JavaScript Pointer Analysis with Adaptive Heap Abstraction

WENYUAN XU, Aarhus University, Denmark
ANDERS MØLLER, Aarhus University, Denmark

The conventional approach to represent objects in static program analysis is to use allocation-site abstraction. This design choice may lead to redundant computations when many abstract objects are similar. Existing mechanisms that aim to merge such objects are not effective for JavaScript. We propose a novel adaptive heap abstraction technique that during analysis discovers and merges similar abstract objects, thereby reducing the analysis complexity while preserving most of the precision.

The technique has been implemented in a state-of-the-art program analyzer for JavaScript. On a collection of 96 challenging programs, it yields a 2X speedup on average (up to 17X) with a negligible loss of precision. The experimental results also show the effects of various analysis configurations.

CCS Concepts: • **Theory of computation → Program analysis**.

Additional Key Words and Phrases: program analysis, points-to analysis, call graphs

ACM Reference Format:
Wenyuan Xu and Anders Møller. 2026. JavaScript Pointer Analysis with Adaptive Heap Abstraction. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE038 (July 2026), 21 pages. https://doi.org/10.1145/3797133

1 Introduction

Pointer analysis is an essential component in static analysis of programs with pointers or objects [1, 21, 24]. In the JavaScript programming language, which is the focus of this work, functions are a kind of objects and type information is generally unavailable, so pointer analysis is crucial for construction of call graphs and for inferring interprocedural dataflow. Computing such information is valuable for vulnerability detection and many other software development tasks [7, 18, 25, 27]. However, making pointer analysis scalable to large, real-world applications remains challenging, which motivates research in new techniques to improve performance without sacrificing precision.

The common choice of heap abstraction in pointer analysis is *allocation-site abstraction*, which abstracts objects by their syntactic allocation site [4]. Objects that are allocated at the same source code location are grouped together as one abstract object, whereas objects created at different locations are treated as distinct by the analysis. A variant is *field-based* analysis [7] which groups all objects into a single abstract object and only distinguishes heap locations according to field names. That approach may scale well but generally leads to unacceptable precision losses. Context-sensitivity is a general technique for improving precision of pointer analysis [23]. Instead of only abstracting by allocation site, context-sensitive pointer analysis uses more fine-grained abstractions that also distinguish objects by the calling context at the allocations. Although the precision gained by context-sensitivity in some cases leads to performance improvements, adding context-sensitivity generally makes analysis less scalable. Another variant is *type abstraction* [2] which

Authors' Contact Information: Wenyuan Xu, Aarhus University, Aarhus, Denmark, wenyuan.xu@cs.au.dk; Anders Møller, Aarhus University, Aarhus, Denmark, amoeller@cs.au.dk.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/7-ARTFSE038
https://doi.org/10.1145/3797133

refines allocation-site abstraction using object types, but that approach is not directly applicable to dynamically typed languages like JavaScript.

Hardekopf and Lin [9] have demonstrated that allocation-site abstraction can lead to redundant analysis computations when objects from different allocation sites are equivalent in the sense that the objects appear in exactly the same points-to sets, which is called *location equivalence*. They proposed a technique named LE for detecting location equivalence and effectively merging the equivalent abstract objects to improve analysis performance. LE is a pre-analysis designed for C programs and is not applicable to a dynamic language like JavaScript, where indirect calls are pervasive. Other approaches [16, 28] either rely on type information, which is difficult to obtain in JavaScript, or employ a preliminary, less precise pointer analysis to accelerate a subsequent, more precise one, which does not help on our situation where the baseline analysis is already flow-insensitive and context-insensitive. The limitations of existing work are described in detail in Section 5; altogether, none of the related techniques that have been developed for other languages can easily be adapted to speed up state-of-the-art pointer analysis for JavaScript.

In this work, we propose a novel technique inspired by widening [5] to coarsen allocation-site heap abstraction during subset-based pointer analysis for JavaScript. Unlike the LE technique, our approach detects object similarities during analysis rather than using a pre-analysis, which means that it can leverage partially computed points-to information. This is not simply an adaptation of Hardekopf and Lin's notion of location equivalence. Instead, we design a set of heuristics that use information from the program syntax and the points-to sets to identify when and which abstract values can advantageously be represented using a coarser abstraction. Functions are treated differently than ordinary objects to keep the algorithm simple and to prevent precision losses in call graph construction.

We implement the technique on top of a state-of-the-art program analysis framework for JavaScript and evaluate it on a set of real-world cases drawn from GitHub and the npm registry. The results demonstrate that adaptive heap abstraction achieves an average speedup of 2.03× on 96 challenging cases. In the best case, it delivers a 17× improvement, and for the most time-consuming case, it yields nearly a 7× speedup. Moreover, the technique enables successful analysis of 69 additional cases that the baseline fails to complete. Across all benchmarks, the loss in precision remains modest, with an average increase of only 2.28% call edges.

In summary, our contributions are as follows:

- We propose an adaptive heap abstraction technique, presented as an extension of the wave-propagation pointer analysis algorithm [19].
- We design effective heuristics, based on a novel notion of abstract object similarity that combines syntactic and points-to information. Merging is performed for abstract objects that are sufficiently similar and as early as possible to maximize the effect.
- We perform a large-scale empirical evaluation on real-world JavaScript cases, demonstrating the scalability benefits of our approach with minimal loss in precision, and evaluating different design choices.

2 Motivation

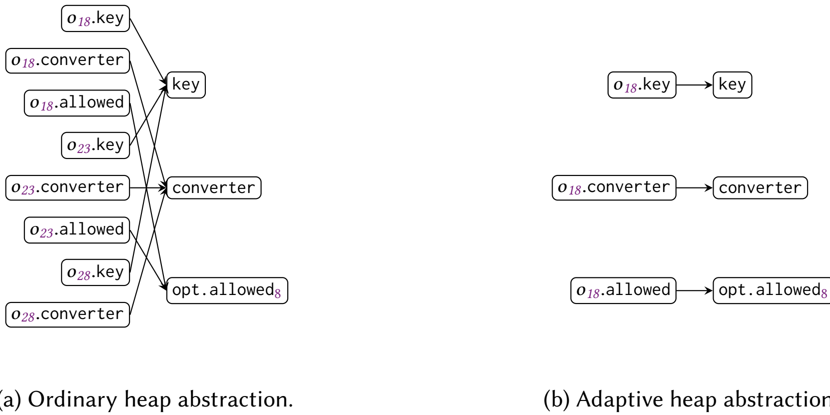
Consider the JavaScript code in Figure 1 from the package *undici*. The `dictionaryConverter` function defined in lines 1–16 returns a function that processes an array of converter objects, such as the one in lines 17–32. In that function, the `key`, `converter`, and `allowed` properties of each of those objects are read in lines 5, 8, and 9. In an ordinary subset-based points-to analysis with allocation-site abstraction (see, e.g., [1, 21, 24]), the corresponding part of the flow graph looks like Figure 2a. Here, nodes represent program variables, expressions or object properties, and edges represent direct

```
1  webidl.dictionaryConverter = function (converters) {
2      return (dictionary) => {
3          let dict = {}
4          for (const opt of converters) {
5              const { key, converter } = opt
6              let value = dictionary[key]
7              value = converter(value)
8              if (opt.allowed
9                      && !opt.allowed.includes(value)) {
10                 throw Exception(/*...*/)
11             }
12             dict[key] = value
13         }
14         return dict;
15     }
16 }
17 webidl.converters.RequestInit = webidl.dictionaryConverter([
18     {
19         key: 'cache',
20         converter: webidl.converters.DOMString,
21         allowed: requestCache
22     },
23     {
24         key: 'referrerPolicy',
25         converter: webidl.converters.DOMString,
26         allowed: referrerPolicy
27     },
28     {
29         key: 'mode',
30         converter: webidl.converters.DOMString
31     }
32 ]);
```

Fig. 1. A code snippet from the package *undici*.

dataflow relations. The abstract objects that model the three objects in the array are denoted o_{18}, o_{23}, and o_{28}, respectively. We observe that in this situation, there is no reason to distinguish between these three abstract objects. Adaptive heap abstraction (presented in Section 3) intuitively collapses them into one abstract object, for example using o_{18} as representative as shown in Figure 2b. This leads to a reduction of the number of flow graph nodes and edges without affecting the concretization of the points-to sets of the program variables and expressions.

