

# 000 001 002 003 004 005 ABSINT-AI: AGENTIC HEAP ABSTRACTIONS FOR 006 ABSTRACT INTERPRETATION 007 008 009

010 **Anonymous authors**  
011 Paper under double-blind review  
012  
013  
014  
015  
016  
017  
018  
019  
020  
021  
022  
023  
024  
025  
026  
027  
028  
029

## 030 ABSTRACT 031

032 Static program analysis is a foundational technique in software engineering for  
033 reasoning about program behavior. Traditional static analysis algorithms model  
034 programs as logical systems with well-defined semantics, but rely on uniform,  
035 hard-coded heap abstractions. This limits their precision and flexibility, especially  
036 in dynamic languages like JavaScript, where heap structures are heterogeneous  
037 and difficult to analyze statically. In this work, we introduce ABSINT-AI, a  
038 language-model-guided static analysis framework that augments abstract interpre-  
039 tation with adaptive, per-object heap abstractions for Javascript. This enables  
040 the analysis to leverage high-level cues, such as naming conventions and access  
041 patterns, without requiring brittle, hand-engineered heuristics. Importantly, the LM  
042 agent operates within a bounded interface and never directly manipulates program  
043 state, preserving the soundness guarantees of abstract interpretation. To evaluate  
044 our approach, we focus on a soundness-critical task: determining whether object  
045 property accesses may result in undefined or null dereferences. This task directly  
046 models a common requirement in compiler optimizations, where proving that an  
047 access is safe enables the removal of dynamic checks or simplifies code motion. On  
048 this task, ABSINT-AI reduces false positives by up to 34% compared to traditional  
049 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%.

## 050 1 INTRODUCTION 051

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

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

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

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

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

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

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

## 089 2 MOTIVATING EXAMPLE

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

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

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

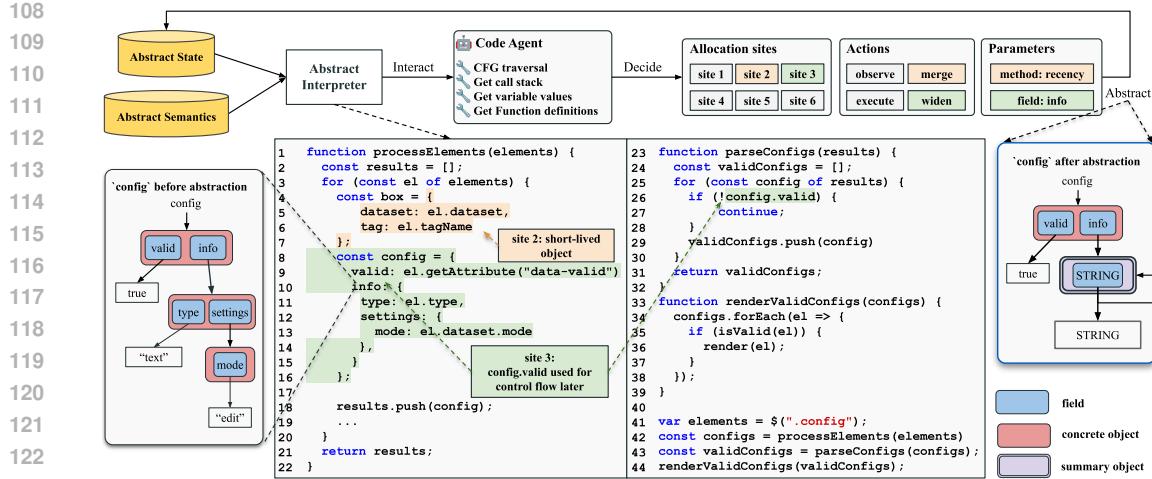


Figure 1: When ABSINT-AI encounters an unbounded loop, it suspends analysis and interacts with the language model agent for abstraction decisions. The agent selects a recency abstraction for `box` and a field-based widening for the `info` field of `config`, preserving relevant structure while ensuring 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.

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 or widening only fields relevant to the analysis domain. 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 adaptive abstractions.

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

171  
 172  
 173 **3.2 ABSTRACT INTERPRETATION**  
 174  
 175

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

180 **Abstract Domain.** Our abstract domain keeps track of heap objects using concrete  
 181 nodes and summary nodes. Summary nodes represent a set of possible concrete nodes.  
 182

183 Each node is a dictionary from primitive or ab-  
 184 stract values to other values. Our domain of  
 185 primitive values is based off of TAJJS (Jensen  
 186 et al., 2009), one of the first abstract interpre-  
 187 tation based analyses for Javascript. The abstract  
 188 domain and transfer functions are fixed; the LM  
 189 agent does not alter the semantics of the anal-  
 190 ysis. Its role is limited to guiding when and  
 191 where widening and merging operators are ap-  
 192 plied. Additional details on our abstract domain  
 193 can be found in the appendix. The most impor-  
 194 tant runtime decision of ABSINT-AI is deciding  
 195 when summarize heap nodes. We keep two sepa-  
 196 rate heap structures, referred to as the local heap  
 197 and global heap.

