including the heap, code, and abstraction history. Based on this inspection, it selects appropriate abstraction strategies for each allocation site, such as merging objects using recency-abstraction, field sets, or value similarity. If the available information is insufficient to make a confident decision, the agent can request additional targeted analysis by executing the loop body for more iterations to refine its understanding. This interactive, goal-directed behavior enables adaptive, context-sensitive abstraction decisions and also allows the abstractions themselves to reflect higher-level semantic concepts. For example, if objects contain a `role` field, the agent can select a value-sensitive abstraction that merges all "teachers" into one object and all "students" into another, allowing domain-specific concepts to guide the abstractions themselves.

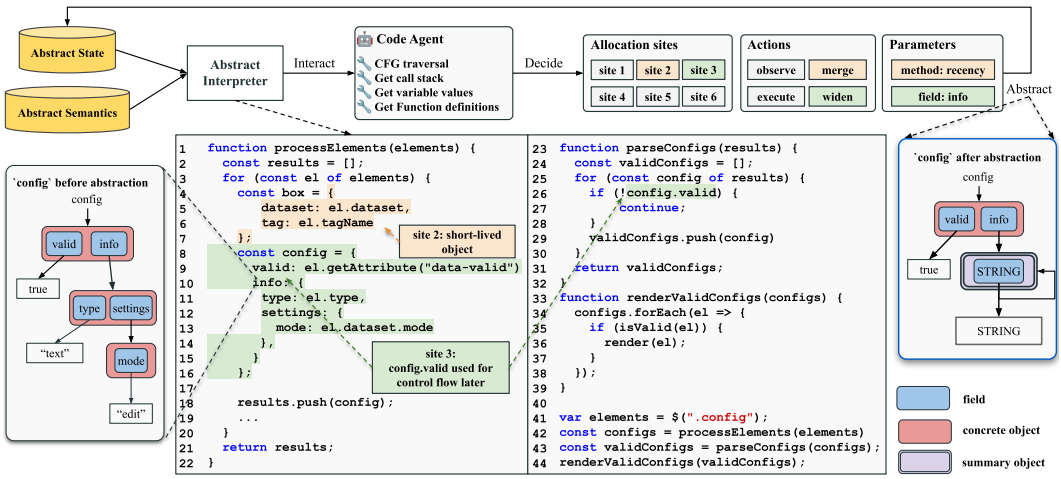
We evaluate our approach on the downstream task of detecting accesses to non-existent object fields, a common source of runtime errors in JavaScript. We compare our system against WALA (Santos & Dolby, 2022) and TAJS (Jensen et al., 2009), two state-of-the-art static analysis frameworks that are representative of conventional heap abstraction strategies. Our evaluation of real-world JavaScript programs shows that ABSINT-AI achieves up to a 34% reduction in false positives while maintaining soundness. Our ablations show that this improvement stems not just from more expressive abstractions, but from the agent's ability to interact with the analysis and adapt its choices to the program context. When run with fixed symbolic abstractions or using the LM in a single-shot, non-interactive mode, the false positive rate increases by 88% and 25%, respectively. These results highlight the benefit of adaptive, semantically informed heap abstractions in improving the practical effectiveness of sound JavaScript analysis.

2 MOTIVATING EXAMPLE

Static analyses rely on heap abstractions (summaries of sets of objects), to reason about dynamic, heap-manipulating programs. The precision of these abstractions has a huge impact: too coarse and the analysis produces spurious warnings; too fine and it may never converge.

Modern JavaScript programs often construct diverse heap objects with different structural patterns and semantic roles, even within the same control-flow context. A one-size-fits-all heap abstraction applied uniformly across the entire program can lead to loss of precision or unnecessary state explosion. Consider the example in Figure 1, where each iteration of processElements allocates two distinct objects: a short-lived wrapper (`box`), and a structured configuration object (`config`). Each of these demands a different abstraction strategy. For instance, box can be aggressively summarized without affecting soundness, while `config` exhibits a fixed field structure where only a single field, `valid`, must remain precise for correct downstream control flow. While it is theoretically possible to hand-engineer heuristics that assign abstraction strategies based on object structure or access patterns, doing so at scale quickly becomes brittle, complex, and difficult to maintain. To the best of our knowledge, existing analyses do not adapt their heap abstractions per object, due to the complexity and brittleness of manually encoding such decisions.

However, many real-world objects contain semantic hints in field names or surrounding code that indicate how they should be abstracted. For example, the field `valid` suggests that the `config`




```
1  function processElements(elements) {
2    const results = [];
3    for (const el of elements) {
4      const box = {
5          dataset: el.dataset,
6          tag: el.tagName
7      };
8      const config = {
9        valid: el.getAttribute("data-valid")
10       info: {
11         type: el.type,
12         settings: {
13           mode: el.dataset.mode
14       },
15       }
16     };
17
18     results.push(config);
19     ...
20   }
21   return results;
22 }
```

```
23 function parseConfigs(results) {
24   const validConfigs = [];
25   for (const config of results) {
26     if (!config.valid) {
27       continue;
28     }
29     validConfigs.push(config)
30   },
31   return validConfigs;
32 }
33 function renderValidConfigs(configs) {
34   configs.forEach(el => {
35     if (isValid(el)) {
36       render(el);
37     }
38   });
39 }
40
41 var elements = $(".config");
42 const configs = processElements(elements)
43 const validConfigs = parseConfigs(configs);
44 renderValidConfigs(validConfigs);
```

Figure 1: When ABSINT-AI encounters an unbounded loop, it suspends analysis and interacts with the language model agent for abstraction decisions. The agent selects a recency abstraction for `box` and a field-based widening for the `info` field of `config`, preserving relevant structure while ensuring convergence. A concrete instance of `config` is shown on the left, with its abstracted form on the right. These per-allocation-site abstraction decisions guide the analysis to a sound fixpoint.

object encodes access control logic, which is later reflected in a guard on `config.valid`. These high-level concepts such as "valid" configurations are difficult to capture using purely syntactic heuristics or static types, but are easily interpretable by language models. An agentic abstraction strategy can leverage such semantic cues to select more appropriate abstractions: preserving distinctions between roles, merging only safe-to-abstract fields, or even proposing domain-informed widenings. This enables adaptive precision where it matters, and aggressive summarization where it doesn't—leading to more efficient and accurate analyses.

In ABSINT-AI, a language model acts as an agent that guides heap abstraction dynamically over the course of the analysis. Returning to the example in Figure 1, the agent might decide to apply recency abstraction to the short-lived `box` object and a field-set abstraction to the structured `config` object (preserving only `config.valid`). These decisions are not hardcoded: the agent queries the analysis environment for relevant context (such as variable values and function definitions), and may request additional loop iterations to test its abstraction choices. Crucially, all semantics and state transitions are handled by a traditional abstract interpreter, ensuring that soundness is preserved. The agent's role is purely to steer how the heap is abstracted, enabling more precise and efficient analysis by tailoring abstraction to the semantics of the program.

3 METHODOLOGY

ABSINT-AI is based on traditional abstract interpretation, but queries an LM to decide how to merge summary nodes at key points in the analysis. The workflow of ABSINT-AI can be found in Figure 1.

3.1 BACKGROUND

Static program analysis. Static program analysis aims to reason about all possible executions of a program. A key property is *soundness*, meaning the analysis never misses a real bug (no false negatives). The tradeoff is *precision*: overly coarse reasoning introduces spurious warnings (false positives).

To ensure scalability, analyses use abstractions that merge unbounded program behaviors (e.g., integers, heap objects) into finite summaries. For heap-manipulating languages like JavaScript, this typically means summarizing many concrete objects into a smaller set of abstract objects. The challenge is choosing what to merge: aggressive abstraction hurts precision, while conservative abstraction may prevent convergence. Prior work (Kanvar & Khedker, 2016) has developed many hand-written heuristics for heap abstractions. Our approach replaces such heuristics with LM-guided, context-sensitive abstraction.