In the actual *undici* code, there are not only 3 objects and 3 property read operations but 50 objects and 7 property read operations, and there are many functions like `dictionaryConverter`, so the reduction in analysis complexity is significant. Suppose a base variable (in this case, `opt`) points to l abstract objects, each object has k ancestors (including the object itself) in the prototype chain, m property read operations applied to this variable, and each object property access resolves to n target objects. It then takes time $O(l \times k \times m \times n)$ to compute the points-to information for this part of the program. Merging the abstract objects in the *undici* example corresponds to reducing l.



(a) Ordinary heap abstraction. (b) Adaptive heap abstraction.

Fig. 2. Flow graph before and after abstract objects have been merged using adaptive heap abstraction.

In other situations, merging abstract objects corresponds to reducing n (if target objects are similar) or k (if ancestor objects are similar).

One important question remains: How and when can we decide which abstract objects to merge? If merging only few abstract objects or merging late in the analysis process, it will have little effect. If merging too many or too early, analysis precision may degrade significantly. (Unlike, e.g., the LE approach, we choose to tolerate some precision loss in order to maximize the performance improvements.) Furthermore, we need to be able to decide without incurring a major overhead to the pointer analysis. In the following section, we present an analysis design that reaches these objectives.

3 Approach

In this section, we present the adaptive heap abstraction technique as an extension of a basic pointer analysis for a simplified variant of JavaScript.

3.1 Light-Weight JavaScript

To better illustrate the concept of adaptive heap abstraction, we first introduce a lightweight JavaScript language, with different kinds of statements shown in Table 1 (the right column will be discussed in Section 3.2): object allocations, function definitions, variable assignments, function return, property read and write operations, and function calls. For simplicity, we assume each function takes exactly one parameter, though it is straightforward to extend the model to multiple parameters. Since we focus on flow-insensitive pointer analysis in this paper, control-flow constructs such as loops and conditionals are omitted, so a program is modeled as a set of statements. We also ignore many JavaScript language features including dynamically computed property accesses, prototype-based inheritance, getters and setters, etc.; how to handle those features is not important for understanding how adaptive heap abstraction works. The implementation used for the experimental evaluation (Section 4) supports the full JavaScript language.

In JavaScript, functions are a special kind of objects. A *function token* f_i represents a function defined at program location i, and an *object token* o_i represents non-function objects allocated at location i.[1] A *token* t_i is either a function token or an object token. For a function at location i, we let $param_i$ and ret_i denote its parameter and return variable, respectively.

[1]In practice, allocation sites are represented by source location ranges like $\langle$*file*, *startLine*, *startColumn*, *endLine*, *endColumn*$\rangle$.

Table 1. Light-weight JavaScript and pointer analysis rules.

Type	Statement	Constraint Rule
Object Alloc.	$x = \{\}_i$	$o_i \in pt(x)$
Function Definition	$x = p \Rightarrow_i \{\ldots\}$	$f_i \in pt(x)$
Assignment	$x = y$	$pt(y) \subseteq pt(x)$
Return	$return_i\ x$	$pt(x) \subseteq pt(ret_i)$
Property Read	$x = y.f$	$\dfrac{t_i \in pt(y)}{pt(t_i.f) \subseteq pt(x)}$
Property Write	$x.f = y$	$\dfrac{t_i \in pt(x)}{pt(y) \subseteq pt(t_i.f)}$
Function Call	$x = y(z)$	$\dfrac{f_i \in pt(y)}{pt(z) \subseteq pt(param_i) \quad pt(ret_i) \subseteq pt(x)}$

3.2 Basic Pointer Analysis

The right column of Table 1 presents the declarative rules for classical flow- and context-insensitive Andersen-style pointer analysis [1], where $pt(\cdot)$ denotes the points-to set of a program variable, expression, or abstract object property (collectively called *variables*). Each statement is modeled as a constraint on the points-to sets. Object allocations, function definitions, assignments, and return statements are modeled as simple set constraints. Property reads, property writes, and function calls are modeled by conditional constraints. The analysis result is defined as the least solution to the constraints for the given program. If a call graph is desired, it can be extracted from the points-to sets of the callees at the function calls.

One way to operationalize the analysis constraint rules is using a simplified version of Pereira and Berlin's wave propagation algorithm [19] as shown in Algorithm 1. The wave propagation algorithm takes as input a program represented as a set of statements (as defined in Table 1) and produces the points-to relation pt. Subset relations are represented using a set of directed edges, G. Both pt and G are initially empty.

The main procedure, *Analyze*, begins by invoking *Init*, which processes the simple constraints that model object allocations, function definitions, direct assignments, and return statements. The helper function $S.add(i)$ adds item i into set S (i.e., $S \leftarrow S \cup \{i\}$).

Following initialization, the analysis enters a fixed-point loop (lines 5–10), where it alternates between two sub-procedures: *Propagate* and *AddEdges*. The *Propagate* step ensures that for each subset edge $n \longrightarrow n'$ in G, the points-to set of n is propagated to n'. The *AddEdges* step processes the conditional constraints. For each property read, property write, and function call, if the receiver or callee has new token, the procedure adds the corresponding derived subset edges to G.

The pseudocode omits several important optimizations present in the full wave propagation implementation, such as using worklists to track only the changed variables and subset edges in last iteration, allowing *Propagate* and *AddEdges* to process only the changed parts, as well as topological sorting with cycle elimination. While these optimizations are essential for performance and our approach is compatible with and can benefit from them, the simplified algorithm presented here suffices for explaining the core ideas of adaptive heap abstraction.

Algorithm 1: Simplified wave propagation algorithm

```
   Input   : P, program to analyze
   Output  : pt, points-to relation
1  pt ← ∅                                                  // points-to relation
2  G ← ∅                                                   // flow graph edges
3  Procedure Analyze()
4      Init()
5      repeat
6          pt′ ← pt
7          G′ ← G
8          Propagate()
9          AddEdges()
10     until pt′ = pt and G′ = G
11 Procedure Init()
12     foreach x = {}_i ∈ P do
13         pt(x).add(o_i)
14     foreach x = p ⇒_i {...} ∈ P do
15         pt(x).add(f_i)
16     foreach x = y ∈ P do
17         G.add(y → x)
18     foreach return_i x ∈ P do
19         G.add(x → ret_i)
20 Procedure Propagate()
21     repeat
22         pt′ ← pt
23         foreach u → v ∈ G, t_i ∈ pt(u) do
24             pt(v).add(t_i)
25     until pt′ = pt
26 Procedure AddEdges()
27     foreach x = y.f ∈ P, t_i ∈ pt(y) do
28         G.add(t_i.f → x)
29     foreach x.f = y ∈ P, t_i ∈ pt(x) do
30         G.add(y → t_i.f)
31     foreach x = y(z) ∈ P, f_i ∈ pt(y) do
32         G.add(z → param_i)
33         G.add(ret_i → x)
```

3.3 Adaptive Heap Abstraction

Algorithm 2 presents the basic framework of our approach, which builds on top of Algorithm 1 with a few key modifications (marked in blue).