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

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

211 We draw a distinction between the local and global heap because JavaScript programs tend to be  
 212 reactive, with execution driven largely by external events. This has important implications for analysis,  
 213 as the analysis can't assume the program will simply execute starting at the beginning from a well  
 214 defined initial state. Take the example in Figure 2, where `inc_global` is invoked by an event handler  
 215 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.

```

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.

216 **Agent Invocation.** A key challenge in abstract interpretation is to reach a fixpoint without losing too  
 217 much precision when analyzing potentially unbounded loops. Because fixpoint computation requires  
 218 merging abstract states across iterations, the choice of how to abstract heap objects allocated within  
 219 the loop has a direct impact on both the precision and termination of the analysis.

220 Take the example in Figure 1. There are two objects, `box` and `config`. Each loop iteration allocates  
 221 two objects: `box`, which is short-lived and well-suited to recency abstraction, and `config`, which  
 222 contains a critical field (`valid`) that must remain precise. A uniform abstraction by allocation site  
 223 would collapse these distinctions, introducing spurious behaviors. ABSINT-AI addresses this by  
 224 invoking the LM agent at unbounded loops to choose abstraction strategies per object, balancing  
 225 semantic precision with soundness and convergence. **The agent is only invoked at unbounded loop**  
 226 **joins, not at if–then–else merge points. Conditional branches use standard abstract joins and do not**  
 227 **require agent intervention.**

228

229 **3.3 AGENTIC HEAP ABSTRACTIONS**  
 230

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

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

241 The agent’s outputs are limited to a predefined set of sound abstraction strategies, and  
 242 it never directly manipulates program state or executes code. The underlying ab-  
 243 stract interpreter remains responsible for all semantic computation and fixpoint reasoning.  
 244 This architectural separation allows  
 245 us to embed an adaptive, learning-  
 246 driven agent within a sound static anal-  
 247 ysis framework—enabling high-level  
 248 decision-making informed by context  
 249 and semantics, while preserving for-  
 250 mal correctness guarantees.

251 **Agent Interaction.** The agent is ini-  
 252 tialized with the current abstract state,  
 253 including visible variables, relevant al-  
 254 location site data, and any previously  
 255 encountered heap shapes. It then en-  
 256 ters an interactive decision-making  
 257 loop. During this loop, the agent can  
 258 issue queries to the abstract state for  
 259 more information, such as request-  
 260 ing variable values, inspecting func-  
 261 tion definitions, or examining the heap  
 262 shape. If the available information is  
 263 insufficient, the agent may also post-  
 264 pone its decision making by request-  
 265 ing additional abstract loop iterations,  
 266 allowing it to observe how the heap  
 267 evolves over time. This enables the  
 268 agent to defer commitment while gath-  
 269 ering contextual evidence. **We exper-**  
**imented with providing the full pro-**  
**gram and abstract state directly in the**

---

**Algorithm 1** Agentic Heap Abstraction Algorithm

**Require:** Loop  $\mathcal{L}$ , Analysis state  $\mathcal{S}$ , Allocation Sites  $\mathcal{A}$

- 1:  $b \leftarrow 0$  {Interaction counter (queries + executions)}
- 2:  $\mathcal{A}' = \text{NONE}$
- 3: **while**  $b < \text{budget}$  **do**
- 4:   Agent selects action  $a \in \{\text{INFO}, \text{EXEC}, \text{SELECT}\}$
- 5:   **if**  $a = \text{INFO}$  **then**
- 6:     Agent queries  $\mathcal{S}$  for program information
- 7:      $b \leftarrow b + 1$
- 8:   **else if**  $a = \text{EXEC}$  **then**
- 9:     Abstract Interpreter executes one iteration of the
- 10:     loop
- 11:      $b \leftarrow b + 1$
- 12:   **else if**  $a = \text{SELECT}$  **then**
- 13:     Agent selects sites  $\mathcal{A}' \subseteq \mathcal{A}$  to abstract
- 14:     **break**
- 15:   **end if**
- 16: **end while**
- 17: **if**  $\mathcal{A}' = \text{NONE}$  **then**
- 18:   Agent selects  $\mathcal{A}' \subseteq \mathcal{A}$  to abstract
- 19: **end if**
- 20: **for**  $a_i \in \mathcal{A}'$  **do**
- 21:   Agent selects (Strategy,Parameters)
- 22:   Updated mapping in  $\mathcal{S}$  from  $a_i$  to strategy for  $\mathcal{L}$
- 23: **end for**

---

270 prompt, but the abstract heap often ex-  
 271 ceeded the model’s context window  
 272 for larger or deeply nested programs. To ensure stable, reproducible behavior, the agent instead  
 273 accesses information incrementally through `INFO` queries, retrieving only the specific variable or  
 274 function summaries needed for each decision.

275 The interaction is bounded: the agent operates under a fixed query and iteration budget to ensure  
 276 termination. Once satisfied, the agent returns a set of abstraction directives, specifying how the  
 277 interpreter should merge and widen objects associated with each allocation site. The interpreter then  
 278 executes the loop abstracting the heap as directed by the agent. If the abstract state does not reach  
 279 a fixpoint within five iterations, it re-queries the agent for new abstraction strategies. Algorithm 1  
 280 contains a detailed description of our procedure.

281 The agent performs two decision stages:

283 **1. Selecting which allocation sites to summarize.**

284 • At each loop iteration, the interpreter identifies allocation sites whose abstract states changed.  
 285 • The agent receives a prompt containing:  
 286     – The loop body and relevant code snippet.  
 287     – A summary of changed allocation sites (object structures, points-to sets).  