Abstract interpretation. Abstract interpretation (Cousot & Cousot, 1977) soundly approximates program behavior by tracking an abstract state that summarizes all possible concrete states. Each program operation updates the abstract state according to sound rules; for loops, iterative application yields a fixpoint that safely over-approximates all executions. For heap-manipulating programs, this requires a heap abstraction that merges potentially unbounded sets of objects into finite summary objects (Sagiv et al., 1998; Kanvar & Khedker, 2016). Traditional analyses rely on hand-crafted heuristics for when and how to introduce summaries. Our work instead uses a language model to guide these choices adaptively. (We provide a more detailed overview of abstract interpretation and heap abstractions in Appendix A.)

3.2 Abstract Interpretation

Abstract interpretation requires an abstract domain as well as modeling of the heap. In this section, we briefly describe our abstract domain, our two-level representation of the heap, and when we invoke the LM for summarization. The full analysis supports prototypal inheritance, recursion, loops, and closures. Additional details can be found in the appendix.

Abstract Domain. Our abstract domain keeps track of heap objects using concrete nodes and summary nodes. Summary nodes represent a set of possible concrete nodes. Each node is a dictionary from primitive or abstract values to other values. Our domain of primitive values is based off of TAJS (Jensen et al., 2009), one of the first abstract interpretation based analyses for Javascript. Additional details on our abstract domain can be found in the appendix. The most important runtime decision of ABSINT-AI is deciding when summarize heap nodes. We keep two separate heap structures, referred to as the local heap and global heap.

```
1  var global = 0;
2  var global_obj = {};
3  function inc_global() {
4      let obj = {f: 1};
5      obj.f += 1;
6      global = global + obj.f;
7  }
8  function access_obj() {
9      if (global > 10) {
10         var f = global_obj.foo.bar; // bug
11     }
12 }
13 var btn1 = document.createElement("button");
14 var btn2 = document.createElement("button");
15 btn1.addEventListener("click", inc_global);
16 btn2.addEventListener("click", access_obj);
```

Figure 2: `inc_global` needs to be run at least 10 times before the bug on line 11 is triggered.

Local heap. The local heap is used for precise representation for objects within local procedures, such as a local object allocation in a function call. It is flow-sensitive (Kildall, 1973), taking into account the order of statements. For example, in Figure 2, `obj` on line 4 is tracked in the local heap.

Global heap. The global heap is a much less precise representation for objects that are accessed and manipulated by multiple functions. The global heap captures all possible relationships between globally visible objects at any point in the execution. The global heap is motivated by flow-insensitive analysis (Weihl, 1980; Cousot & Cousot, 1977). This has two benefits: (1) It is much cheaper, as we don't have to keep track of a separate heap for each program location, and (2) it allows different functions to be analyzed independently; the global heap considers all the possible heap states at the point when the function is invoked, and the analysis of the function can reveal if any additional relationships need to be added to the global heap. Summarization only happens in the global heap.

We draw a distinction between the local and global heap because JavaScript programs tend to be reactive, with execution driven largely by external events. This has important implications for analysis, as the analysis can't assume the program will simply execute starting at the beginning from a well defined initial state. Take the example in Figure 2, where `inc_global` is invoked by an event handler and must be executed at least 10 times in order to trigger the bug on line 11. Keeping two separate heaps allows us to to track global dependencies while not losing precision for local procedures.

Agent Invocation. A key challenge in abstract interpretation is to reach a fixpoint without losing too much precision when analyzing potentially unbounded loops. Because fixpoint computation requires

merging abstract states across iterations, the choice of how to abstract heap objects allocated within the loop has a direct impact on both the precision and termination of the analysis.

Take the example in Figure 1. There are two objects, `box` and `config`. Each loop iteration allocates two objects: `box`, which is short-lived and well-suited to recency abstraction, and `config`, which contains a critical field (`valid`) that must remain precise. A uniform abstraction by allocation site would collapse these distinctions, introducing spurious behaviors. ABSINT-AI addresses this by invoking the LM agent at unbounded loops to choose abstraction strategies per object, balancing semantic precision with soundness and convergence.

3.3 AGENTIC HEAP ABSTRACTIONS

The agent in our framework serves as an interactive component embedded within the analysis loop. Its role is to select heap abstraction strategies, but unlike a static classifier, it behaves as an agent that operates under partial information and interacts with its environment to gather context before acting.

The agent is not invoked as a one-shot oracle. Instead, it operates as a environment-interacting agent that gathers information over time. To make informed abstraction decisions, the agent interacts with the abstract interpreter and the abstract state to selectively gather semantic information from the program. Rather than exposing the entire program or heap state, which would overwhelm the agent and obscure the relevant context, we treat the interpreter as a queryable environment. This avoids a common challenge in machine learning for code: programs often contain far more information than an LLM can meaningfully process, especially in settings with deep heap structure.

The agent's outputs are limited to a predefined set of sound abstraction strategies, and it never directly manipulates program state or executes code. The underlying abstract interpreter remains responsible for all semantic computation and fixpoint reasoning. This architectural separation allows us to embed an adaptive, learning-driven agent within a sound static analysis framework—enabling high-level decision-making informed by context and semantics, while preserving formal correctness guarantees.

Agent Interaction. The agent is initialized with the current abstract state, including visible variables, relevant allocation site data, and any previously encountered heap shapes. It then enters an interactive decision-making loop. During this loop, the agent can issue queries to the abstract state for more information, such as requesting variable values, inspecting function definitions, or examining the heap shape. If the available information is insufficient, the agent may also postpone its decision making by requesting additional abstract loop iterations, allowing it to observe how the heap evolves over time. This enables the agent to defer commitment while gathering contextual evidence.

Algorithm 1 Agentic Heap Abstraction Algorithm

Require: Loop $\mathcal{L}$, Analysis state $\mathcal{S}$, Allocation Sites $\mathcal{A}$
$b \leftarrow 0$ {Interaction counter (queries + executions)}
$\mathcal{A}' = \text{NONE}$
while $b <$ budget **do**
  Agent selects action $a \in \{\text{INFO}, \text{EXEC}, \text{SELECT}\}$
  if $a = \text{INFO}$ **then**
    Agent queries $\mathcal{S}$ for program information
    $b \leftarrow b+1$
  else if $a = \text{EXEC}$ **then**
    Interpreter executes one iteration of the loop
    $b \leftarrow b+1$
    continue
  else if $a = \text{SELECT}$ **then**
    Agent selects sites $\mathcal{A}' \subseteq \mathcal{A}$ to abstract
    break
  end if
end while
if $\mathcal{A}' = \text{NONE}$ **then**
  Agent selects $\mathcal{A}' \subseteq \mathcal{A}$ to abstract
end if
for $a_i \in \mathcal{A}'$ **do**
  Agent selects (Strategy,Parameters)
  Updated mapping in $\mathcal{S}$ from a_i to strategy for $\mathcal{L}$
end for

The interaction is bounded: the agent operates under a fixed query and iteration budget to ensure termination. Once satisfied, the agent returns a set of abstraction directives, specifying how the interpreter should merge and widen objects associated with each allocation site. The interpreter then executes the loop abstracting the heap as directed by the agent. If the abstract state does not reach a fixpoint within

five iterations, it re-queries the agent for new abstraction strategies. Algorithm 1 contains a detailed description of our procedure.

Information Gathering. The agent gathers information through a small set of read-only queries to ABSINT-AI:

- **Variable inspection**: Requesting abstract values of in-scope variables.
- **Function introspection**: Retrieving the definition of local functions in scope.
- **Loop execution**: Requesting additional iterations to observe how heap structures evolve.

These interactions allow the agent to incrementally reduce uncertainty and focus attention on semantically meaningful heap behaviors without drastically increasing the input size. In particular, loop execution supports deliberate abstraction delay, giving the agent a richer view of program dynamics before committing to a strategy.