The general idea is that in each iteration of the main loop, we check for each variable if the size of its points-to set exceeds a predefined threshold, in which case its tokens are grouped and then merged. The rationale is that a large points-to set is an early sign of the pattern observed in the motivating example in Section 2. We refer to this process as *merging* the tokens; it will be discussed in detail in Sections 3.4 and 3.5. When merging tokens, variables that represent properties of the corresponding objects are also merged. For example, when merging the tokens o_{18}, o_{23}, and o_{28}

Algorithm 2: Analysis with adaptive heap abstraction

```
1  A ← ∅                                   // variables whose tokens need to be merged
2  B ← ∅                                   // variables whose tokens have been merged
3  Procedure Analyze()
4      Init()
5      repeat
6          foreach v ∈ A do
7              v′ ← Rep(v)
8              if v′ ∉ B then
9                  Merge(v′)
10                 B.add(v′)
11         A ← ∅
12         pt′ ← pt
13         G′ ← G
14         Propagate()
15         AddEdges()
16     until pt′ = pt and G′ = G
17 Procedure Propagate()
18     repeat
19         pt′ ← pt
20         foreach u → v ∈ G do
21             AddTokens(u, v)
22     until pt′ = pt
23 Procedure AddEdges()
24     foreach x = y.f ∈ P, t_i ∈ pt(y) do
25         t′_i ← Rep(t_i)
26         G.add(t′_i.f → x)
27     foreach x.f = y ∈ P, t_i ∈ pt(x) do
28         t′_i ← Rep(t_i)
29         G.add(y → t′_i.f)
30     foreach x = y(z) ∈ P, f_i ∈ pt(y) do
31         G.add(z → param_i)
32         G.add(ret_i → x)
33 Procedure AddTokens(u, v)
34     foreach o_i ∈ pt(u) do
35         pt(v).add(Rep(o_i))
36     foreach f_i ∈ pt(u) do
37         pt(v).add(f_i)
38     if |pt(v)| ≥ M then
39         A.add(v)
```

from Section 2, we also merge the variables o_{18}.key, o_{23}.key, and o_{28}.key, and similarly for the other object properties.

In the global state, in addition to pt and G we introduce two additional sets: A, which tracks variables whose tokens are candidates for merging, and B, which tracks variables with tokens that have already been merged. $Rep(\cdot)$ is a function (defined later as part of Algorithm 3) that provides

the representative for a merged token or variable. For the motivating example, we chose o_{18} as representative for o_{23} and o_{28}, so $Rep(o_{23}) = Rep(o_{28}) = o_{18}$, and $Rep(o_{23}.\texttt{key}) = Rep(o_{28}.\texttt{key}) = o_{18}.\texttt{key}$.

The modifications to the base algorithm can be grouped into three parts:

(1) At lines 6–10, for each variable v in A, the algorithm first looks up its representative variable $Rep(v)$, denoted as v'. If the tokens of v' have not yet been merged, it performs the merging and then adds v' to B. We will discuss the *Merge* function in detail in Section 3.4.
(2) At line 21 in the propagation phase, instead of directly propagating tokens along edges $u \longrightarrow v$, the algorithm invokes *AddTokens* (lines 33–39). This helper function distinguishes between object tokens o_i and function tokens f_i: for object tokens, their representative $Rep(o_i)$ is propagated; for function tokens, the original token is propagated as-is. The reason behind this distinction will be discussed in detail in Section 3.4. If the size of the points-to set $pt(v)$ exceeds a pre-defined threshold M, the variable v is added to A for future merging.
(3) At lines 25 and 28, for property read and write operations, the representative $Rep(t_i)$ is used instead of the original token t_i when constructing subset edges.

The performance improvements of adaptive heap abstraction comes from multiple factors. Most importantly, part 3 of the modifications contributes to performance improvements by reducing the number of subset edges created for property read and write operations. As previous literature has shown, handling conditional constraints usually dominates the total analysis time [19]. The resulting reduction in the size of the constraint graph G further lowers the cost of subsequent propagation steps. If a worklist-based optimization is used, this also reduces the number of times the conditional constraints are triggered. Part 2 of the modifications also improves performance, by reducing the number of propagated tokens, although this accounts for a smaller fraction of the gain. Conversely, finding the representative tokens and variables via the *Rep* function introduces a small overhead. The experimental evaluation (Section 4) shows that the savings amply outweigh the overhead.

To summarize the high-level idea and partly answer the question from Section 2 about how and when to choose which abstract objects to merge, we emphasize two central design choices: (1) Merging is triggered when a variable's points-to set exceeds a predefined size (line 38). We choose a default threshold $M = 50$ and explore other values in Section 4. (2) Each variable is selected for merging at most once (line 8), in order to prevent redundant calls to *Merge*. We elaborate on this design choice in Section 3.4.

3.4 The Merge Function

Algorithm 3 shows the *Merge* function,[2] which consists of two main phases:

(1) In the preparation phase (lines 3–5), tokens in $pt(v)$ are grouped according to their *signatures*, computed via the function $Sig(t)$. Tokens with the same signature are considered similar for the purpose of merging.
(2) In the main phase (lines 6–13), for each group of tokens S, one of them, $head(S)$, is selected as the representative. All other tokens in the group are redirected to t_0 via the $RedirectToken(t, t_0)$ operation. This redirection updates internal references and merges associated property variables (lines 19–20). For object tokens (denoted *Obj*), non-representative tokens are removed from $pt(v)$ (line 11), and only the representative $Rep(t_0)$ is retained (line 13).

[2] In the pseudocode, the notation [ ] represents the empty map, and $Keys(m)$ and $Values(m)$ denote, respectively, the keys and the values of map m.

Algorithm 3: The Merge function

```
1  π ← []                                              // union-find map
2  Procedure Merge(v)
3      σ ← []                                          // map from signatures to token sets
4      foreach t ∈ pt(v) do
5          σ(Sig(t)).add(t)
6      foreach S ∈ Values(σ) do
7          t_0 ← head(S)
8          foreach t ∈ S do
9              RedirectToken(t, t_0)
10             if t ∈ Obj then
11                 pt(v).remove(t)
12         if t_0 ∈ Obj then
13             pt(v).add(Rep(t_0))
14 Procedure RedirectToken(t_1, t_2)
15     t'_1 ← Rep(t_1)
16     t'_2 ← Rep(t_2)
17     if t'_1 ≠ t'_2 then
18         π(t'_1) ← t'_2
19         forall p ∈ Props(t'_1) do
20             RedirectVar(t'_1.p, t'_2.p)
21 Procedure RedirectVar(v_1, v_2)
22     v'_1 ← Rep(v_1)
23     v'_2 ← Rep(v_2)
24     if v'_1 ≠ v'_2 then
25         π(v'_1) ← v'_2
26         AddTokens(v'_1, v'_2)
27 Procedure Rep(x)
28     if x ∈ Keys(π) then
29         π(x) ← Rep(π(x))
30         return π(x)
31     return x
```

The function $RedirectToken(t_1, t_2)$ performs redirection between two tokens. It first retrieves the representative tokens t'_1 and t'_2 using $Rep(t_1)$ and $Rep(t_2)$, respectively. If the two representatives differ, redirection is performed in two steps:

(1) The redirection relation is recorded in a union-find structure π, by setting $\pi(t'_1) \leftarrow t'_2$, indicating that t'_1 has been redirected to t'_2.
(2) All property variables associated with t'_1 are redirected to their counterparts on t'_2 using $RedirectVar(t'_1.p, t'_2.p)$ for each property $p \in Props(t'_1)$ (lines 19–20). Here, $Props(t)$ is a helper function that retrieves all properties currently associated with token t.

