Reducing Static Analysis Unsoundness with Approximate Interpretation

MATHIAS RUD LAURSEN, Aarhus University, Denmark
WENYUAN XU, Aarhus University, Denmark
ANDERS MØLLER, Aarhus University, Denmark

Static program analysis for JavaScript is more difficult than for many other programming languages. One of the main reasons is the presence of dynamic property accesses that read and write object properties via dynamically computed property names. To ensure scalability and precision, existing state-of-the-art analyses for JavaScript mostly ignore these operations although it results in missed call edges and aliasing relations.

We present a novel dynamic analysis technique named approximate interpretation that is designed to efficiently and fully automatically infer likely determinate facts about dynamic property accesses, in particular those that occur in complex library API initialization code, and how to use the produced information in static analysis to recover much of the abstract information that is otherwise missed.

Our implementation of the technique and experiments on 141 real-world Node.js-based JavaScript applications and libraries show that the approach leads to significant improvements in call graph construction. On average the use of approximate interpretation leads to 55.1% more call edges, 21.8% more reachable functions, 17.7% more resolved call sites, and only 1.5% fewer monomorphic call sites. For 36 JavaScript projects where dynamic call graphs are available, average analysis recall is improved from 75.9% to 88.1% with a negligible reduction in precision.

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

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

ACM Reference Format:
Mathias Rud Laursen, Wenyuan Xu, and Anders Møller. 2024. Reducing Static Analysis Unsoundness with Approximate Interpretation. *Proc. ACM Program. Lang.* 8, PLDI, Article 194 (June 2024), 24 pages. https://doi.org/10.1145/3656424

1 INTRODUCTION

Call graph analysis and points-to analysis are essential techniques in static program analysis that provide the foundation for automated vulnerability detection, optimizations, refactorings, program comprehension, etc. Even though soundness is desirable, most static analyses by design miss some call edges or points-to facts [Livshits et al. 2015; Smaragdakis and Kastrinis 2018]. Despite decades of research, achieving a good balance between *precision* (having only few spurious call edges and points-to relations), *recall* (missing only few actual call edges and points-to relations) and *analysis time* remains difficult. This particularly applies for a dynamic language like JavaScript due to its lack of static types, its dynamic object model, and its complex standard library functions.

A key challenge in static analysis for JavaScript is how to model dynamic property accesses. In JavaScript, objects are essentially dictionaries that map property names (usually strings) to values

Authors' addresses: Mathias Rud Laursen, mrud@cs.au.dk, Aarhus University, Aarhus, Denmark; Wenyuan Xu, wenyuan.xu@cs.au.dk, Aarhus University, Aarhus, Denmark; Anders Møller, amoeller@cs.au.dk, Aarhus University, Aarhus, Denmark.

ACM 2475-1421/2024/6-ART194
https://doi.org/10.1145/3656424

and can be modified freely at runtime. Properties can be accessed either as fixed names, e.g. `x.foo`, or as dynamically computed names, e.g. `x[p]`, where the property name is computed by evaluating the expression `p`. One analysis approach is to over-approximate [Jensen et al. 2009; Kashyap et al. 2014; Lee et al. 2012]. For a dynamic property write operation, `x[p] = y`, where the value of `p` is not statically known, this will conservatively assign the abstract values of `y` to all possible properties of the `x` object. In some situations, partial knowledge of `p` is available, for example, the analysis may know a certain prefix of `p`'s possible values or that `p`'s values are always numeric [Jensen et al. 2009; Madsen and Andreasen 2014]. Nevertheless, experience has shown that over-approximating approaches generally lead to severe losses of precision and poor analysis performance, making such analyzers unscalable to large, real-world JavaScript programs. For this reason, other analyzers instead simply ignore dynamic property read and write operations where the property name is not statically known [Feldthaus et al. 2013; Guarnieri and Livshits 2009; Nielsen et al. 2021]. Such analyzers generally scale well but often suffer from low recall, which may be unacceptable for many use cases, especially in vulnerability detection. Recent work has shown that dynamic property accesses are the main reason for unsoundness in such algorithms for JavaScript [Chakraborty et al. 2022]. Dynamic property accesses and other forms of dynamic object manipulation are prevalent in libraries and frameworks. Even if one mainly cares about analyzing application code, unsound analysis of code in such dependencies often results in highly unsound analysis of the application code. The dynamic language features are often used for initializing APIs, involving copying of functions between object properties.

In this paper we present an effective approach to improve JavaScript analysis accuracy based on a novel hybrid of static and dynamic analysis. Using a scalable but unsound static analysis as a starting point, the approach is designed to reduce the degree of analysis unsoundness focusing specifically on dynamic property read and write operations and related standard library functions, by the use of information collected via a dynamic pre-analysis.