Abstraction decisions. Once the agent has identified which allocation sites require abstraction, it selects a merging strategy for each. This determines how objects allocated at that site are grouped during join operations. The agent chooses from the following predefined strategies:

- **Allocation-site merge**: Collapses all objects created at the same program location into a single abstract object.
- **Recency merge**: Preserves the most recently allocated object at that site; merges older instances.
- **Field-sensitive merge**: Groups objects with the same fields.
- **Role-based merge**: Partitions objects based on semantically meaningful field values (e.g., role), allowing distinctions like "student" vs. "teacher" to be preserved.

In particular, role-based merging requires semantic understanding of field names and value meanings; it is very difficult to implement role-based merging using purely symbolic techniques. Identifying that a specific field should guide abstraction boundaries is often a decision that depends on natural language cues and program intent.

After selecting a merging strategy for an allocation site, the agent also specifies a widening strategy. Widening determines how abstract heap objects are generalized over time as they are revisited across loop iterations. The agent chooses from the following strategies:

- **Field-set widening**: widen a selected subset of fields, leave the others concrete.
- **Field merging**: Merge the fields together, and select another widening strategy for the values. This is for handling infinitely growing objects.
- **Full widening**: recursively widen the entire object into a single shape.
- **Depth-based widening**: Collapse structures beyond a fixed depth threshold

These strategies allow the agent to control the granularity of abstraction per object: preserving precise structure where it matters while widening aggressively in parts of the heap that are less semantically relevant. As with merging, widening strategies are selected per allocation site and parameterized to balance precision with scalability.

3.4 DOWNSTREAM TASK

As a downstream task to test the precision of ABSINT-AI, we detect the following situations (1) accessing a property of `null` or `undefined` and (2) reading an absent property of an object.

Abstracting unnecessarily can lead to false positives. Take the example in Figure 3. If `userId` on line 1 gets abstracted to the abstract NUMBER type, then the object access on line 3 is reported as a possible read of an absent property. `userId` could take the value of all possible numbers, but `names` only has the the property `100`.

```
1  let userId = 100; // abstracted to NUMBER.
2  let names = {100: "Jane"};
3  names[userId]; // False positive
```

Figure 3: False positive due to `userId` getting abstracted to the abstract NUMBER type.

Intersection of multiple runs. Different abstraction choices in a program can lead to different sets of reported bugs. For example, when analyzing the program in Figure 3, ABSINT-AI may choose to abstract the `userId` field in some runs but leave it concrete in others. This variation can affect

which false positives are reported. However, because each run is individually sound, any bug that does not appear in *any* run is guaranteed not to be real. This allows us to improve precision by taking the intersection of reported bugs across multiple runs (similar in spirit to self-consistency approaches (Wang et al., 2022b)) while preserving full soundness.

4 EVALUATION

Our evaluation focuses on two key questions: (1) How does our system perform compared to existing static analysis tools? (2) How important is agentic decision-making relative to fixed symbolic strategies or direct LLM prediction? To answer these, we compare against two established baselines (TAJS and WALA), conduct targeted ablations isolating the role of the agent, and present a case study demonstrating the system's ability to preserve meaningful heap structure.

4.1 BASELINES

TAJS. TAJS (Type Analysis for JavaScript) is a performs flow-sensitive, context-sensitive, and partially path-sensitive static analyzer designed for sound and scalable analysis of JavaScript programs Jensen et al. (2009). TAJS is based on abstract interpretation, including specialized heap abstractions such as allocation-site abstraction and recency abstraction, to model JavaScript's dynamic object behavior.

WALA. WALA (T. J. Watson Libraries for Analysis) is a general-purpose static analysis framework that supports multiple languages, including JavaScript Santos & Dolby (2022). Unlike TAJS, WALA is not based on abstract interpretation and performs flow-insensitive heap analysis, using a combination of allocation-site abstraction and context-sensitive pointer analysis.

Symbolic ABSINT-AI. We also include a baseline that runs ABSINT-AI using a fixed abstraction configuration without LM guidance. This baseline selects a conservative widening strategy across all allocation sites, simulating how our analysis would perform without agentic control. It serves to isolate the contribution of the LM-driven adaptivity from the underlying analysis framework. Symbolic ABSINT-AI begins with recency-based merging and a depth-1 field-sensitive abstraction. If the loop fails to converge within 50 iterations, it switches to widening the entire object while maintaining recency-based merging. If convergence still fails after another 50 iterations, it falls back to a fully allocation-site-based abstraction.

Dataset. To evaluate our approach, we curated a benchmark of 17 self-contained JavaScript programs from the Big Code dataset Raychev et al. (2016) and Github. We filtered for programs that were self-contained and did not use builtins excessively, as this greatly increases the imprecision of the analysis (`Math.floor`, for example, requires modeling the `Math` library to analyze precisely). These require substantial modeling effort and introduce orthogonal complexity. We also excluded object-oriented programs that rely too heavily on classes and `let` statements, since TAJS and WALA do not support Javascript features after ES2015. For context, prior work such as TAJS evaluated on 8 programs (Jensen et al., 2009), underscoring the difficulty of assembling larger benchmarks for sound JavaScript analysis. A detailed description of the dataset can be found in the Appendix.

4.2 PERFORMANCE

We evaluate ABSINT-AI using three different language models: GPT-4o-mini, GPT-4.1-mini, and Qwen3-32B. To compare against TAJS and WALA, we measure the number of (1) possible accesses to a property of `null` or `undefined` or (2) possible reads of an absent property of an object. In this setting, lower values indicate greater precision, reflecting fewer spurious results caused by imprecise heap abstraction. We run ABSINT-AI 10 times across our benchmark per model across our 17-program benchmark and report the mean results in Table 1.

Our agent-guided approach reports significantly fewer false positives than either baseline, achieving an average reduction of approximately 20%. This improvement stems from the agent's ability to select tailored abstraction strategies that avoid over-merging or premature widening, which often cause TAJS and WALA to lose key field or value distinctions.

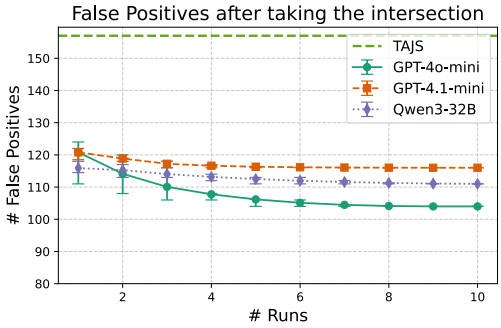


Figure 4: Running multiple times and taking the intersection of the reported bugs allows us to improve precision while maintaining soundness.

Table 1: Overall mean performance across the Dataset. #FP stands for False Positives. Fewer is better.

	Model	# FP↓	% Improve
Baselines	TAJS	157	0%
	WALA	312	-98.7%
	Symbolic ABSINT-AI	220	-28.6%
Mean	GPT-4o-mini	125	20.4%
	GPT-4.1-mini	127	19.1%
	Qwen3-32B	117	25.5%
Intersection	GPT-4o-mini	104	33.7%
	GPT-4.1-mini	116	26.1%
	Qwen3-32B	111	29.0%
	Full Intersection	97	38.2%

Intersection. As described in Section 3.4, one benefit of maintaining soundness is that we can safely take the intersection of reported errors across multiple runs, improving precision without risking missed bugs. Figure 4 shows the effect of taking intersections across multiple runs. As expected, the language model often makes different abstraction decisions, leading to partially overlapping sets of reported warnings. By intersecting the results across multiple runs, either for a single model or across all three, we can substantially reduce false positives. On average, intersecting runs from a single model improves precision by 8%; intersecting all 30 runs across all models yields a 13% reduction in false positives over any individual run. We find that intersecting the top 3–4 runs gives the steepest improvement, with diminishing returns after 6 runs.