Computing token signatures and grouping tokens by their signatures takes time, so we have to do it selectively. Consider a scenario where merging has already been applied to a variable, causing its points-to set to become small. As the analysis progresses, the points-to set may continue to accumulate tokens from other variables that have also undergone merging. This may cause the points-to set to exceed the threshold again, suggesting that merging should be performed again.

However, in such cases, it is likely that the newly added tokens all have distinct signatures—because they were already merged previously. As a result, significant time would be wasted on computing signatures and performing grouping, without leading to more merging. Thus, the design choice of applying merging to each variable at most once (line 8 in Algorithm 2) may cause some merging opportunities to be missed, but it substantially reduces the overhead.

The function $RedirectVar(v_1, v_2)$ behaves similarly. It first obtains the representative variables v'_1 and v'_2 via $Rep(v_1)$ and $Rep(v_2)$. If they are different, it records the redirection by setting $\pi(v'_1) \leftarrow v'_2$ and propagates the points-to information by adding all tokens from $pt(v'_1)$ into $pt(v'_2)$.

The $Rep(x)$ function implements the *find* operation of a union-find (also called disjoint-set) data structure. It recursively resolves the representative of x by traversing π until reaching a representative that has no parent redirection. Path compression is also applied during this process to flatten the structure for future lookups.

Function tokens are handled differently than object tokens, both in Algorithm 2 and in Algorithm 3. Specifically, function tokens are never replaced by their representatives (but we still merge the variables that represent their object properties, since functions can have properties like any other objects). This may seem as a missed opportunity for the merging technique, however, we opted for this design choice because preliminary experiments revealed two problems if treating function tokens in the same way as object tokens: (1) if one function token were to represent multiple functions, more analysis work would be needed to model the flow of function arguments and return values and to build the call graph; and (2) we observed that merging function tokens is more brittle than merging object tokens, in the sense that merging just a few too many functions using the signature heuristics (Section 3.5) would sometimes lead to significant precision losses. Nevertheless, many variations of these design choices are conceivable, which may be worthwhile to explore in future work.

3.5 Token Signatures

The token signatures, i.e., the $Sig(t)$ function, is defined such that tokens have the same signature if they represent "similar" objects, like the contents of the array in the motivating example in Section 2. The optimal definition would maximize the performance gains of the analysis, but this would be much too expensive to compute. Instead, we choose a pragmatic, heuristic design that sometimes may be too fine-grained (resulting in missed opportunities for performance improvements) or too coarse (resulting in precision losses) but allows the token signatures to be computed quickly and seems to work well in practice. The signature of a token t is a tuple,

$$Sig(t) = \langle kind, module, name, params, prop \rangle$$

where

- *kind* is the kind of the token, which is determined syntactically from the allocation site,[3]
- *module* is the module (i.e., file) in which the token is defined,
- *name* is the declared name of the function if t is a function token and it has a name (otherwise, it is the empty string),
- *params* is the number of parameters in the function declaration if t is a function token (otherwise, the value is set to 0), and
- *prop* is a hash value of $Props(t)$ (i.e., the set of property names being accessed on token t).

For example, o_{18}, o_{23} and o_{28} from Section 2 have the same signature. The definition is inspired by the notion of type-consistency by Tan et al. [28] but is easier to compute. Most components of

[3]The token kinds used in the implementation include `Object`, `Function`, `Array`, and `Class`, plus others that are less important for the merging mechanism.

the signature tuple are trivial to obtain. Only the *prop* component is more expensive, as objects and functions may gain properties dynamically, and hashing string sets is costly. To address this, the implementation caches the computed hash of $Props(t)$ and updates it when more properties are discovered.

4 Evaluation

We implemented our approach on top of Jelly,[4] a state-of-the-art pointer analysis framework for JavaScript and TypeScript that supports modern JavaScript language features, including prototypes, getters/setters, iterators, asynchronous operations, and models of ECMAScript native functions. Adaptive heap abstraction is compatible with all of this. An artifact containing the implementation and all experimental data is available at https://zenodo.org/records/19554781.

Other JavaScript analysis tools, specifically ODGen [14], FAST [10], and Graph.js [8], are not used in the evaluation because they, unlike Jelly, ignore many JavaScript language features including ECMAScript native functions, getter and setters, and iterators, resulting in highly unsound analysis results. Furthermore, Graph.js only analyzes individual files, not whole programs, and ignores all indirect calls. ODGen is known to suffer from severe scalability issues [8, 10]. FAST is an improvement of ODGen but still ignores essential language features and fails to analyze all the benchmarks described below.

The evaluation answers the following research questions:

(1) How does adaptive heap abstraction affect the analysis time for challenging real-world JavaScript programs?
(2) To what extent does the approach affect the precision?
(3) How do the different heuristics for detecting tokens to merge affect the precision and performance of the analysis?
(4) How does the approach perform compared to Hardekopf and Lin's LE technique [9]?

In all analysis runs discussed below, we set a time limit of 1 hour and a memory limit of 30 GB. All experiments are conducted on an Ubuntu server with 2x AMD EPYC 7773X CPU@2.2 GHz.

To obtain a dataset of real-world JavaScript programs that are difficult to analyze, we initially selected the top 2,000 most-starred GitHub repositories written in JavaScript or TypeScript, along with the top 1,000 most downloaded npm packages. Then we excluded any program that took less than 180 seconds to analyze using the baseline Jelly analyzer, as performance improvements are less interesting for such programs that can already be analyzed relatively quickly. Next, we excluded programs that could not be analyzed by the baseline Jelly analyzer with default settings within the selected time and memory limits. This resulted in 96 relevant benchmark programs, 82 from GitHub and 14 from npm. The list includes popular packages, such as, *axios*, *babel-core*, *lodash*, *webpack*, and *react-router*. On average, each of them has over 150 libraries, 1,000 modules, and 15,000 functions. The full list can be found in the artifact.

For the npm packages, we exclude test files and raw source files (such as those ending in `.jsx` or `.ts`). Test files are excluded for two reasons: (1) when analyzing libraries, the test code is not relevant for users, and (2) the test code largely relies on the same testing frameworks. Although our early experiments showed that adaptive heap abstraction significantly speeds up analysis of such frameworks, including them would reduce the overall diversity of our dataset.