 288 • The agent can issue a number of **information gathering** requests (for example, querying the  
 289 current abstract value of a variable or requesting a summary of a function’s behavior) or a  
 290 simulated loop execution with the abstract interpreter.  
 291 • Finally, the agent selects a set of allocation sites to summarize or merge, ensuring convergence  
 292 before the next iteration.

294 **2. Choosing a merging strategy and widening strategy for each selected site.**

295 • For every selected allocation site, the interpreter asks the agent to choose one of the predefined  
 296 parameterizable **merging strategies** for that site.  
 297 • The agent picks among them using natural-language cues from code and variable names.  
 298 • After picking a merging strategy, the interpreter asks the agent to choose one of the predefined  
 299 parameterizable **widening strategies**.

300 Every action the agent can take is predefined, finite, and sound—it cannot invent new abstractions,  
 301 only select among existing ones—and all interactions are deterministic within the interpreter.

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

306 • **Variable inspection:** Requesting abstract values of in-scope variables.  
 307 • **Function introspection:** Retrieving the definition of local functions in scope.  
 308 • **Loop execution:** Requesting additional iterations `with the abstract interpreter` to observe how  
 309 heap structures evolve.

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

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

319 • **Allocation-site merge:** Collapses all objects created at the same program location into a single  
 320 abstract object.  
 321 • **Recency merge:** Preserves the most recently allocated object at that site; merges older instances.  
 322 • **Field-sensitive merge:** Groups objects with the same fields.  
 323 • **Role-based merge:** Partitions objects based on semantically meaningful field values (e.g., role),  
 324 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.

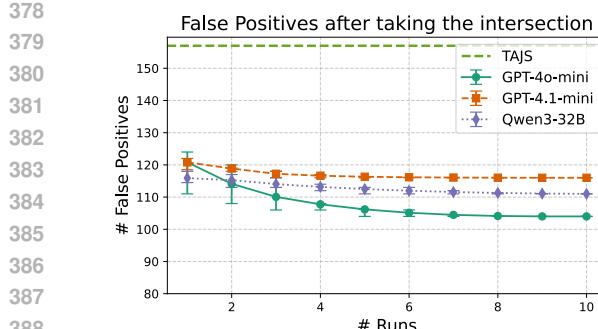
## 4 EVALUATION

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



389  
390 Figure 4: Running multiple times and taking the in-  
391 tersection of the reported bugs allows us to improve  
392 precision while maintaining soundness.

378  
379 Table 1: Overall mean performance across the Dataset.  
380 #FP stands for False Positives. Fewer is better.

Model	# FP $\downarrow$	% Improve
Baselines	TAJS	157 0%
	WALA	312 -98.7%
	Symbolic ABSINT-AI	220 -28.6%
Mean	GPT-4.0-mini	125 20.4%
	GPT-4.1-mini	127 19.1%
	Qwen3-32B	117 25.5%
Intersection	GPT-4.0-mini	104 33.7%
	GPT-4.1-mini	116 26.1%
	Qwen3-32B	111 29.0%
Full Intersection		97 38.2%

393 not based on abstract interpretation and performs flow-insensitive heap analysis, using a combination  
394 of allocation-site abstraction and context-sensitive pointer analysis.

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

403 **Dataset.** To evaluate our approach, we curated a benchmark of 17 self-contained JavaScript programs  
404 from the Big Code dataset Raychev et al. (2016), the V8 benchmark suite, and Github. We filtered  
405 for programs that were self-contained and did not use builtins excessively, as this greatly increases  
406 the imprecision of the analysis (`Math.floor`, for example, requires modeling the `Math` library to  
407 analyze precisely). These require substantial modeling effort and introduce orthogonal complexity.  
408 We also excluded object-oriented programs that rely too heavily on classes and `let` statements, since  
409 TAJS and WALA do not support Javascript features after ES2015. For context, prior work such as  
410 TAJS evaluated on 8 programs (Jensen et al., 2009), underscoring the difficulty of assembling larger  
411 benchmarks for sound JavaScript analysis. **All 17 benchmarks were manually inspected to confirm  
412 that the property of interest (absence of unsafe property accesses) holds.** A detailed description of the  
413 dataset can be found in the Appendix.

## 4.2 PERFORMANCE

414 We evaluate ABSINT-AI using three different language models: GPT-4.0-mini, GPT-4.1-mini, and  
415 Qwen3-32B. To compare against TAJS and WALA, we measure the number of (1) possible accesses  
416 to a property of `null` or `undefined` or (2) possible reads of an absent property of an object. In  
417 this setting, lower values indicate greater precision, reflecting fewer spurious results caused by  
418 imprecise heap abstraction. We run ABSINT-AI 10 times across our benchmark per model across our  
419 17-program benchmark and report the mean results in Table 1.

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

424 **Intersection.** As described in Section 3.4, one benefit of maintaining soundness is that we can safely  
425 take the intersection of reported errors across multiple runs, improving precision without risking  
426 missed bugs. Figure 4 shows the effect of taking intersections across multiple runs. As expected, the  
427 language model often makes different abstraction decisions, leading to partially overlapping sets of  
428 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. Of the 500 seconds required to run across our dataset, 189 seconds correspond to agent interaction—the network latency and inference time of querying the agent—while the remaining time reflects the Python interpreter’s overhead and a more detailed heap representation. When the agent is disabled in Symbolic ABSINT-AI, the analysis yields comparable precision to TAJs/WALA but remains slower. 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 TAJ. 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 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  $3 \times 3$  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 `h`

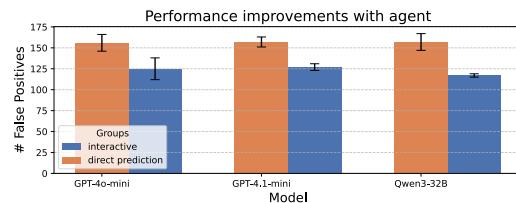


Figure 5: Performance improvements of an interactive agent vs. direct abstraction prediction.

on consistently performs about 25% worse across