Run time. We also compare the runtime performance of ABSINT-AI against TAJS and WALA. As expected, ABSINT-AI is slower, primarily due to our prototype implementation in Python, whereas both TAJS and WALA are written in Java. Much of the overhead comes from the interpreter itself, *not* from querying the agent. For example, when using GPT-4.1-mini, ABSINT-AI takes 500 seconds to run across our dataset, 189 of which is spent on agent interaction. In contrast, TAJS and WALA complete their analysis in approximately 20 seconds.

4.3 ABLATIONS

Ablation with symbolic abstractions. To isolate the contribution of the agent itself, we conducted an ablation study comparing ABSINT-AI to a purely symbolic variant that uses the same abstraction strategies but without agentic selection. In this setup, the analysis starts with the most precise abstractions and applies a fixed conservative widening strategy if the loop fails to converge within 10 iterations. If the analysis still does not converge after 20 minutes, we terminate and collect any reported warnings up to that point.

Table 1 shows that this symbolic version performs significantly worse: despite failing to converge on five benchmarks, it still produces 28.6% more false positives than TAJS. This highlights that the benefit of ABSINT-AI does not come merely from using expressive abstractions, but from the agent's ability to adaptively choose when and how to apply them based on program context.

Ablation with non-agent LLM. To isolate the impact of agentic interaction, we compare our full system to a variant that uses the same language model, but in a non-agentic, single-shot setting. In this baseline, the model is prompted to select abstraction strategies directly, without the ability to query the interpreter, inspect intermediate state, or request additional loop iterations. This version performs consistently worse than our full system, show that the ability for the model to gather evidence and defer commitment is important for robust and context-sensitive

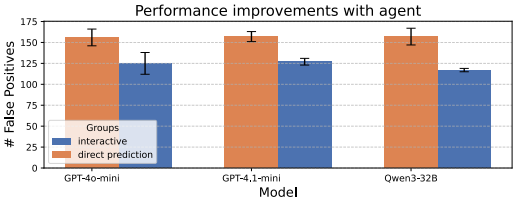


Figure 5: Performance improvements of an interactive agent vs. direct abstraction prediction.

decisions. As seen in Figure 5, the direct prediction consistently performs about 25% worse across our benchmarks.

4.4 Case Study on Conway's Game of Life

To illustrate the benefits of agent-guided abstraction, we present a case study from our benchmark based on Conway's Game of Life in Figure 6. The `cell_state` variable represents a 3×3 grid of integers, updated over `n` iterations by the `newGeneration` function. While the contents change, the structure remains fixed across iterations; a property inherent to the game's rules. AbsInt-AI identifies that only the integer values need to be abstracted, preserving the shape of the array and producing a precise heap abstraction.

```
var cell_state = [
  [0, 1, 0],
  [0, 1, 0],
  [0, 1, 0]
]
var n = parseInt($("#iterations"));
for (var i = 0; i < n; i++) {
  cell_state = newGeneration(cell_state);
}
```

Figure 6: A snippet from Conway's Game of Life.

In contrast, symbolic baselines often over-abstract the structure itself, prematurely merging array shapes and losing row-level distinctions. This highlights how the agent draws on both program syntax and semantic cues such as common data patterns to guide more precise abstraction decisions.

5 Related Work

LMs in program analysis. LMs have been applied to a wide range of program analysis tasks, including type inference, fuzzing, vulnerability and resource leak detection, code summarization, and fault localisation (Peng et al., 2023; Wei et al., 2023; Wang et al., 2023b; Xia et al., 2024; Yang et al., 2023b;a; Deng et al., 2023; Mathews et al., 2024; Liu et al., 2023; Wang et al., 2023a; Mohajer et al., 2023; Cai et al., 2023; Geng et al., 2024; Ahmed et al., 2024; Wang et al., 2022a; Wu et al., 2023). However, none have been applied to static analysis while preserving soundness guarantees. More recently, several neurosymbolic approaches combine static analysis with LMs: LLift (Li et al., 2024a) filters false positives from UBITect (Zhai et al., 2020), IRIS (Li et al., 2024b) augments CodeQL (Avgustinov et al., 2016) for taint analysis, and InferROI (Wang et al., 2024) detects resource leaks in Java programs. While effective at improving precision, all of these systems sacrifice soundness once neural predictions are introduced.

Program analysis for Javascript. Much prior work on JavaScript analysis has focused on unsound but pragmatic tools for bug finding and security. These tools aim to detect likely vulnerabilities or errors in real-world programs, often trading soundness for scalability and precision (Li et al., 2022; Fass et al., 2019; Kang et al., 2023; Yu et al., 2023; Guo et al., 2024; Kang et al., 2025). While effective for finding particular security issues in practice, these approaches do not provide soundness guarantees. As a result, they are not suitable for many downstream tasks that depend on full program coverage, such as compiler optimizations or transformations, where missing even a single feasible behavior can invalidate correctness. Our work, by contrast, maintains the formal soundness of abstract interpretation while improving its precision via adaptive heap abstraction.

6 Limitations and Conclusion

Scalability. A limitation of AbsInt-AI is that it does not scale to large JavaScript codebases (e.g., 2,000+ lines). This is a broader issue with JavaScript static analysis: neither TAJS nor WALA converged on such programs in our experiments. The challenge stems from the dynamic and object-heavy nature of real-world JavaScript. While our agent-guided approach adds adaptivity, our prototype and reliance on whole-program analysis similarly limit scalability. Addressing this is an important direction for future work.

In this work, we propose a method to augment static analyzers with an agentic LM for heap abstractions. We present AbsInt-AI as a proof-of-concept and an evaluation showing that augmenting static analysis with LMs can have a dramatic improvement on the precision without losing soundness guarantees.

7 Reproducibility Statement

We have included our source code along with instructions to reproduce the experiments in the supplementary material.

References

URL https://angular.dev/tools/cli/aot-compiler.

Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T. Barr. Automatic semantic augmentation of language model prompts (for code summarization), 2024.

Gábor Antal, Péter Hegedűs, Zoltán Herczeg, Gábor Lóki, and Rudolf Ferenc. Is javascript call graph extraction solved yet? a comparative study of static and dynamic tools. *IEEE Access*, 11: 25266–25284, 2023. doi: 10.1109/ACCESS.2023.3255984.

Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. Ql: Object-oriented queries on relational data. In *European Conference on Object-Oriented Programming*, 2016. URL https://api.semanticscholar.org/CorpusID:13385963.

brettz9. Brettz9/espree: An esprima-compatible javascript parser. URL https://github.com/brettz9/espree.

Yufan Cai, Yun Lin, Chenyan Liu, Jinglian Wu, Yifan Zhang, Yiming Liu, Yeyun Gong, and Jin Song Dong. On-the-fly adapting code summarization on trainable cost-effective language models. In A. Oh, T. Neumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 56660–56672. Curran Associates, Inc., 2023.

Satish Chandra, Colin S Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Youngil Choi. Type inference for static compilation of javascript. *ACM SIGPLAN Notices*, 51(10):410–429, 2016.

Patrick Cousot and Radhia Cousot. 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*, pp. 238–252, 1977.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models, 2023.

Aurore Fass, Michael Backes, and Ben Stock. Jstap: A static pre-filter for malicious javascript detection. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pp. 257–269, 2019.

Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript ide services. In *2013 35th International Conference on Software Engineering (ICSE)*, pp. 752–761, 2013. doi: 10.1109/ICSE.2013.6606621.

Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3608134. URL https://doi.org/10.1145/3597503.3608134.

Zhiyong Guo, Mingqing Kang, VN Venkatakrishnan, Rigel Gjomemo, and Yinzhi Cao. Reactappscan: Mining react application vulnerabilities via component graph. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pp. 585–599, 2024.

Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 54–61, 2001.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In *Static Analysis: 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings 16*, pp. 238–255. Springer, 2009.

Mingqing Kang, Yichao Xu, Song Li, Rigel Gjomemo, Jianwei Hou, VN Venkatakrishnan, and Yinzhi Cao. Scaling javascript abstract interpretation to detect and exploit node. js taint-style vulnerability. In *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 1059–1076. IEEE, 2023.