The technique is inspired by a key observation that has been explored in previous work [Andreasen and Møller 2014; Schäfer et al. 2013]: When dynamic object manipulation is used for initializing library APIs, it often happens in execution contexts that are independent of the program's input. Schäfer et al. [2013] introduced the notion of *determinacy*: a variable x is called determinate at a given program location ℓ and calling context c if the value of x is always the same when ℓ is reached in context c. This means that, even though the program code may involve complex dynamic object manipulation, highly polymorphic functions, etc., a single concrete execution that reaches ℓ in context c will reveal the value.

The existing techniques that have tried to take advantage of determinacy have had limited success only. The dynamic determinacy technique by Schäfer et al. [2013] relies on a mechanism called counterfactual execution, which has not been demonstrated to work on large scale for real-world programs, and perhaps more critically, it can only be exploited in static analyses that support both context sensitivity and loop unrolling in a way that is closely aligned with the dynamic analysis. The static determinacy technique by Andreasen and Møller [2014] instead works by selectively applying highly context-sensitive and path-sensitive analysis to the relevant program parts, with such high precision that the static analysis essentially executes determinate program code concretely. This approach has shown to work well on small programs but has still not proven effective on larger scale.

The approach we propose here is a lightweight variant of dynamic determinacy analysis that is based on forced execution [Peng et al. 2014] instead of counterfactual execution. Like dynamic determinacy analysis, we aim to automatically infer determinacy facts, but without insisting on soundness. We refer to this mechanism as *approximate interpretation*.

Given a program to be analyzed, the pre-analysis performs approximate interpretation on the program. This is a fully automatic dynamic analysis that results in a collection of *hints* about the possible values involved, in particular, in the dynamic property accesses. The hints are subsequently consumed by the main analysis, which is a static call graph and points-to analysis that is extended to take advantage of the hints.

Unlike traditional abstract interpretation, approximate interpretation is a form of forced execution and as such may reach program states that are not possible in real executions. This means that the hints it produces may not always be actual determinacy facts. However, if it gets off course, it will only cause a loss of precision of the main analysis, which is often more tolerable than having a low recall. By allowing such approximation we obtain a technique that is simpler and more effective than dynamic determinacy analysis. Most importantly, the hints collected by approximate interpretation contain relational information which makes them usable also in context-insensitive static analysis, unlike the non-relational determinacy facts produced by the algorithm by Schäfer et al. [2013].

In summary, the main contributions of this paper are:

- We propose the notion of *approximate interpretation* as a form of forced execution that is designed to fully automatically learn likely determinate facts (called *hints*) about the behavior of complex object manipulation code in JavaScript programs.
- We demonstrate that the information produced by approximate interpretation can easily be incorporated into traditional static call graph and points-to analysis to reduce the degree of unsoundness.
- We present experimental results based on an implementation of the approximate interpretation algorithm and an extension of a state-of-the-art static call graph analyzer for JavaScript. For 141 real-world JavaScript applications and libraries from GitHub and npm, the approach leads to 55.1% more call edges (from call sites to functions) in the produced call graphs, 21.8% more reachable functions (measuring reachability from the top-level code of the modules of the main package), and 17.7% more resolved call sites (i.e., call sites where at least one callee is found). The additional call edges cause only 1.5% fewer monomorphic call sites (i.e., call sites with at most one callee), which indicates that the analysis precision is not significantly affected. For 36 JavaScript projects where dynamic call graphs are available (produced via the project test suites), analysis recall is improved from 75.9% to 88.1% on average, while precision is reduced only by 1.5%. In one case, recall increases from 40.1% to 98.0% with no reduction in precision. The approximate interpretation phase takes between 0.6 seconds and 51 seconds per program with the current implementation.

2 MOTIVATING EXAMPLE

Figure 1a shows the code for a small web server written with the Express library. After loading the library, an Express web application object is created (line 2), a handler for HTTP GET requests is registered (lines 3–6), and the web server is started (line 7).

Resolving the calls to the `get` and `listen` methods on the application object is difficult because of the way the Express library dynamically initializes its API. The blue edges shown in the figure indicate which functions are being invoked at those two call sites. The function being called by `express()` in Figure 1a (line 2) is the one named `createApplication` in Figure 1b. It returns the web application object, which is in fact a function (lines 14–16), but first twice calls a function `mixin` that performs a form of mixin composition to define properties of the web application object. The `mixin`

```
1  const express = require('express');
2  const app = express();
3  app.get('/', function(req, res) {
4      res.send('Hello world!');
5      server.close();
6  });
7  var server = app.listen(8080);
```

(a) A “Hello world!” Express web server.

```
8  var mixin = require('merge-descriptors');
9  var proto = require('./application');
10 ...
11 exports = module.exports = createApplication;
12 ...
13 function createApplication() {
14   var app = function(req, res, next) {
15     app.handle(req, res, next);
16   };
17   mixin(app, EventEmitter.prototype, false);
18   mixin(app, proto, false);
19   ...
20   return app;
21 }
```