```
1 var cell_state = [
2   [0, 1, 0],
3   [0, 1, 0],
4   [0, 1, 0]
5 ]
6 var n = parseInt($("#iterations"));
7 for (var i = 0; i < n; i++) {
8   cell_state = newGeneration(cell_state);
9 }
```

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

486 In contrast, symbolic baselines often over-abstract the structure itself, prematurely merging array  
 487 shapes and losing row-level distinctions. This highlights how the agent draws on both program syntax  
 488 and semantic cues such as common data patterns to guide more precise abstraction decisions.  
 489

## 490 5 RELATED WORK

491 **LMs in program analysis.** LMs have been applied to a wide range of program analysis tasks,  
 492 including type inference, fuzzing, vulnerability and resource leak detection, code summarization,  
 493 and fault localisation (Peng et al., 2023; Wei et al., 2023; Wang et al., 2023b; Xia et al., 2024;  
 494 Yang et al., 2023b;a; Deng et al., 2023; Mathews et al., 2024; Liu et al., 2023; Wang et al., 2023a;  
 495 Mohajer et al., 2023; Cai et al., 2023; Geng et al., 2024; Ahmed et al., 2024; Wang et al., 2022a;  
 496 Wu et al., 2023). However, none have been applied to static analysis while preserving soundness  
 497 guarantees. More recently, several neurosymbolic approaches combine static analysis with LMs:  
 498 LLift (Li et al., 2024a) filters false positives from UBITect (Zhai et al., 2020), IRIS (Li et al., 2024c)  
 499 augments CodeQL (Avgustinov et al., 2016) for taint analysis, and InferROI (Wang et al., 2024)  
 500 detects resource leaks in Java programs. While effective at improving precision, all of these systems  
 501 sacrifice soundness once neural predictions are introduced.  
 502

503 **Program analysis for Javascript.** Much prior work on JavaScript analysis has focused on unsound  
 504 but pragmatic tools for bug finding and security. These tools aim to detect likely vulnerabilities or  
 505 errors in real-world programs, often trading soundness for scalability and precision (Li et al., 2022;  
 506 Fass et al., 2019; Kang et al., 2023; Yu et al., 2023; Guo et al., 2024; Kang et al., 2025). While  
 507 effective for finding particular security issues in practice, these approaches do not provide soundness  
 508 guarantees. As a result, they are not suitable for many downstream tasks that depend on full program  
 509 coverage, such as compiler optimizations or transformations, where missing even a single feasible  
 510 behavior can invalidate correctness. Our work, by contrast, maintains the formal soundness of abstract  
 511 interpretation while improving its precision via adaptive heap abstraction.  
 512

513 **LMs in sound reasoning.** Machine learning has been used to guide compiler optimization selec-  
 514 tion (Ansel et al., 2014; Huang et al., 2019), proof search and theorem proving (Bansal et al., 2019),  
 515 as well as in program synthesis (Li et al., 2024b) and SAT/SMT solving (Ganesh et al., 2022), where  
 516 learned components suggest strategies or rule orderings without affecting overall soundness. In  
 517 contrast, abstract-interpretation-based program analysis forms a distinct line of work, traditionally  
 518 relying solely on manually designed heuristics for abstraction and widening. To our knowledge, no  
 519 prior system has incorporated large language models or other ML components into this framework  
 520 while preserving soundness. Our method is the first to do so by constraining the LLM to select among  
 521 a fixed, verified set of abstraction operators within a sound abstract domain.  
 522

## 523 6 LIMITATIONS AND CONCLUSION

524 **Scalability.** A limitation of ABSINT-AI is that it does not scale to large JavaScript codebases (e.g.,  
 525 2,000+ lines). This is a broader issue with JavaScript static analysis: neither TAJs nor WALA  
 526 converged on such programs in our experiments. The challenge stems from the dynamic and  
 527 object-heavy nature of real-world JavaScript. While our agent-guided approach adds adaptivity, our  
 528 prototype and reliance on whole-program analysis similarly limit scalability. Addressing this is an  
 529 important direction for future work.  
 530

531 In this work, we propose a method to augment static analyzers with an agentic LM for heap abstrac-  
 532 tions. We present ABSINT-AI as a proof-of-concept and an evaluation showing that augmenting  
 533 static analysis with LMs can have a dramatic improvement on the precision without losing soundness  
 534 guarantees.  
 535

## 536 7 REPRODUCIBILITY STATEMENT

537 We have included our source code along with instructions to reproduce the experiments in the  
 538 supplementary material.  
 539

## 540 REFERENCES

541  
542 URL <https://angular.dev/tools/cli/aot-compiler>.543 Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T. Barr. Automatic semantic  
544 augmentation of language model prompts (for code summarization), 2024.545 Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom,  
546 Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program  
547 autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and*  
548 *compilation*, pp. 303–316, 2014.549  
550 Gábor Antal, Péter Hegedűs, Zoltán Herczeg, Gábor Lóki, and Rudolf Ferenc. Is javascript call  
551 graph extraction solved yet? a comparative study of static and dynamic tools. *IEEE Access*, 11:  
552 25266–25284, 2023. doi: 10.1109/ACCESS.2023.3255984.553 Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. Ql: Object-oriented  
554 queries on relational data. In *European Conference on Object-Oriented Programming*, 2016. URL  
555 <https://api.semanticscholar.org/CorpusID:13385963>.556 Kshitij Bansal, Sarah Loos, Markus Rabe, Christian Szegedy, and Stewart Wilcox. Holist: An envi-  
557 ronment for machine learning of higher order logic theorem proving. In *International Conference*  