To measure the analysis performance, we use the following metrics: running time, the size of the points-to sets (calculated as $\sum_{v \in V} |pt(v)|$), and the size of the subset graph (defined as the number

[4] https://github.com/cs-au-dk/jelly

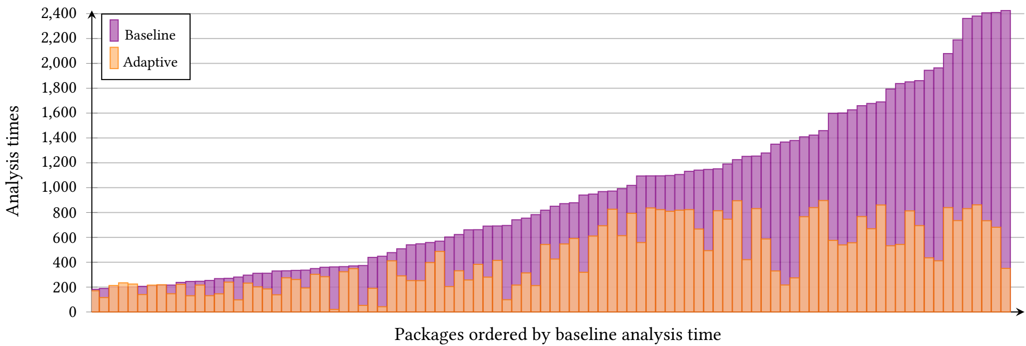


Fig. 3. Analysis times with and without adaptive heap abstraction.

of subset edges). To assess the impact on precision, we use the following metrics as done in prior work on program analysis for JavaScript [12, 13]:

- *Number of call edges*: The total number of call edges in the call graph. Distinct call sites within the same function are treated as separate edges.
- *Number of monomorphic call sites*: A call site is classified as monomorphic if it has at most one associated call edge.
- *Number of polymorphic call sites*: The number of call sites that have multiple call edges.

We do not use dynamic call graphs to measure precision because, to the best of our knowledge, there is no available tool that can generate high-quality call graphs for our setting. Existing dynamic call graph generators lack TypeScript support and only work well when comprehensive test suites are available.

4.1 Result for RQ1 (Performance)

The running times of the baseline analysis and the analysis with adaptive heap abstraction are shown in Figure 3, where each bar represents a benchmark program. As the results indicate, adaptive heap abstraction reduces analysis time in most cases.

For smaller cases, where the baseline analysis completes in approximately 3–5 minutes, adaptive heap abstraction provides limited performance gains. In some cases, it is even slightly slower than the baseline. This is because such cases exhibit relatively little imprecision, resulting in only a small number of variables exceeding the merging threshold. Meanwhile, the mechanism introduces additional overhead to compute signatures, maintain representative mappings, and manage redirection checks. In contrast, larger and more complex cases appear to benefit substantially from adaptive heap abstraction. The threshold is reached more frequently, and when many tokens have the same signature, merging effectively reduces redundant propagation and conditional constraint computation. For example, in the rightmost case in Figure 3, adaptive heap abstraction achieves a performance improvement of approximately 7× compared to the baseline.

Another promising trend observable in Figure 3 is that, for the cases in the right half of the figure—i.e., larger and more complex codebases—the baseline analysis time continues to grow rapidly. In contrast, the analysis time with adaptive heap abstraction shows a slower increase. This suggests that the approach not only improves performance on large cases in our benchmarks but is also likely to scale more gracefully as program size and complexity grow, and we expect the trend to continue beyond the chosen time limit.

A manual inspection of a small sample of programs where adaptive heap abstraction has little effect on analysis speed shows that other techniques would be needed to improve performance in those cases—tweaking the heuristics (in particular, the threshold and the token signature definition) has little effect.

Although the approach requires additional data structures for bookkeeping, the memory overhead of analysis is reduced because tokens and variables are merged. For the 47 benchmarks whose memory consumption was successfully profiled (the remaining benchmarks time out when enabling memory profiling), average peak memory usage per package decreases from 8.84 GB to 4.33 GB. All benchmarks use less memory than the baseline, and 24 use less than 50%.

In general, the performance improvement brought by adaptive heap abstraction is substantial. On average, it achieves a speedup of 2.03× over the baseline across 96 real-world JavaScript programs. Among these, 76 (79.2%) cases see at least a 1.3× speedup, 64 (66.7%) achieve at least 1.5×, and 41 (42.7%) exceed 2.0×. In the best case, adaptive heap abstraction achieves a maximum speedup of 17× compared to the baseline. Moreover, the reduction in memory usage follows a similar trend. The approach enables the analysis to successfully complete on 69 additional programs that the baseline fails to handle within the resource limits. This further demonstrates that the technique improves scalability and makes it possible to analyze larger and more complex JavaScript codebases.

4.2 Result for RQ2 (Precision)

To investigate the impact of adaptive heap abstraction on analysis precision, we examined three key metrics of the computed call graphs: the number of call edges, the number of polymorphic call sites, and the number of monomorphic call sites. Evaluating pointer analysis precision using such call graph metrics follows prior work (e.g., [9, 13, 16, 28]).

From the results, we observe that although the technique introduces some precision loss in certain cases, the overall impact remains limited. Specifically, in 67 (69.8%) of the cases, the call graph size increases by less than 1% compared to the baseline. In 94 (97.9%) of the cases, the number of monomorphic call sites decreases by less than 0.1%. Similarly, 75 (78.1%) of the cases gain less than 1% additional polymorphic call sites. Notably, more than 20 benchmarks exhibit exactly the same number of monomorphic and polymorphic call sites before and after applying adaptive heap abstraction. These results indicate that, despite the merging, the most critical aspects of analysis precision are largely preserved.

This precision stability is primarily due to our merging heuristics. First, the threshold mechanism ensures that merging is only triggered in the presence of imprecision. Second, the special treatment of function tokens avoids mixing callees in the analysis constraints for function calls, thereby minimizing its impact on call graph accuracy. Third, the token signature mechanism guarantees that only tokens with sufficiently similar characteristics are merged.

While the overall impact on precision is low, we identify two notable outliers in our evaluation. The first case *metroui* has a significant increase in the number of call edges (from 491,135 to 587,537) when adaptive heap abstraction is enabled. However, examining the same case across the other two precision metrics reveals only minor differences: The number of monomorphic call sites decreases slightly from 32,266 to 32,107 (a 0.05% decrease), and the number of polymorphic call sites increases modestly from 4,310 to 4,477 (a 3.8% increase). Considering that the technique yields a 1.4× speedup in this case, we consider this as an acceptable trade-off between performance and precision.

In the second case *husky*, adaptive heap abstraction introduces around 1,600 additional polymorphic call sites (from 5,972 to 7,591). These new call sites relate to JavaScript's getters and setters. In Jelly, property accesses are modeled as call sites if the accessed object defines a getter or setter. The increase occurs because of merging of objects that have getters or setters with those that do not have them. This behavior is rare and can be mitigated in future work by refining the property hash

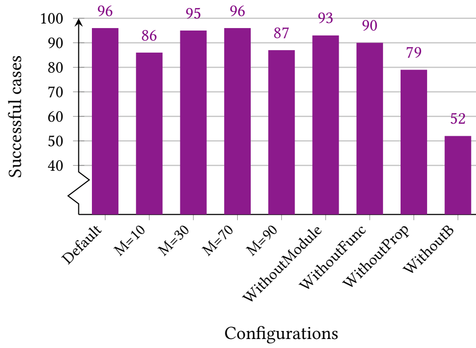


Fig. 4. Different configurations of adaptive heap abstraction.

to incorporate getter/setter information. Meanwhile, since the technique gives a 2.96× speedup, we also consider the precision loss in this case to be acceptable.