Zifeng Kang, Muxi Lyu, Zhengyu Liu, Jianjia Yu, Runqi Fan, Song Li, and Yinzhi Cao. Follow my flow: Unveiling client-side prototype pollution gadgets from one million real-world websites. In *2025 IEEE Symposium on Security and Privacy (SP)*, pp. 991–1008. IEEE, 2025.

Vini Kanvar and Uday P. Khedker. Heap abstractions for static analysis. *ACM Computing Surveys*, 49 (2):1–47, June 2016. ISSN 1557-7341. doi: 10.1145/2931098. URL http://dx.doi.org/10.1145/2931098.

Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 194–206, 1973.

Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proc. ACM Program. Lang.*, 8(OOPSLA1), April 2024a. doi: 10.1145/3649828. URL https://doi.org/10.1145/3649828.

Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Mining node. js vulnerabilities via object dependence graph and query. In *31st USENIX Security Symposium (USENIX Security 22)*, pp. 143–160, 2022.

Ziyang Li, Saikat Dutta, and Mayur Naik. Llm-assisted static analysis for detecting security vulnerabilities, 2024b. URL https://arxiv.org/abs/2405.17238.

Puzhuo Liu, Chengnian Sun, Yaowen Zheng, Xuan Feng, Chuan Qin, Yuncheng Wang, Zhi Li, and Limin Sun. Harnessing the power of llm to support binary taint analysis, 2023.

Noble Saji Mathews, Yelizaveta Brus, Yousra Aafer, Meiyappan Nagappan, and Shane McIntosh. Llbezpeky: Leveraging large language models for vulnerability detection, 2024.

Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. Skipanalyzer: A tool for static code analysis with large language models, 2023.

Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R. Lyu. Generative type inference for python, 2023.

Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. *SIGPLAN Not.*, 51(1):761–774, January 2016. ISSN 0362-1340. doi: 10.1145/2914770.2837671. URL https://doi.org/10.1145/2914770.2837671.

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(1):1–50, 1998.

Joanna CS Santos and Julian Dolby. Program analysis using wala (tutorial). In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1819–1819, 2022.

Paul B Schneck. A survey of compiler optimization techniques. In *Proceedings of the ACM annual conference*, pp. 106–113, 1973.

Manuel Serrano. On javascript ahead-of-time compilation performance (keynote). In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*, pp. 1–1, 2022.

Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of javascript. In James Noble (ed.), *ECOOP 2012 – Object-Oriented Programming*, pp. 435–458, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-31057-7.

Shashank Srikant, Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, Gaoyuan Zhang, and Una-May O'Reilly. Generating adversarial computer programs using optimized obfuscations. *arXiv preprint arXiv:2103.11882*, 2021.

Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R. Lyu. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pp. 382–394, New York, NY, USA, 2022a. Association for Computing Machinery. ISBN 9781450394130. doi: 10.1145/3540250.3549113. URL https://doi.org/10.1145/3540250.3549113.

Chong Wang, Jianan Liu, Xin Peng, Yang Liu, and Yiling Lou. Llm-based resource-oriented intention inference for static resource leak detection, 2023a.

Chong Wang, Jianan Liu, Xin Peng, Yang Liu, and Yiling Lou. Boosting static resource leak detection via llm-based resource-oriented intention inference, 2024. URL https://arxiv.org/abs/2311.04448.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022b.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation, 2023b.

Jiayi Wei, Greg Durrett, and Isil Dillig. Typet5: Seq2seq type inference using static analysis, 2023.

William E Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 83–94, 1980.

Yonghao Wu, Zheng Li, Jie M. Zhang, Mike Papadakis, Mark Harman, and Yong Liu. Large language models in fault localisation, 2023.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models, 2024.

Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. White-box compiler fuzzing empowered by large language models, 2023a.

Chenyuan Yang, Zijie Zhao, and Lingming Zhang. Kernelgpt: Enhanced kernel fuzzing via large language models, 2023b.

Jianjia Yu, Song Li, Junmin Zhu, and Yinzhi Cao. Coco: Efficient browser extension vulnerability detection via coverage-guided, concurrent abstract interpretation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2441–2455, 2023.

Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, pp. 39–51, 2022.

Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V. Krishnamurthy, and Paul Yu. Ubitect: a precise and scalable method to detect use-before-initialization bugs in linux kernel. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pp. 221–232, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3409686. URL https://doi.org/10.1145/3368089.3409686.

A BACKGROUND

Soundness and precision. Traditional static program analysis is often split between sound and unsound analyses. Soundness is the quality of static analyzers which guarantees that the analysis models an *over-approximation* of the target program's behavior, but may model behaviors that do not actually occur in any execution. The *precision* of the analysis is the extent to which the analysis avoids such spurious results. In short, a program analysis is *sound* if there are no false negatives. A program analysis is *precise* if there are not many false positives.

Abstractions in static analysis. Static analysis algorithms achieve scalability and soundness by using *abstractions* in their analysis. Programs often manipulate unbounded resources (e.g., integers, heap structures). Abstractions merge a potentially infinite set of objects into a single *summary* object to ensure convergence and for scalability. A key challenge is choosing *what* to abstract in the target program to ensure convergence while retaining as much important information as possible. There has been a rich body of literature on improving precision and scalability of heap abstractions (Kanvar & Khedker, 2016). In this work, we use an LM to decide what should be abstracted in the target program.

Abstract Interpretation. Abstract interpretation is a framework for analyzing programs by soundly approximating their behavior through the use of an *abstract state* that summarizes the set of possible states that a program can be in at different points in the execution (Cousot & Cousot, 1977). For simple programs manipulating scalar values, the abstract state is usually a simple mapping from variable names to abstract values representing sets of numbers. For example, an integer variable may be assigned the abstract value POSITIVE, representing all positive integers, to indicate the fact that its concrete value is guaranteed to be a positive value on any execution of the program. Abstract interpretation works by interpreting the program using rules that describe how each operation available in the language transforms the abstract state into new abstract states. For example, a rule may indicate that the addition of two POSITIVE numbers always results in a positive number. Soundness of the analysis is guaranteed by ensuring the soundness of each individual rule; for programs with loops, the analysis needs to be executed iteratively, and the theory of abstract interpretation ensures that once the abstract states converge to a fixpoint, this fixpoint will be a sound representation of the set of possible states that any execution of the program can reach.

For heap manipulating programs, the abstract state must include an abstraction of the heap which represents all the possible states of the heap a program might exhibit at a given point in time (Sagiv et al., 1998). There is an extensive literature on heap abstractions (Kanvar & Khedker, 2016), but all of them have a few elements in common. One important element is the use of *summarization* to represent multiple objects which may be living in the heap at a given point in the execution as a single *summary object*. Summarization allows the analysis to use a bounded representation for the potentially unbounded set of objects that can live on the heap on any arbitrary execution. Traditional abstract interpretation frameworks rely on complex heuristics to determine when and how to introduce summary nodes during program analysis to allow the analysis to maintain precision while quickly converging to a reasonably sized representation of the abstract heap. Our goal for this work is to replace those heuristics with an LM which can take advantage of its background knowledge of concepts used in the code as expressed through variable names, field names and comments.

B ABSTRACT INTERPRETATION DETAILS

B.1 ANALYSIS DETAILS

Functions In Javascript, functions are stored as objects on the heap. We include a `__code__` property storing the function body to be executed. At the beginning of the analysis, ABSINT-AI scans the entire program, and generates a *schema* for each function. The schema for each function contains which variables are local to the function and which variables are accessed by other functions. We refer to variables that are local as *private*, and variables that are accessed by other functions as *shared*. Each time a function is executed, an environment is initialized according to the schema for that function. When a function is defined, is initialized with a `__hf__` field set to the current heap frame.

The `__hf__` field is used to model scopes and closures. When the function returns, the stack frame σ is popped from the stack, and the stack pointer is decremented.