558 *on Machine Learning*, pp. 454–463. PMLR, 2019.559  
560 brettz9. Brettz9/espree: An esprima-compatible javascript parser. URL <https://github.com/brettz9/espree>.561  
562 Yufan Cai, Yun Lin, Chenyan Liu, Jinglian Wu, Yifan Zhang, Yiming Liu, Yeyun Gong, and Jin Song  
563 Dong. On-the-fly adapting code summarization on trainable cost-effective language models. In  
564 A. Oh, T. Neumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural*  
565 *Information Processing Systems*, volume 36, pp. 56660–56672. Curran Associates, Inc., 2023.566  
567 Satish Chandra, Colin S Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank  
568 Tip, and Youngil Choi. Type inference for static compilation of javascript. *ACM SIGPLAN Notices*,  
569 51(10):410–429, 2016.570 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis  
571 of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM*  
572 *SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 238–252, 1977.573  
574 Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large  
575 language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models,  
2023.576  
577 Aurore Fass, Michael Backes, and Ben Stock. Jstap: A static pre-filter for malicious javascript  
578 detection. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pp.  
579 257–269, 2019.580 Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction  
581 of approximate call graphs for javascript ide services. In *2013 35th International Conference on*  
582 *Software Engineering (ICSE)*, pp. 752–761, 2013. doi: 10.1109/ICSE.2013.6606621.583  
584 Vijay Ganesh, Sanjit A Seshia, and Somesh Jha. Machine learning and logic: a new frontier in  
585 artificial intelligence. *Formal Methods in System Design*, 60(3):426–451, 2022.586  
587 Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao,  
588 and Xiangke Liao. Large language models are few-shot summarizers: Multi-intent comment  
589 generation via in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference*  
590 *on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing  
591 Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3608134. URL <https://doi.org/10.1145/3597503.3608134>.592  
593 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.

594 Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001*  
 595 *ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp.  
 596 54–61, 2001.

597

598 Qijing Huang, Ameer Haj-Ali, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John  
 599 Wawrzynek. Autophase: Compiler phase-ordering for hls with deep reinforcement learning. In  
 600 *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing*  
 601 *Machines (FCCM)*, pp. 308–308. IEEE, 2019.

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

605

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

609 Zifeng Kang, Muxi Lyu, Zhengyu Liu, Jianjia Yu, Runqi Fan, Song Li, and Yinzhi Cao. Follow my  
 610 flow: Unveiling client-side prototype pollution gadgets from one million real-world websites. In  
 611 *2025 IEEE Symposium on Security and Privacy (SP)*, pp. 991–1008. IEEE, 2025.

612 Vini Kanvar and Uday P. Khedker. Heap abstractions for static analysis. *ACM Computing Surveys*, 49  
 613 (2):1–47, June 2016. ISSN 1557-7341. doi: 10.1145/2931098. URL <http://dx.doi.org/10.1145/2931098>.

614

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

617

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

621

622 Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Mining node. js vulnerabilities via object  
 623 dependence graph and query. In *31st USENIX Security Symposium (USENIX Security 22)*, pp.  
 624 143–160, 2022.

625

626 Yixuan Li, Julian Parsert, and Elizabeth Polgreen. Guiding enumerative program synthesis with  
 627 large language models. In *International Conference on Computer Aided Verification*, pp. 280–301.  
 628 Springer, 2024b.

629

630 Ziyang Li, Saikat Dutta, and Mayur Naik. Llm-assisted static analysis for detecting security vulnera-  
 631 bilities, 2024c. URL <https://arxiv.org/abs/2405.17238>.

632

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

635

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

638

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

642

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

645

646 Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy  
 647 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>.

648

649 Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages  
 650 with destructive updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*,  
 651 20(1):1–50, 1998.

648 Joanna CS Santos and Julian Dolby. Program analysis using wala (tutorial). In *Proceedings of the*  
 649 *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of*  
 650 *Software Engineering*, pp. 1819–1819, 2022.

651

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

654

655 Manuel Serrano. On javascript ahead-of-time compilation performance (keynote). In *Proceedings of*  
 656 *the 19th International Conference on Managed Programming Languages and Runtimes*, pp. 1–1,  
 657 2022.

658

659 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking  
 660 for points-to analysis of javascript. In James Noble (ed.), *ECOOP 2012 – Object-Oriented*  
 661 *Programming*, pp. 435–458, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-  
 662 642-31057-7.

663

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

667

668 Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R. Lyu.  
 669 No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In  
 670 *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on*  
 671 *the Foundations of Software Engineering*, ESEC/FSE 2022, pp. 382–394, New York, NY, USA,  
 672 2022a. Association for Computing Machinery. ISBN 9781450394130. doi: 10.1145/3540250.  