In summary, the adaptive heap abstraction mechanism introduces only minor precision losses: the experimental results show on average 2.28% more call edges, 0.05% fewer monomorphic call sites, and 1.33% additional polymorphic call sites.

4.3 Results for RQ3 (Configuration)

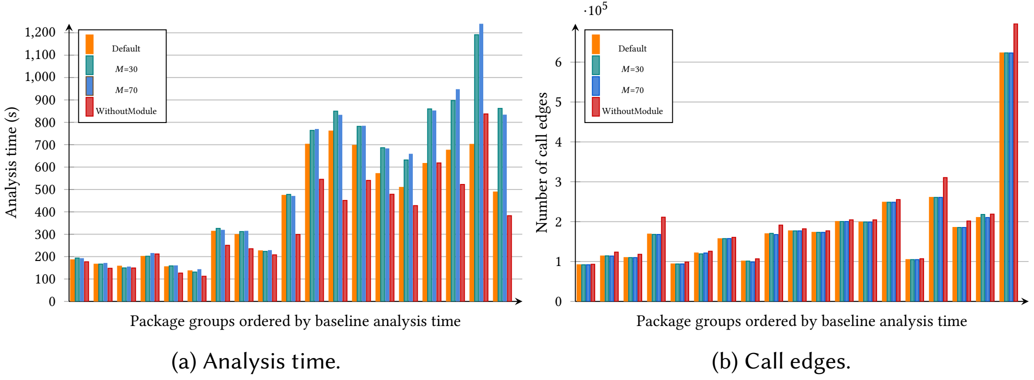
To answer RQ3, we evaluated the impact of different design choices by changing one setting at a time. In the **Default** configuration used in the evaluation for RQ1 and RQ2, the merging threshold M is set to 50 and all the features described in Section 3 were enabled. The following configurations were tested as variations of the default settings:

***M*=10, *M*=30, *M*=70, *M*=90:** set the threshold to 10, 30, 70, or 90, respectively;
WithoutModule: exclude the module information from the token signature;
WithoutFunc: exclude the function name and the number of parameters from the token signature;
WithoutProp: exclude the property hash from the token signature;
WithoutB: allow merging to be applied to the same variable multiple times (i.e., disabling the check $v' \notin B$ in line 8 in Algorithm 2).

Before examining individual metrics such as analysis time and call graph size, we first evaluate whether each configuration can successfully analyze all benchmarks within the resource bounds. The results are shown in Figure 4. We observe that not all configurations are able to successfully analyze all targets. Among them, only the **Default** and ***M*=70** configurations complete all cases, while some of them still merit further investigation like ***M*=30** fails in only one case, and **WithoutModule** fails in three. The ***M*=10** and ***M*=90** configurations cause analysis to fail on, respectively, 10 and 9 of the benchmarks.

Setting the threshold parameter M low causes tokens to be merged too early. Since our signature mechanism relies on information about the abstract objects involved, early merging increases the risk of imprecision. Moreover, since merging is applied to each variable at most once, early merging may occur before the variable has accumulated enough similar tokens, reducing its effectiveness. Conversely, if M is set too high, merging is delayed, such that many tokens may have already triggered many conditional constraints by the time merging is performed, meaning that the analysis gains little or no performance benefit.

Among the configurations for token signature computation (**WithoutModule**, **WithoutFunc**, and **WithoutProp**), we find that the property hash is the most critical component. Disabling it



(a) Analysis time.

(b) Call edges.

Fig. 5. Precision metrics for different configurations.

leads to 17 failed cases, highlighting its importance in effectively grouping similar tokens. Function-related information is also valuable, while module information appears to be less important.

Overall, we see that the information that is extracted from the analysis state during the analysis, i.e., the number of tokens in the points-to sets and the sets of property names that have been discovered for each token, contributes significantly to the analysis performance improvements. The other information that is used for making the decisions about which tokens to merge, i.e., the module names and syntactic information about the functions, plays a minor role but is still useful.

The results for **WithoutB** show the importance of applying merging to each variable only once. As previously discussed, this design avoids repeated computation of token signatures. In addition, it helps prevent over-merging: tokens that represent similar objects are likely to appear together in a variable's points-to set during the same iteration, and merging the first such variable should be sufficient. Allowing merging on the same variable multiple times increases the risk of incorrectly merging later tokens into existing token groups, causing imprecision for other variables that only contain a subset of those tokens.

To further investigate the ***M*=30**, ***M*=70**, and **WithoutModule** configurations, we selected 92 benchmarks that all three configurations successfully analyze. Figure 5 presents the analysis times (Figure 5a) and the number of call edges (Figure 5b) for each configuration. To obtain a concise overview of the comparison results, each bar represents a group of five benchmarks (the benchmarks are ordered by their baseline analysis time and divided into consecutive groups of five, with the last group containing any remaining cases). For each group and metric, we compute the geometric mean for each configuration.

From the results, we observe that among the threshold configurations, the **Default** setting remains the most balanced and generally optimal choice. Whether the threshold is set to 30, 50 or 70 has little effect on the analysis precision, according to the number of call edges. In terms of running time, we observe that across packages with different baseline analysis time, $M = 50$ consistently emerges as a good choice. Choosing $M = 30$ or $M = 70$ causes objects to be merged either too early or too late. This trend holds regardless of the code complexity/size, suggesting that even for larger and more time-consuming packages, using a higher or lower threshold would not yield better performance than $M = 50$.

The **WithoutModule** configuration is particularly interesting: in most groups, it achieves faster analysis than the default setting. However, its drawback is also evident—it produces a less precise

call graph, and in three cases, this imprecision leads to analysis failure due to exceeding the 1 hour timeout or the 30 GB memory limit.

In general, the default configuration offers the best overall trade-off between performance and precision of the configurations studied in this comparison. Disabling the use of module information in the token signature may be a viable alternative in some scenarios, as it can yield additional speedup. However, this comes at the cost of increased imprecision and a risk of analysis failure.

4.4 Results for RQ4 (LE)

As discussed in Section 1, the LE technique by Hardekopf and Lin [9] also aims to improve analysis performance by merging abstract values, so it is relevant to perform an in-depth comparison between that that approach and our proposed adaptive heap abstraction technique. LE was designed, implemented and evaluated for analysis of C programs, but we have made a best-effort adaptation of LE into the Jelly analyzer for JavaScript to enable an experimental comparison.

To implement the LE technique for JavaScript analysis, it is necessary to make several assumptions and adaptations. Most importantly, LE relies on a distinction between direct and indirect variables. Although all call sites in JavaScript are theoretically indirect, we assume that the analyzed program never reassigns a different function to the identifier variable used in a function or class declaration. This allows some call sites to be classified syntactically as direct. Further details of our adaptation are detailed in Appendix A in the artifact. LE operates in two phases: an preparing phase for identifying location equivalence and a main phase for leveraging this equivalence information to perform pointer analysis. We only measure the main phase when comparing analysis time.