Scopes and Closures Whenever a function is called, a new stack frame σ is pushed, along with a corresponding heap frame. The stack pointer for the current stack frame is updated to point to σ. The private variables for that function are stored in the stack frame σ, and any shared variables are stored in the heap frame. The heap frame is initialized with a parent field `__parent__` which is used to model the scope chain. The `__parent__` field points to the `__hf__` field for the function being initialized.

To lookup a variable name in the environment, ABSINT-AI first checks the current stack frame. If it finds a value for the variable, it returns the value. If it doesn't, it checks the corresponding heap frame for the stack frame, and then follows the chain of `__parent__` pointers until it finds the variable.

Recursion ABSINT-AI keeps track of all functions that have been called but have not finished executing yet. Whenever it encounters a recursive call, ABSINT-AI sets the return value to a recursive placeholder and stores a hash of the function that is called. When the function returns, ABSINT-AI checks the return values and any allocated heap objects for recursive placeholders for the function and fills them in with the return values.

B.2 ENVIRONMENT

In this section we describe how ABSINT-AI represents the abstract state. We define concrete and abstract values. H_L refers to the concrete heap, H_G refers to the global heap, and σ refers to the stack. τ is an abstract type, C refers to constants, obj and $\widetilde{obj}$ refer to concrete and abstract objects. val and $\widetilde{val}$ refer to the values that a variable can take.

$$
\begin{array}{rcl}
val & ::= & a \mid obj \mid \widetilde{val} \\
\widetilde{val} & ::= & C \mid \widetilde{a} \mid \tau \mid \widetilde{obj} \\
\tau & ::= & Bool \mid Null \mid Num \mid String \\
obj & ::= & \tau \rightarrow val \mid C \rightarrow val \\
\widetilde{obj} & ::= & \tau \rightarrow \widetilde{val} \mid C \rightarrow \widetilde{val} \\
H_L & ::= & a \rightarrow val \\
H_G & ::= & \widetilde{a} \rightarrow \widetilde{val} \\
\sigma & ::= & C \rightarrow val
\end{array}
$$

B.3 SYNTAX

$$
\begin{array}{rcl}
op & ::= & + \mid - \mid \div \mid \cdot \mid ... \\
E & ::= & id \mid E.field \mid E[E] \mid foo(E) \mid E_1[E_2](E_3,E4,...) \mid \text{function}(x_0,x_1,...)\{S\} \\
 & & \mid \text{new } foo(E_1,E_2,...) \mid C \mid \{f:E\} \\
varDef & ::= & \text{var } id=E \mid \text{let } id=E \mid \text{const } id=E \\
Stmt & ::= & varDef \mid id=E \mid \\
 & & E.f=E \mid E[E]=E \mid \text{def } foo(x_1,x_2,...,x_n)\{Stmt\} \mid \\
 & & \text{if } (E)\{Stmt\} \text{ else } \{Stmt\} \mid \text{class } foo\{Stmt\} \mid \\
 & & \text{return } E \mid \text{for (varDef; E; Stmt)}\{Stmt\} \\
 & & \text{for (varDef in E)}\{Stmt\} \mid \text{while } (E)\{Stmt\} \mid Stmt;Stmt
\end{array}
$$

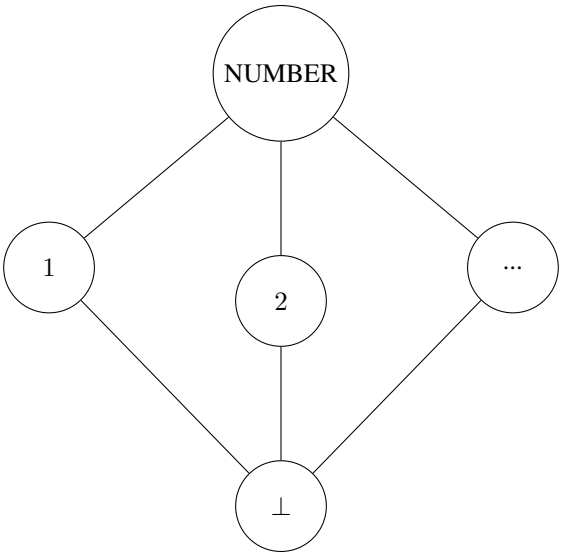


Figure 7: Number Lattice.

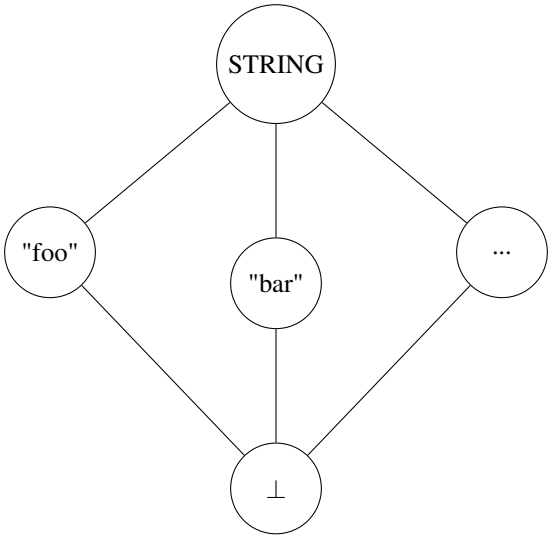


Figure 8: String Lattice.

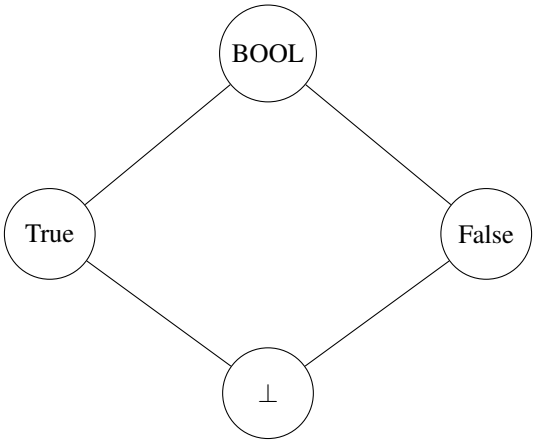


Figure 9: Boolean Lattice.

Figure 10: Null Singleton.

B.4 SEMANTICS

B.4.1 FUNCTIONS

This section is several functions we use, such looking up a variable name and initializing a new schema for a function.

lookup(id)

$$\frac{s \equiv \emptyset \quad \theta = \emptyset}{\langle lookup(H_L, H_G, s, id) \to \theta \rangle}$$

$$\frac{s \in H_L \quad id \in H_L(s) \quad \theta = s}{\langle lookup(H_L, H_G, s, id) \to \theta \rangle}$$

$$\frac{s \in H_G \quad id \in H_G(s) \quad \theta = s}{\langle lookup(H_L, H_G, s, id) \to \theta \rangle}$$

$$\frac{s \in H_L \quad id \notin H_L(s) \quad \theta = lookup(H_L, H_G, H_L(s).par, id)}{\langle lookup(H_L, H_G, s, id) \to \theta \rangle}$$

$$\frac{s \in H_G \quad id \notin H_G(s) \quad \theta = lookup(H_L, H_G, H_G(s).par, id)}{\langle lookup(H_L, H_G, s, id) \to \theta \rangle}$$

initialize(schema)

$$\frac{H_L[a \mapsto \{schema.public, par \mapsto \sigma.hf\}] \quad \sigma'.\_secret \mapsto \{schema.secret\} \quad \sigma'.hf \mapsto a}{initialize(schema) \to H_L, H_G, \sigma :: \sigma'}$$

return_from_schema

$$\frac{\sigma \equiv \sigma' :: v}{return\_from\_schema \to H_L, H_G, \sigma'}$$