 673 3549113. URL <https://doi.org/10.1145/3540250.3549113>.

674

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

677

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

680

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

684

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

687

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

689

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

693

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

696

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

699

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

702

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

705

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

702 Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. An  
 703 extensive study on pre-trained models for program understanding and generation. In *Proceedings*  
 704 *of the 31st ACM SIGSOFT international symposium on software testing and analysis*, pp. 39–51,  
 705 2022.

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

## 714 715 A BACKGROUND

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

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

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

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

756 **B ABSTRACT INTERPRETATION DETAILS**  
757758 **B.1 ANALYSIS DETAILS**  
759760 **Functions** In Javascript, functions are stored as objects on the heap. We include a `__code__` property  
761 storing the function body to be executed. At the beginning of the analysis, ABSINT-AI scans the  
762 entire program, and generates a *schema* for each function. The schema for each function contains  
763 which variables are local to the function and which variables are accessed by other functions. We  
764 refer to variables that are local as *private*, and variables that are accessed by other functions as *shared*.  
765 Each time a function is executed, an environment is initialized according to the schema for that  
766 function. When a function is defined, is initialized with a `__hf__` field set to the current heap frame.  
767 The `__hf__` field is used to model scopes and closures. When the function returns, the stack frame  $\sigma$   
768 is popped from the stack, and the stack pointer is decremented.  
769770 **Scopes and Closures** Whenever a function is called, a new stack frame  $\sigma$  is pushed, along with a  
771 corresponding heap frame. The stack pointer for the current stack frame is updated to point to  $\sigma$ .  
772 The private variables for that function are stored in the stack frame  $\sigma$ , and any shared variables are  
773 stored in the heap frame. The heap frame is initialized with a parent field `__parent__` which is used  
774 to model the scope chain. The `__parent__` field points to the `__hf__` field for the function being  
775 initialized.  
776777 To lookup a variable name in the environment, ABSINT-AI first checks the current stack frame. If it  
778 finds a value for the variable, it returns the value. If it doesn't, it checks the corresponding heap frame  
779 for the stack frame, and then follows the chain of `__parent__` pointers until it finds the variable.  
780781 **Recursion** ABSINT-AI keeps track of all functions that have been called but have not finished  
782 executing yet. Whenever it encounters a recursive call, ABSINT-AI sets the return value to a recursive  
783 placeholder and stores a hash of the function that is called. When the function returns, ABSINT-AI  
784 checks the return values and any allocated heap objects for recursive placeholders for the function  
785 and fills them in with the return values.  
786787 **B.2 ENVIRONMENT**  
788789 In this section we describe how ABSINT-AI represents the abstract state. We define concrete and  
790 abstract values.  $H_L$  refers to the concrete heap,  $H_G$  refers to the global heap, and  $\sigma$  refers to the  
791 stack.  $\tau$  is an abstract type,  $C$  refers to constants,  $obj$  and  $\widetilde{obj}$  refer to concrete and abstract objects.  
792  $val$  and  $\widetilde{val}$  refer to the values that a variable can take.  
793

794 
$$\begin{aligned} val &::= a \mid obj \mid \widetilde{val} \\ \widetilde{val} &::= C \mid \widetilde{a} \mid \tau \mid \widetilde{obj} \\ \tau &::= Bool \mid Null \mid Num \mid String \\ obj &::= \tau \rightarrow val \mid C \rightarrow val \\ \widetilde{obj} &::= \tau \rightarrow \widetilde{val} \mid C \rightarrow \widetilde{val} \\ H_L &::= a \rightarrow val \\ H_G &::= \widetilde{a} \rightarrow \widetilde{val} \\ \sigma &::= C \rightarrow val \end{aligned}$$
  
795

800 **B.3 SYNTAX**  
801

802 
$$\begin{aligned} op &::= + \mid - \mid \div \mid \cdot \mid \dots \\ E &::= id \mid E.field \mid E[E] \mid foo(E) \mid E_1[E_2](E_3, E_4, \dots) \mid \text{function}(x_0, x_1, \dots) \{ S \} \\ &\mid \text{new } foo(E_1, E_2, \dots) \mid C \{ f : E \} \\ varDef &::= \text{var } id = E \mid \text{let } id = E \mid \text{const } id = E \\ Stmt &::= varDef \mid id = E \mid \\ &\quad E.f = E \mid E[E] = E \mid \text{def } foo(x_1, x_2, \dots, x_n) \{ Stmt \} \mid \\ &\quad \text{if } (E) \{ Stmt \} \text{ else } \{ Stmt \} \mid \text{class } foo \{ Stmt \} \mid \\ &\quad \text{return } E \mid \text{for } (\text{varDef}; E; Stmt) \{ Stmt \} \mid \\ &\quad \text{for } (\text{varDef in } E) \{ Stmt \} \mid \text{while } (E) \{ Stmt \} \mid Stmt; Stmt \end{aligned}$$
  
803  
804  
805  
806  
807  
808  
809

810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827

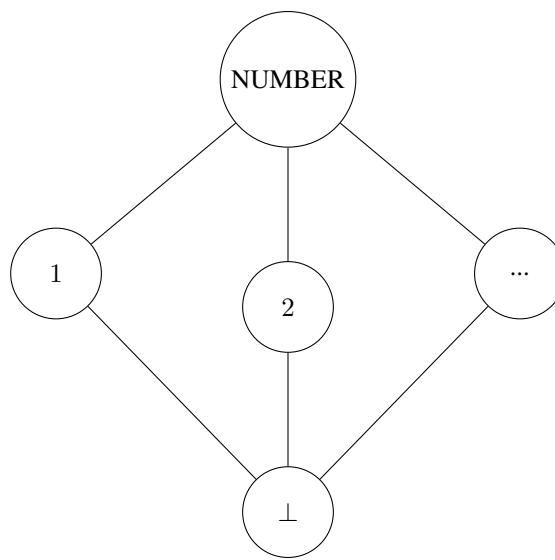


Figure 7: Number Lattice.

828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846

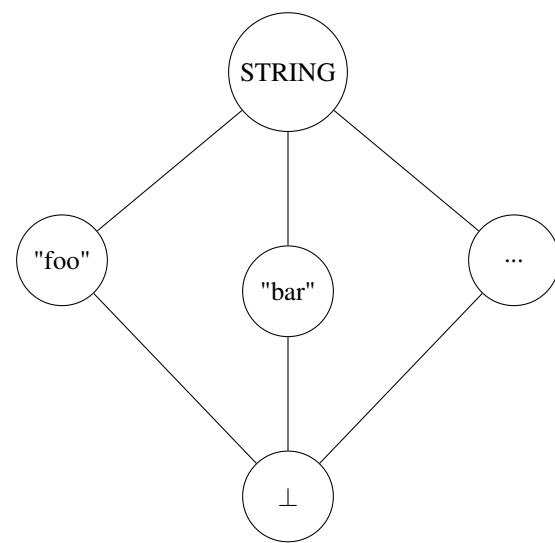


Figure 8: String Lattice.

847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863

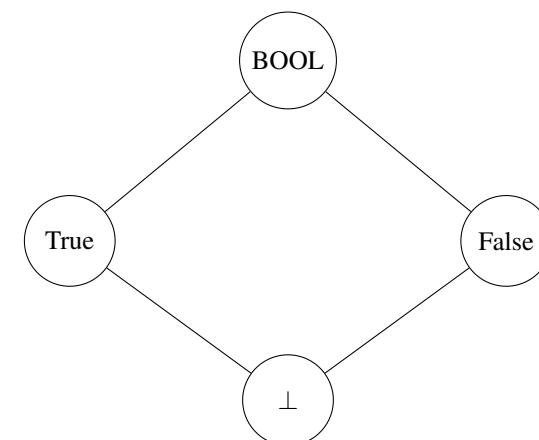