The experimental results show that LE does not significantly reduce analysis time, even with the assumption about call sites: There is only 2% speedup on average, and only 5 benchmarks experience a speedup of over 1.5X. LE identifies on average 93 equivalent objects compared to 3,332 similar objects in our approach. There are two main explanations for this. First, the strict definition of location equivalence causes LE to miss opportunities for merging tokens. As an example, consider the code in Figure 1. The abstract objects o_{18}, o_{23}, and o_{28} are not location equivalent because they correspond to different indices of the array object defined in lines 17-32, so they can not be merged by LE. Our approach merges them because they are determined to be sufficiently similar, without negatively impacting the precision of the call graph. As demonstrated in RQ2, this is a common situation. Second, indirect variables are pervasive in JavaScript (unlike in C) due to higher-order function calls, method invocations, and frequent object read/write operations (including getter and setters), which prevents LE from detecting equivalences of their abstract values. Altogether, because LE detects very few location equivalences, we conclude it is ineffective for JavaScript pointer analysis.

4.5 Threats to Validity

The benchmarks are drawn from the most popular projects on GitHub and npm, thereby representing widely used JavaScript codebases that are complex and challenging to analyze. The experiments consistently show that most of the benchmarks benefit from adaptive heap abstraction, which indicates that the technique is broadly applicable and effective in real-world scenarios. Still, it is unknown whether the experimental results generalize to other JavaScript codebases.

When selecting benchmark cases, it is possible that different projects share many common dependencies, which may raise concerns about dataset diversity. However, variations in dependency combinations and usage patterns can lead to substantially different effects on analysis behavior and performance.

Modern hardware and operating systems employ various optimizations, such as memory and disk caching, which can cause subsequent runs of the same to execute slightly faster. To mitigate

this noise, we warm up the environment by running the analysis twice before recording the actual results under different configurations. We see in the experiments that the runtime variance for the same program is within approximately 10 seconds after warm-up, while the speedup achieved through adaptive heap abstraction significantly exceeds this margin. Moreover, internal metrics—such as, the reduction in points-to set sizes and the number of subset edges—further confirm that the observed performance gains are due to the merging itself, rather than external noise.

The approach is implemented on top of the Jelly analysis tool, which is implemented in TypeScript. As a result, both the performance of the JavaScript runtime (Node.js) and the design choice in Jelly can influence the observed effectiveness of adaptive heap abstraction. For example, JavaScript lacks built-in hash functions for objects and does not provide native hash maps, requiring us to implement such mechanisms manually. These operations would likely be faster in compiled languages such as C/C++ or Rust, possibly making adaptive heap abstraction even more efficient in those environments. Moreover, Jelly comprehensively models most JavaScript language features, unlike other more light-weight analysis tools. The experimental results observed for Jelly may not generalize to such tools.

When answering RQ3, we modify only one variable at a time to isolate its effect. While exploring combinations of variables might yield different results, doing so would significantly increase experimental complexity without altering the main conclusions or diminishing the demonstrated benefits of adaptive heap abstraction. Nonetheless, we welcome future work that explores effective combinations of variables or develops new heuristics to further improve the effectiveness.

5 Related Work

When JavaScript programs are executed, arbitrarily many objects may be generated, so static program analyses need to apply abstractions to ensure termination. The choice of abstraction has a crucial effect on the analysis performance and precision. Allocation-site abstraction [4] is a natural and widely used abstraction, used in, for example, the classical subset-based pointer analysis by Andersen [1]. It is based on the idea that objects that are allocated at the same program location are likely to be similar and can hence safely be collapsed, whereas collapsing objects created at different locations is more likely to cause precision losses. This style of pointer analysis is used in popular tools, such as, WALA [30] and Soot [29] (now SootUp [11]).

Steensgaard's unification-based pointer analysis [26] uses a more coarse abstraction, which results in an almost-linear time algorithm but with much lower precision. The adaptive heap abstraction technique presented in this paper is inspired by Steengaard's approach and also uses a union-find data structure to maintain equivalences, but performs unification less aggressively, thereby almost preserving the precision of the baseline analysis.

Other points-to analysis techniques use type-based abstraction [15, 31], which is typically less precise than allocation-site abstraction and only works for statically typed languages, thereby excluding JavaScript. (Even though TypeScript is commonly used, npm packages usually do not contain detailed type information.)

The design of our approach is inspired by widening, as introduced by Cousot and Cousot [5]. Widening is a general technique for accelerating convergence during program analysis. It is often used in connection to numerical domains and has also been applied to heap abstractions (e.g. [3]), but to our knowledge not directly in relation to allocation-site abstraction. In this paper, we instead group and merge tokens to accelerate convergence, similar in spirit to widening.

The observation that plain allocation-site abstraction is sometimes unnecessarily fine-grained has previously led to techniques that resemble the one presented in this paper. Most importantly, Hardekopf and Lin [9] proposed the two methods HVN and LE for detecting pointer equivalence (variables with identical points-to sets) and location equivalence (tokens that always appear together

in any points-to set), respectively. As discussed in Sections 1 and 4, detecting and merging location equivalent tokens can be considered an alternative to our approach, however, the results for RQ4 show that LE is not effective for JavaScript. Our approach merges tokens if they appear together in some points-to set (not necessary in all points-to sets, as in LE), and only when a set is sufficiently large and the tokens have similar syntactic properties. Unlike HVN and LE, merging opportunities are discovered during the points-to analysis itself, leveraging information from the partially computed points-to sets. Moreover, HVN and LE by design preserve precision, whereas we choose to tolerate small precision losses, allowing more aggressive and flexible merging strategies.

Nasre [17] introduced notions of approximate pointer equivalence and location equivalence for identifying and merging points-to sets and pointed-to sets that are similar but not necessarily identical. This technique has been developed for field-insentive pointer analysis for C, and it is unknown how it could be adapted to field-sensitive analysis for JavaScript.

Nagaraj and Govindarajan [16] proposed an approach to accelerate flow-sensitive pointer analysis for C, by first running a flow-insensitive pointer analysis and applying frequent itemset mining to find frequently occurring object sets, and then merging those into single summary objects in flow-sensitive analysis. That approach cannot be used to improve flow-insensitive analysis.

Another related approach is Mahjong by Tan et al. [28], which is designed to speed up allocation-site-based context-sensitive points-to analysis for Java. Based on a pre-analysis, Mahjong merges abstract objects where any sequence of field accesses leads to objects of the same type. The work on Mahjong has inspired our design, particularly in the inclusion of property information when computing token signatures. The pre-analysis in Mahjong is a context-insensitive Andersen-style points-to analysis, making the approach unsuitable for speeding up context-insensitive analysis. Additionally, Mahjong relies on static types, which are not available in JavaScript programs. The lack of static types and the frequent use of indirect calls in JavaScript programs have motivated our choice of not using pre-analysis but designing the adaptive heap abstraction algorithm such that token mergings are decided during the points-to analysis itself.

WALA [30] provides different strategies for heap abstraction. For example, one strategy treats all instances of the same class created within a code block as a single abstract object, as long as the number of instances does not exceed a threshold. This idea resembles our approach of computing signatures using the module information. However, WALA's techniques rely on type information, which is not readily available in JavaScript. In contrast, our approach is designed to identify similar tokens without relying on types.

Many other techniques (e.g., cycle elimination [6], offline variable substitution [20], and set-based preprocessing [22]) aim to improve analysis performance, not by merging tokens as in adaptive heap abstraction, but by merging variables. Such techniques can be applied in combination with adaptive heap abstraction. Also, in context-sensitive points-to analysis [23], allocation-site abstraction is refined by calling contexts. This improves precision compared to context-insensitive analysis but generally makes analysis slower. Our approach has been developed to speed up context-insensitive points-to analysis, but we believe it can also be incorporated into context-sensitive analysis. It may be interesting for future work to evaluate how adaptive heap abstraction performs in combination with such other analysis techniques.