B.4.2 SMALL-STEP SEMANTICS

$$\langle H_L, H_G, \sigma, S \rangle \to \langle H'_L, H'_G, \sigma', S' \rangle$$

$$\text{id}\quad \frac{}{\langle H_L,H_G,\sigma,id\rangle \to \langle H_L,H_G,\sigma,lookup(id)\rangle}$$

$$\text{E.field}\quad \frac{\langle H_L,H_G,\sigma,E\rangle \to \langle H'_L,H'_G,\sigma',V\rangle}{\langle H_L,H_G,\sigma,\text{E.field}\rangle \to \langle H'_L,H'_G,\sigma',\text{get(V,field)}\rangle}$$

$$E_1[E_2]\quad \frac{\langle H_L,H_G,\sigma,E_2\rangle \to \langle H'_L,H'_G,\sigma',V_2\rangle \quad \langle H'_L,H'_G,\sigma',E_1\rangle \to \langle H''_L,H''_G,\sigma'',V_1\rangle}{\langle H_L,H_G,\sigma,E_1[E_2]\rangle \to \langle H'_L,H'_G,\sigma',\text{get}(V_1,V_2)\rangle}$$

$$foo(E_0,E_1,\ldots)\quad \frac{\langle lookup(foo)\to V, V.\_\_type \equiv Function\rangle \quad \langle H_L,H_G,\sigma,E_0,E_1,\ldots\rangle \to \langle H'_L,H'_G,\sigma',V_0,V_1,\ldots\rangle}{\langle H_L,H_G,\sigma,foo(E_0,E_1,\ldots)\rangle \to \langle H'_L[x_0\mapsto V_0,x_1\mapsto V_1,\ldots],H'_G,\sigma',initialize(V.\_\_code);V.\_\_code\rangle}$$

$$E_1[E_2](E_3,E_4,\ldots)\quad \frac{\langle H_L,H_G,\sigma,E_0,E_1,\ldots\rangle \to \langle H'_L,H'_G,\sigma',V_0,V_1,V_2,\ldots\rangle \quad \langle get(V_0,V_1)\to V, V.\_\_type \equiv Function\rangle}{\langle H_L,H_G,\sigma,foo(E_0,E_1,\ldots)\rangle \to \langle H'_L[x_0\mapsto V_0,x_1\mapsto V_1,\ldots],H'_G,\sigma'[this\mapsto V_0],V.\_\_code\rangle}$$

$$\text{function}(x_0,x_1,\ldots)\{S\}\quad \frac{}{\langle H_L,H_G,\sigma,\text{function}(x_0,x_1,\ldots)\rangle \to \langle H'_L[a'\mapsto\{\ldots,prototype:a\},a\mapsto],H'_G,\sigma',a'\rangle}$$

$$\text{new foo}(E_0,E_1,\ldots)\quad \frac{\langle lookup(foo)\to V\rangle \quad \langle V.\_\_type\equiv Class\rangle \quad \langle E_0,E_1,\ldots\rangle \to \langle V_0,V_1,\ldots\rangle}{\langle H_L,H_G,\sigma,\text{new foo}(E_0,E_1,\ldots)\rangle \to \langle H'_L,H'_G,\sigma'[this\mapsto V],init();get(prototype(V),constructor)(V_0,V_1,\ldots)\rangle}$$

$$\frac{\langle lookup(foo)\to V\rangle \quad \langle V.\_\_type\equiv Function\rangle \quad \langle E_0,E_1,\ldots\rangle \to \langle V_0,V_1,\ldots\rangle}{\langle H_L,H_G,\sigma,\text{new foo}(E_0,E_1,\ldots)\rangle \to \langle H'_L,H'_G,\sigma',V.\_\_code(V_0,V_1,\ldots)\rangle}$$

$$\{f_1:E_1,f_2:E_2,\ldots\}\quad \frac{\langle H_L,H_G,\sigma,E_1,E_2,\ldots\rangle \to \langle H'_L,H'_G,\sigma',V_1,V_2,\ldots\rangle}{\langle H_L,H_G,\sigma,\{f_1:E_1,f_2:E_2,\ldots\}\rangle \to \langle H_L[a\mapsto\{f_1:V_1,f_2:V_2,\ldots,\_\_type:\text{object}\}],H_G,\sigma,a\rangle}$$

$$\text{(var x = E)}\quad \frac{\langle H_L,H_G,\sigma,E\rangle \to \langle H'_L,H'_G,\sigma',V\rangle \quad \theta = lookup(x) \quad \theta\in H_L \quad \text{fr} = H_L[\theta] \quad \text{fr'} = fr[id\mapsto V]}{\langle H_L,H_G,\sigma,x=E\rangle \to \langle H'_L[\theta\mapsto fr'],H'_G,\sigma',skip\rangle}$$

$$\frac{\langle H_L,H_G,\sigma,E\rangle \to \langle H'_L,H'_G,\sigma',V\rangle \quad \theta = lookup(x) \quad \theta\in H_G \quad \text{fr} = H_G[\theta] \quad \text{fr'} = fr[id\mapsto V\cup fr[id]]}{\langle H_L,H_G,\sigma,x=E\rangle \to \langle H'_L,H'_G[\theta\mapsto fr'],\sigma',skip\rangle}$$

$$\text{(x.f = E)}\quad \frac{lookup(x)\equiv a \quad \theta = H_L(a) \quad \langle H_L,H_G,\sigma,E\rangle \to \langle H'_L,H'_G,\sigma',V\rangle}{\langle H_L,H_G,\sigma,x.f=E\rangle \to \langle H'_L[\theta[f\mapsto V]],H'_G,\sigma'],skip\rangle}$$

$$\frac{lookup(x)\equiv \widetilde{a} \quad \widetilde{\theta} = H_G(\widetilde{a}) \quad \langle H_L,H_G,\sigma,E\rangle \to \langle H'_L,H'_G,\sigma',V\rangle}{\langle H_L,H_G,\sigma,x=E\rangle \to \langle H'_L,H'_G[\widetilde{\theta}[f\mapsto V],\sigma',skip\rangle}$$

(x[E] = E')

$$\frac{lookup(x) \equiv a \quad \theta = H_L(a) \quad \langle H_L, H_G, \sigma, E, E' \rangle \to \langle H'_L, H'_G, \sigma', V, V' \rangle}{\langle H_L, H_G, \sigma, x[f] = E \rangle \to \langle H'_L[\theta[V \mapsto V']], H'_G, \sigma' \rangle, skip \rangle}$$

$$\frac{lookup(x) \equiv \widetilde{a} \quad \widetilde{\theta} = H_G(\widetilde{a}) \quad \langle H_L, H_G, \sigma, E, E' \rangle \to \langle H'_L, H'_G, \sigma', V, V' \rangle}{\langle H_L, H_G, \sigma, x = E \rangle \to \langle H'_L, H'_G[\widetilde{\theta}[V \mapsto V'], \sigma', skip \rangle}$$

(def foo($x_0, x_1, \ldots, x_n$){Stmt})

$$\frac{\theta = lookup(foo) \quad \theta \in \sigma}{\langle H_L, H_G, \sigma, x[f] = E \rangle \to \langle H_L[a \mapsto \ldots, prototype : a', a' \mapsto \{\}], H_G, \sigma[\theta \mapsto a], skip \rangle}$$

$$\frac{\theta = lookup(foo) \quad \theta \in H_L}{\langle H_L, H_G, \sigma, x[f] = E \rangle \to \langle H_L[a \mapsto \ldots, prototype : a', a' \mapsto \{\}, \theta \mapsto a], H_G, \sigma, skip \rangle}$$

$$\frac{\theta = lookup(foo) \quad \theta \in H_G}{\langle H_L, H_G, \sigma, x[f] = E \rangle \to \langle H_L, H_G[a \mapsto \ldots, prototype : a', a' \mapsto \{\}, \theta \mapsto \theta \cup a], \sigma, skip \rangle}$$