Figure 9: Boolean Lattice.



Figure 10: Null Singleton.

#### 874 B.4 SEMANTICS

##### 875 B.4.1 FUNCTIONS

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

$$\begin{array}{c}
 \text{lookup(id)} \quad \frac{s \equiv \emptyset \quad \theta = \emptyset}{\langle \text{lookup}(H_L, H_G, s, id) \rightarrow \theta \rangle} \\
 \\ 
 \frac{s \in H_L \quad id \in H_L(s) \quad \theta = s}{\langle \text{lookup}(H_L, H_G, s, id) \rightarrow \theta \rangle} \\
 \\ 
 \frac{s \in H_G \quad id \in H_G(s) \quad \theta = s}{\langle \text{lookup}(H_L, H_G, s, id) \rightarrow \theta \rangle} \\
 \\ 
 \frac{s \in H_L \quad id \notin H_L(s) \quad \theta = \text{lookup}(H_L, H_G, H_L(s).par, id)}{\langle \text{lookup}(H_L, H_G, s, id) \rightarrow \theta \rangle} \\
 \\ 
 \frac{s \in H_G \quad id \notin H_G(s) \quad \theta = \text{lookup}(H_L, H_G, H_G(s).par, id)}{\langle \text{lookup}(H_L, H_G, s, id) \rightarrow \theta \rangle} \\
 \\ 
 \text{initialize(schema)} \quad \frac{H_L[a \mapsto \{schema.public, par \mapsto \sigma.hf\}] \quad \sigma'.secret \mapsto \{schema.secret\} \quad \sigma'.hf \mapsto a}{\text{initialize}(schema) \rightarrow H_L, H_G, \sigma :: \sigma'} \\
 \\ 
 \text{return\_from\_schema} \quad \frac{\sigma \equiv \sigma' :: v}{\text{return\_from\_schema} \rightarrow H_L, H_G, \sigma'}
 \end{array}$$

##### 914 B.4.2 SMALL-STEP SEMANTICS

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

918  
919         $\text{id} \quad \frac{}{\langle H_L, H_G, \sigma, \text{id} \rangle \rightarrow \langle H_L, H_G, \sigma, \text{lookup}(\text{id}) \rangle}$   
920  
921         $\text{E.field} \quad \frac{\langle H_L, H_G, \sigma, E \rangle \rightarrow \langle H'_L, H'_G, \sigma', V \rangle}{\langle H_L, H_G, \sigma, \text{E.field} \rangle \rightarrow \langle H'_L, H'_G, \sigma', \text{get}(\text{V.field}) \rangle}$   
922  
923  
924         $E_1[E_2] \quad \frac{\langle H_L, H_G, \sigma, E_2 \rangle \rightarrow \langle H'_L, H'_G, \sigma', V_2 \rangle \quad \langle H'_L, H'_G, \sigma', E_1 \rangle \rightarrow \langle H''_L, H''_G, \sigma'', V_1 \rangle}{\langle H_L, H_G, \sigma, E_1[E_2] \rangle \rightarrow \langle H'_L, H'_G, \sigma', \text{get}(V_1, V_2) \rangle}$   
925  
926  
927         $\text{foo}(E_0, E_1, \dots) \quad \frac{\langle \text{lookup}(\text{foo}) \rightarrow V, V.\_type \equiv \text{Function} \rangle \quad \langle H_L, H_G, \sigma, E_0, E_1, \dots \rangle \rightarrow \langle H'_L, H'_G, \sigma', V_0, V_1, \dots \rangle}{\langle H_L, H_G, \sigma, \text{foo}(E_0, E_1, \dots) \rangle \rightarrow \langle H'_L[x_0 \mapsto V_0, x_1 \mapsto V_1, \dots], H'_G, \sigma', \text{initialize}(V.\_code); V.\_code \rangle}$   
928  
929  
930         $E_1[E_2](E_3, E_4, \dots) \quad \frac{\langle H_L, H_G, \sigma, E_0, E_1, \dots \rangle \rightarrow \langle H'_L, H'_G, \sigma', V_0, V_1, \dots \rangle \quad \langle \text{get}(V_0, V_1) \rightarrow V, V.\_type \equiv \text{Function} \rangle}{\langle H_L, H_G, \sigma, \text{foo}(E_0, E_1, \dots) \rangle \rightarrow \langle H'_L[x_0 \mapsto V_0, x_1 \mapsto V_1, \dots], H'_G, \sigma'[this \mapsto V_0], V.\_code \rangle}$   
931  
932         $\text{function}(x_0, x_1, \dots)\{S\} \quad \frac{}{\langle H_L, H_G, \sigma, \text{function}(x_0, x_1, \dots) \rangle \rightarrow \langle H'_L[a' \mapsto \{\dots, \text{prototype}: a\}, a \mapsto], H'_G, \sigma', a' \rangle}$   
933  
934  
935         $\text{new foo}(E_0, E_1, \dots) \quad \frac{\langle \text{lookup}(\text{foo}) \rightarrow V \rangle \quad \langle V.\_type \equiv \text{Class} \rangle \quad \langle E_0, E_1, \dots \rangle \rightarrow \langle V_0, V_1, \dots \rangle}{\langle H_L, H_G, \sigma, \text{new foo}(E_0, E_1, \dots) \rangle \rightarrow \langle H'_L, H'_G, \sigma'[this \mapsto V], \text{init}(); \text{get}(\text{prototype}(V), \text{constructor})(V_0, V_1, \dots) \rangle}$   
936  
937  
938         $\frac{\langle \text{lookup}(\text{foo}) \rightarrow V \rangle \quad \langle V.\_type \equiv \text{Function} \rangle \quad \langle E_0, E_1, \dots \rangle \rightarrow \langle V_0, V_1, \dots \rangle}{\langle H_L, H_G, \sigma, \text{new foo}(E_0, E_1, \dots) \rangle \rightarrow \langle H'_L, H'_G, \sigma', V.\_code(V_0, V_1, \dots) \rangle}$   
939  
940         $\{f_1 : E_1, f_2 : E_2, \dots\} \quad \frac{\langle H_L, H_G, \sigma, E_1, E_2, \dots \rangle \rightarrow \langle H'_L, H'_G, \sigma', V_1, V_2, \dots \rangle}{\langle H_L, H_G, \sigma, \{f_1 : E_1, f_2 : E_2, \dots\} \rangle \rightarrow \langle H_L[a \mapsto \{f_1 : V_1, f_2 : V_2, \dots, \_type: \text{object}\}], H_G, \sigma, a \rangle}$   
941  
942  
943         $(\text{var } x = E) \quad \frac{\langle H_L, H_G, \sigma, E \rangle \rightarrow \langle H'_L, H'_G, \sigma', V \rangle \quad \theta = \text{lookup}(x) \quad \theta \in H_L \quad \text{fr} = H_L[\theta] \quad \text{fr}' = \text{fr}[id \mapsto V]}{\langle H_L, H_G, \sigma, x = E \rangle \rightarrow \langle H'_L[\theta \mapsto \text{fr}'], H'_G, \sigma', \text{skip} \rangle}$   
944  
945  
946         $\frac{\langle H_L, H_G, \sigma, E \rangle \rightarrow \langle H'_L, H'_G, \sigma', V \rangle \quad \theta = \text{lookup}(x) \quad \theta \in H_G \quad \text{fr} = H_G[\theta] \quad \text{fr}' = \text{fr}[id \mapsto V \cup \text{fr}[id]]}{\langle H_L, H_G, \sigma, x = E \rangle \rightarrow \langle H'_L, H'_G[\theta \mapsto \text{fr}'], \sigma', \text{skip} \rangle}$   
947  
948  
949         $(x.f = E) \quad \frac{\text{lookup}(x) \equiv a \quad \theta = H_L(a) \quad \langle H_L, H_G, \sigma, E \rangle \rightarrow \langle H'_L, H'_G, \sigma', V \rangle}{\langle H_L, H_G, \sigma, x.f = E \rangle \rightarrow \langle H'_L[\theta[f \mapsto V]], H'_G, \sigma', \text{skip} \rangle}$   
950  
951  
952         $\frac{\text{lookup}(x) \equiv \tilde{a} \quad \tilde{\theta} = H_G(\tilde{a}) \quad \langle H_L, H_G, \sigma, E \rangle \rightarrow \langle H'_L, H'_G, \sigma', V \rangle}{\langle H_L, H_G, \sigma, x = E \rangle \rightarrow \langle H'_L, H'_G[\tilde{\theta}[f \mapsto V]], \sigma', \text{skip} \rangle}$   