6 Conclusion

We have presented an adaptive heap abstraction technique that improves the performance of pointer analysis of complex JavaScript programs. The key idea is the use of information extracted from the partially computed points-to sets during the analysis and from the program syntax to guide selective merging of tokens that represent functions and objects. By identifying and collapsing tokens that are sufficiently similar, the analysis avoids redundant work, thereby saving time. The

experimental results on 92 challenging programs show that the performance improvements are substantial. Most of the programs benefit, especially those with longer baseline analysis time. The merging heuristics sometimes cause slightly too many tokens to be merged, but the precision losses are overall negligible. A number of design choices have been evaluated individually, showing that, in particular, information about which object properties exist is useful when deciding which tokens to merge.

We believe that the presented technique is not limited to JavaScript or our specific implementation, but can be applied more broadly to points-to analysis. We encourage future work on adapting the approach to other analysis tools and programming languages. Additionally, it may be worthwhile to explore other variations of the heuristics to further improve heap abstractions in pointer analysis.

Acknowledgements

This work was supported by the Danish Research Council for Technology and Production, grant ID 10.46540/3105-00037B.

Data Availability

The source code of the implementation and all experimental data are available at https://zenodo.org/records/19554781.

References

[1] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph. D. Dissertation. University of Copenhagen.

[2] George Balatsouras and Yannis Smaragdakis. 2016. Structure-Sensitive Points-To Analysis for C and C++. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9837)*, Xavier Rival (Ed.). Springer, 84–104. https://doi.org/10.1007/978-3-662-53413-7_5

[3] Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. 2007. Shape Analysis with Structural Invariant Checkers. In *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4634)*, Hanne Riis Nielson and Gilberto Filé (Eds.). Springer, 384–401. https://doi.org/10.1007/978-3-540-74061-2_24

[4] David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. 1990. Analysis of Pointers and Structures. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, Bernard N. Fischer (Ed.). ACM, 296–310. https://doi.org/10.1145/93542.93585

[5] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

[6] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. 1998. Partial Online Cycle Elimination in Inclusion Constraint Graphs. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, Jack W. Davidson, Keith D. Cooper, and A. Michael Berman (Eds.). ACM, 85–96. https://doi.org/10.1145/277650.277667

[7] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 752–761. https://doi.org/10.1109/ICSE.2013.6606621

[8] Mafalda Ferreira, Miguel Monteiro, Tiago Brito, Miguel E. Coimbra, Nuno Santos, Limin Jia, and José Fragoso Santos. 2024. Efficient Static Vulnerability Analysis for JavaScript with Multiversion Dependency Graphs. *Proc. ACM Program. Lang.* 8, PLDI (2024), 417–441. https://doi.org/10.1145/3656394

[9] Ben Hardekopf and Calvin Lin. 2007. Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis. In *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4634)*, Hanne Riis Nielson and Gilberto Filé (Eds.). Springer, 265–280. https://doi.org/10.1007/978-3-540-74061-2_17

[10] Mingqing Kang, Yichao Xu, Song Li, Rigel Gjomemo, Jianwei Hou, V. N. Venkatakrishnan, and Yinzhi Cao. 2023. Scaling JavaScript Abstract Interpretation to Detect and Exploit Node.js Taint-style Vulnerability. In *44th IEEE Symposium*

on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023. IEEE, 1059–1076. https://doi.org/10.1109/SP46215.2023.10179352

[11] Kadiray Karakaya, Stefan Schott, Jonas Klauke, Eric Bodden, Markus Schmidt, Linghui Luo, and Dongjie He. 2024. SootUp: A Redesign of the Soot Static Analysis Framework. In *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 14570)*, Bernd Finkbeiner and Laura Kovács (Eds.). Springer, 229–247. https://doi.org/10.1007/978-3-031-57246-3_13

[12] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 423–434. https://doi.org/10.1145/2491956.2462191

[13] Mathias Rud Laursen, Wenyuan Xu, and Anders Møller. 2024. Reducing Static Analysis Unsoundness with Approximate Interpretation. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1165–1188. https://doi.org/10.1145/3656424

[14] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2022. Mining Node.js Vulnerabilities via Object Dependence Graph and Query. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 143–160. https://www.usenix.org/conference/usenixsecurity22/presentation/li-song

[15] Percy Liang and Mayur Naik. 2011. Scaling abstraction refinement via pruning. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 590–601. https://doi.org/10.1145/1993498.1993567

[16] Vaivaswatha Nagaraj and R. Govindarajan. 2015. Approximating flow-sensitive pointer analysis using frequent itemset mining. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015*, Kunle Olukotun, Aaron Smith, Robert Hundt, and Jason Mars (Eds.). IEEE Computer Society, 225–234. https://doi.org/10.1109/CGO.2015.7054202

[17] Rupesh Nasre. 2011. Approximating inclusion-based points-to analysis. In *Proceedings of the 2011 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with PLDI '11, San Jose, CA, USA, June 5, 2011*, Jeffrey S. Vetter, Madanlal Musuvathi, and Xipeng Shen (Eds.). ACM, 66–73. https://doi.org/10.1145/1988915.1988931

[18] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. 2021. Modular call graph construction for security scanning of Node.js applications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 29–41. https://doi.org/10.1145/3460319.3464836

[19] Fernando Magno Quintão Pereira and Daniel Berlin. 2009. Wave Propagation and Deep Propagation for Pointer Analysis. In *Proceedings of the CGO 2009, The Seventh International Symposium on Code Generation and Optimization, Seattle, Washington, USA, March 22-25, 2009*. IEEE Computer Society, 126–135. https://doi.org/10.1109/CGO.2009.9

[20] Atanas Rountev and Satish Chandra. 2000. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, Monica S. Lam (Ed.). ACM, 47–56. https://doi.org/10.1145/349299.349310

[21] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Foundations and Trends in Programming Languages* 2, 1 (2015), 1–69. https://doi.org/10.1561/2500000014

[22] Yannis Smaragdakis, George Balatsouras, and George Kastrinis. 2013. Set-based pre-processing for points-to analysis. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 253–270. https://doi.org/10.1145/2509136.2509524

[23] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 17–30. https://doi.org/10.1145/1926385.1926390

[24] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. Alias Analysis for Object-Oriented Programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer, 196–232. https://doi.org/10.1007/978-3-642-36946-9_8

[25] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation Tracking for Points-To Analysis of JavaScript. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7313)*, James Noble (Ed.). Springer, 435–458. https://doi.org/10.1007/978-3-642-31057-7_20

[26] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 32–41. https://doi.org/10.1145/237721.237727

[27] Kwangwon Sun and Sukyoung Ryu. 2017. Analysis of JavaScript Programs: Challenges and Research Trends. *ACM Comput. Surv.* 50, 4 (2017), 59:1–59:34. https://doi.org/10.1145/3106741

[28] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 278–291. https://doi.org/10.1145/3062341.3062360

[29] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press.

[30] John Whaley and Monica S. Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *SIGPLAN Not.* 39, 6 (June 2004), 131–144. https://doi.org/10.1145/996893.996859

[31] Christian Wimmer, Codrut Stancu, David Kozak, and Thomas Würthinger. 2024. Scaling Type-Based Points-to Analysis with Saturation. *Proc. ACM Program. Lang.* 8, PLDI (2024), 990–1013. https://doi.org/10.1145/3656417

Received 2025-09-08; accepted 2025-12-22