(x[E] = E')

$$\frac{lookup(x) \equiv a \quad \theta = H_L(a) \quad \langle H_L, H_G, \sigma, E, E' \rangle \to \langle H'_L, H'_G, \sigma', V, V' \rangle}{\langle H_L, H_G, \sigma, x[f] = E \rangle \to \langle H'_L[\theta[V \mapsto V']], H'_G, \sigma' \rangle, skip \rangle}$$

$$\frac{lookup(x) \equiv \widetilde{a} \quad \widetilde{\theta} = H_G(\widetilde{a}) \quad \langle H_L, H_G, \sigma, E, E' \rangle \to \langle H'_L, H'_G, \sigma', V, V' \rangle}{\langle H_L, H_G, \sigma, x = E \rangle \to \langle H'_L, H'_G[\widetilde{\theta}[V \mapsto V'], \sigma', skip \rangle}$$

if (E) { Stmt }) else { Stmt' }

$$\frac{\langle H_L, H_G, \sigma, E \rangle \to \langle H'_L, H'_G, \sigma', False \vee \emptyset \rangle}{\langle H_L, H_G, \sigma, \text{if (E)} \{Stmt\} \text{ else } \{Stmt;\} \rangle \to \langle H'_L, H'_G, \sigma' \rangle, Stmt \rangle}$$

$$\frac{\langle H_L, H_G, \sigma, E \rangle \not\to \langle H'_L, H'_G, \sigma', False \vee \emptyset \rangle}{\langle H_L, H_G, \sigma, \text{if (E)} \{Stmt\} \text{ else } \{Stmt'\} \rangle \to \langle H'_L, H'_G, \sigma' \rangle, Stmt' \rangle}$$

class foo[$M_1, M_2, \ldots, M_N$]

$$\frac{class\_obj = \{M_1, M_2, \ldots, M_N\}}{\langle H_L, H_G, \sigma, \text{class foo}[M_1, M_2, \ldots, M_N] \rangle \to \langle H_L[a \mapsto class\_obj], H_G, \sigma, skip \rangle}$$

(return E)

$$\frac{\langle H_L, H_G, \sigma, E \rangle \to \langle H'_L, H'_G, \sigma', V \rangle}{\langle H_L, H_G, \sigma, returnE \rangle \to \langle H'_L, H'_G, \sigma'[returns \mapsto \sigma'[returns] \cup V], skip \rangle}$$

for ([let | var] id in E) $\{Stmt\}$

$$\frac{\langle H_L, H_G, \sigma, E \rangle \to \langle H'_L, H'_G, \sigma', V \rangle \quad V.\_\_proto\_\_ \equiv \emptyset \quad isEmpty(V) \equiv True}{\langle H_L, H_G, \sigma, \text{for ([let | var] id in E) } \{Stmt\} \rangle \to \langle H'_L, H'_G, \sigma', skip \rangle}$$

$$\frac{\langle H_L, H_G, \sigma, E \rangle \to \langle H'_L, H'_G, \sigma', V \rangle \quad V.\_\_proto\_\_ \not\equiv \emptyset \quad isEmpty(V) \equiv True}{\langle H_L, H_G, \sigma, \text{for ([let | var] id in E) } \{Stmt\} \rangle \to \langle H'_L, H'_G, \sigma', \text{for ([let | var] id in V.\_\_proto\_\_) } \{\text{Stmt}\} \rangle}$$

$$\frac{\langle H_L, H_G, \sigma, E \rangle \to \langle H'_L, H'_G, \sigma', V \rangle \quad V \equiv X :: V' \quad \text{varDef.type} \equiv let}{\langle H_L, H_G, \sigma, \text{for (let id in E) } \{Stmt\} \rangle \to \langle H''_L, H''_G, \sigma'', \text{initialize(Stmt);let id=X;Stmt; for (let id in V') \{ Stmt \}} \rangle}$$

$$\frac{\langle H_L, H_G, \sigma, E \rangle \to \langle H'_L, H'_G, \sigma', V \rangle \quad V \equiv X :: V' \quad \text{varDef.type} \equiv var}{\langle H_L, H_G, \sigma, \text{for (let id in E) } \{Stmt\} \rangle \to \langle H'_L, H'_G, \sigma', \text{var id=X;Stmt; for (let id in V') \{ Stmt \}} \rangle}$$

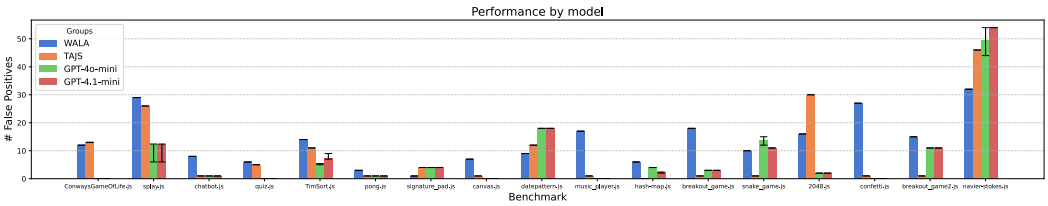


Figure 11: Performance per model on each benchmark program compared to WALA and TAJS.

$$\text{while (E) } \{Stmt\} \quad \frac{\langle H_L, H_G, \sigma, E\rangle \to \langle H'_L, H'_G, \sigma', V\rangle \quad V \in \text{Falsey}}{\langle H_L, H_G, \sigma, \text{while (E) } \{Stmt\}\rangle \to \langle H'_L, H'_G, \sigma', skip\rangle}$$

$$\frac{\langle H_L, H_G, \sigma, E\rangle \to \langle H'_L, H'_G, \sigma', V\rangle \quad V \notin \text{Falsey} \quad \langle H'_L, H'_G, \sigma', \text{Stmt;summarize()} \to \langle H'_L, H'_G, \sigma'\rangle}{\langle H_L, H_G, \sigma, \text{while (E) } \{Stmt\}\rangle \to \langle H'_L, H'_G, \sigma', skip\rangle}$$

$$\frac{\langle H_L, H_G, \sigma, E\rangle \to \langle H'_L, H'_G, \sigma', V\rangle \quad V \notin \text{Falsey} \quad \langle H'_L, H'_G, \sigma', \text{Stmt;summarize()} \to \langle H''_L, H''_G, \sigma''\rangle}{\langle H_L, H_G, \sigma, \text{while (E) } \{Stmt\}\rangle \to \langle H''_L, H''_G, \sigma'', \text{while (E) } \{Stmt\}\rangle}$$

C Implementation and Dataset

Implementation. We implemented AbsInt-AI in 8049 lines of Python, and use Espree brettz9 to parse the Javascript into an AST. We conducted the experiments on a Linux server with two AMD EPYC 7763 64-Core Processors, 128 cores, 1024GB RAM, and 4 NVIDIA RTX 6000 Ada Generation GPUs.

C.1 Dataset

Table 2: Each program and a small description.

Program	#Lines	Description
CGOL.js	65	Conway's Game of Life.
2048.js	234	The 2048 game implemented for the DOM.
breakout_game.js	158	An implementation of the Breakout arcade game for the DOM.
breakout_game2.js	91	A separate implementation of the Breakout arcade game for the DOM.
datepattern.js	91	Testing date string equality
hash-map.js	577	A JavaScript implementation of a HashMap.
confetti.js	400	Confetti animations in the DOM.
pong.js	243	Pong game in the DOM.
snake_game.js	102	Snake game in the DOM.
books.js	504	A library for storing books.
FlashSort.js	84	Flash Sort.
math_sprint.js	345	Math calculations in the DOM.
drawing-app.js	442	A drawing app in the DOM.
TimSort.js	113	Tim Sort.
navier-stokes.js	385	Fluid dynamics simulation using a simplified implementation of the Navier–Stokes equations.
music_player.js	196	Picking between songs to display in the DOM.
splay.js	406	An implementation of a Splay Tree in JavaScript.

D LLM Usage

We used a large language model (ChatGPT, GPT-5, OpenAI) to assist with polishing the writing and improving clarity of exposition. The model was not used to design the methodology, conduct

experiments, or generate results. All technical contributions, data analysis, and conclusions are the authors' own.