953  
954  
955  
956  
957  
958  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
970  
971

972  
973  
974  
975  
976  
977  
978  
979  
980      $(x[E] = E') \frac{lookup(x) \equiv a \quad \theta = H_L(a) \quad \langle H_L, H_G, \sigma, E, E' \rangle \rightarrow \langle H'_L, H'_G, \sigma', V, V' \rangle}{\langle H_L, H_G, \sigma, x[f] = E \rangle \rightarrow \langle H'_L[\theta[V \mapsto V']], H'_G, \sigma', skip \rangle}$   
981  
982  
983      $\frac{lookup(x) \equiv \tilde{a} \quad \tilde{\theta} = H_G(\tilde{a}) \quad \langle H_L, H_G, \sigma, E, E' \rangle \rightarrow \langle H'_L, H'_G, \sigma', V, V' \rangle}{\langle H_L, H_G, \sigma, x = E \rangle \rightarrow \langle H'_L, H'_G[\tilde{\theta}[V \mapsto V']], \sigma', skip \rangle}$   
984  
985  
986      $(\text{def foo}(x_0, x_1, \dots, x_n) \{ \text{Stmt} \}) \frac{\theta = lookup(\text{foo}) \quad \theta \in \sigma}{\langle H_L, H_G, \sigma, x[f] = E \rangle \rightarrow \langle H_L[a \mapsto \dots, \text{prototype}: a', a' \mapsto \{\}], H_G, \sigma[\theta \mapsto a], skip \rangle}$   
987  
988  
989      $\frac{\theta = lookup(\text{foo}) \quad \theta \in H_L}{\langle H_L, H_G, \sigma, x[f] = E \rangle \rightarrow \langle H_L[a \mapsto \dots, \text{prototype}: a', a' \mapsto \{\}, \theta \mapsto a], H_G, \sigma, skip \rangle}$   
990  
991  
992      $\frac{\theta = lookup(\text{foo}) \quad \theta \in H_G}{\langle H_L, H_G, \sigma, x[f] = E \rangle \rightarrow \langle H_L, H_G[a \mapsto \dots, \text{prototype}: a', a' \mapsto \{\}, \theta \mapsto \theta \cup a], \sigma, skip \rangle}$   
993  
994  
995      $(x[E] = E') \frac{lookup(x) \equiv a \quad \theta = H_L(a) \quad \langle H_L, H_G, \sigma, E, E' \rangle \rightarrow \langle H'_L, H'_G, \sigma', V, V' \rangle}{\langle H_L, H_G, \sigma, x[f] = E \rangle \rightarrow \langle H'_L[\theta[V \mapsto V']], H'_G, \sigma', skip \rangle}$   
996  
997  
998      $\frac{lookup(x) \equiv \tilde{a} \quad \tilde{\theta} = H_G(\tilde{a}) \quad \langle H_L, H_G, \sigma, E, E' \rangle \rightarrow \langle H'_L, H'_G, \sigma', V, V' \rangle}{\langle H_L, H_G, \sigma, x = E \rangle \rightarrow \langle H'_L, H'_G[\tilde{\theta}[V \mapsto V']], \sigma', skip \rangle}$   
999  
1000  
1001      $\text{if (E) \{ Stmt \} else \{ Stmt' \}} \frac{\langle H_L, H_G, \sigma, E \rangle \rightarrow \langle H'_L, H'_G, \sigma', False \vee \emptyset \rangle}{\langle H_L, H_G, \sigma, \text{if (E) \{ Stmt \} else \{ Stmt' \}} \rangle \rightarrow \langle H'_L, H'_G, \sigma', Stmt \rangle}$   
1002  
1003  
1004      $\frac{\langle H_L, H_G, \sigma, E \rangle \not\rightarrow \langle H'_L, H'_G, \sigma', False \vee \emptyset \rangle}{\langle H_L, H_G, \sigma, \text{if (E) \{ Stmt \} else \{ Stmt' \}} \rangle \rightarrow \langle H'_L, H'_G, \sigma', Stmt' \rangle}$   
1005  
1006  
1007      $\text{class foo}[M_1, M_2, \dots, M_N] \frac{\text{class\_obj} = \{M_1, M_2, \dots, M_N\}}{\langle H_L, H_G, \sigma, \text{class foo}[M_1, M_2, \dots, M_N] \rangle \rightarrow \langle H_L[a \mapsto \text{class\_obj}], H_G, \sigma, skip \rangle}$   
1008  
1009  
1010      $\text{return E} \frac{\langle H_L, H_G, \sigma, E \rangle \rightarrow \langle H'_L, H'_G, \sigma', V \rangle}{\langle H_L, H_G, \sigma, \text{return E} \rangle \rightarrow \langle H'_L, H'_G, \sigma'[returns \mapsto \sigma'[returns] \cup V], skip \rangle}$   
1011  
1012  
1013      $\text{for ([let | var] id in E) \{ Stmt \}} \frac{\langle H_L, H_G, \sigma, E \rangle \rightarrow \langle H'_L, H'_G, \sigma', V \rangle \quad V.\_\text{proto}\_\equiv \emptyset \quad \text{is Empty}(V) \equiv True}{\langle H_L, H_G, \sigma, \text{for ([let | var] id in E) \{ Stmt \}} \rangle \rightarrow \langle H'_L, H'_G, \sigma', skip \rangle}$   
1014  
1015  
1016      $\frac{\langle H_L, H_G, \sigma, E \rangle \rightarrow \langle H'_L, H'_G, \sigma', V \rangle \quad V.\_\text{proto}\_\not\equiv \emptyset \quad \text{is Empty}(V) \equiv True}{\langle H_L, H_G, \sigma, \text{for ([let | var] id in E) \{ Stmt \}} \rangle \rightarrow \langle H'_L, H'_G, \sigma', \text{for ([let | var] id in V.\_\text{proto}\_) \{ Stmt \}} \rangle}$   
1017  
1018  
1019      $\frac{\langle H_L, H_G, \sigma, E \rangle \rightarrow \langle H'_L, H'_G, \sigma', V \rangle \quad V \equiv X :: V' \quad \text{varDef.type} \equiv let}{\langle H_L, H_G, \sigma, \text{for (let id in E) \{ Stmt \}} \rangle \rightarrow \langle H''_L, H''_G, \sigma'', \text{initialize(Stmt); let id=X; Stmt; for (let id in V') \{ Stmt \}} \rangle}$   
1020  
1021  
1022      $\frac{\langle H_L, H_G, \sigma, E \rangle \rightarrow \langle H'_L, H'_G, \sigma', V \rangle \quad V \equiv X :: V' \quad \text{varDef.type} \equiv var}{\langle H_L, H_G, \sigma, \text{for (let id in E) \{ Stmt \}} \rangle \rightarrow \langle H'_L, H'_G, \sigma', \text{var id=X; Stmt; for (let id in V') \{ Stmt \}} \rangle}$   
1023  
1024  
1025

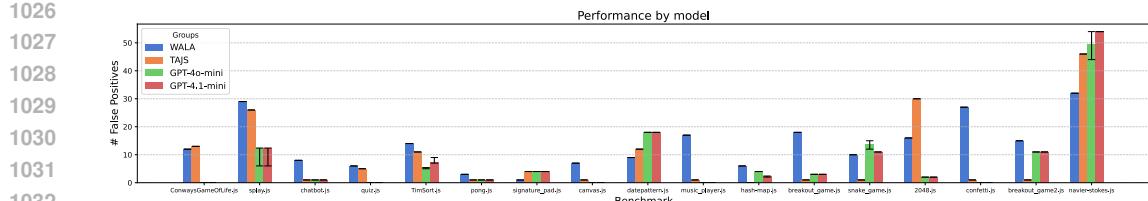


Figure 11: Performance per model on each benchmark program compared to WALA and TAJJS.

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

## C IMPLEMENTATION AND DATASET

**Implementation.** We implemented ABSINT-AI in 8049 lines of Python, and use Esprees brettz 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

1080 experiments, or generate results. All technical contributions, data analysis, and conclusions are the  
1081 authors' own.  
1082  
1083  
1084  
1085  
1086  
1087  
1088  
1089  
1090  
1091  
1092  
1093  
1094  
1095  
1096  
1097  
1098  
1099  
1100  
1101  
1102  
1103  
1104  
1105  
1106  
1107  
1108  
1109  
1110  
1111  
1112  
1113  
1114  
1115  
1116  
1117  
1118  
1119  
1120  
1121  
1122  
1123  
1124  
1125  
1126  
1127  
1128  
1129  
1130  
1131  
1132  
1133