(b) The function in the express module that creates web application objects.

```
22 module.exports = merge;
23 ...
24 function merge(dest, src, redefine) {
25   ...
26   Object.getOwnPropertyNames(src).forEach(function forEachOwnPropertyName(name) {
27     ...
28     var descriptor = Object.getOwnPropertyDescriptor(src, name);
29     Object.defineProperty(dest, name, descriptor);
30   });
31   return dest;
32 }
```

(c) The merge function from the package merge-descriptors (the function is renamed to mixin in Figure 1b).

```
33 var methods = require('methods');
34 ...
35 var app = exports = module.exports = {};
36 ...
37 methods.forEach(function(method) {
38   app[method] = function(path) {
39     ...
40     var route = this._router.route(path);
41     route[method].apply(route, slice.call(arguments, 1));
42     return this;
43   };
44 });
45 ...
46 app.listen = function listen() {
47   var server = http.createServer(this);
48   return server.listen.apply(server, arguments);
49 };
```

(d) Code from the Express module application that initializes methods on application objects (the variable named app in this module refers to object that gets assigned to proto in Figure 1b).

Fig. 1. A JavaScript program where static call graph analysis is challenging due to dynamic language features.

function is implemented in another library, `merge-descriptors`, where the function is named `merge` as shown in Figure 1c. It uses some of JavaScript's other dynamic object manipulation features to copy properties from the `EventEmitter.prototype` and `proto` objects into the web application object.[1]

The variable `proto` in Figure 1b refers to an object defined by the `application` module as shown in Figure 1d. That object is initially empty (line 35) but later gets populated (lines 37–49). Here, `methods` refers to an array of strings of HTTP method names (`"get"`, `"post"`, `"put"`, etc.), which has been built dynamically using some array and string manipulation operations (not shown in the figure). For each of those strings, a method of that name is defined on the object using a dynamic property write operation (line 38). In combination with the mixin mechanism, the call `app.get(...)` in the web server application code thereby ends up invoking the function defined on lines 38–43 in the library code. The `listen` method is defined by a different function (lines 46–49), which is also copied to the web application object by the `mixin` function, so resolving the call `app.listen(...)` similarly involves dynamic object manipulation.

This kind of convoluted code is not uncommon in real-world JavaScript libraries. Existing points-to analysis and call graph construction techniques are to a large extent capable of inferring the flow of objects and functions, but the dynamic object manipulation features cause major challenges. For static analyses that conservatively model dynamic property accesses (e.g., TAJS [Jensen et al. 2009] and SAFE [Lee et al. 2012]), without precise knowledge of the possible string values of the `method` variable, it is impossible to distinguish the various methods that are being defined on the web application object. As mentioned in the introduction, this approach often results in catastrophic losses of analysis precision that render the analysis useless. Conversely, static analyses that unsoundly ignore dynamic property accesses (e.g., WALA [Chakraborty et al. 2022; Feldthaus et al. 2013] and JAM [Nielsen et al. 2021]) miss the call edges from lines 3 and 7 in the application code to the functions in the library code that are invoked when the program is executed.

The approach we propose for analyzing the kind of code shown in the example consists of a combination of a dynamic pre-analysis that infers information about dynamic object manipulation and a static analysis that uses the inferred information to reduce the degree of unsoundness. The key insight is that, although the code is difficult to analyze with traditional static analysis techniques, it is possible to concretely execute code fragments and observe how they use dynamic object accesses and the related constructs. Much of the behavior of the library code is deterministic—specifically, the Express web application objects are initialized in the same way every time `createApplication` is executed—so the observations from a single execution suffice for recovering the missing flow of function values in the static analysis.

The pre-analysis executes the top-level code of each module as well as each function that is discovered in that process using a variant of forced execution, which is explained in detail in Section 3. For this example, by loading and executing the top-level code in the `application` module (Figure 1d), we can observe that an object is created and exported (line 35) and that a function (lines 38–43) is added as methods named `get`, `post`, `put`, etc. on the object. Similarly, by executing the top-level code in the `express` module (Figure 1b), we can observe that the object exported by the `application` module is assigned to the `proto` variable (line 9), and by executing the `createApplication` function (lines 13–21) including the calls to `mixin` we observe that the properties are added to the web application object at the `Object.defineProperty` instruction (line 29). The

[1]The standard library function `Object.getOwnPropertyNames` returns an array of names of properties of the given object, `Object.getOwnPropertyDescriptor` returns a descriptor object for a selected object property, and `Object.defineProperty` defines a property of an object according to a descriptor object.

observations can be collected as a list (here showing only the observations that are relevant for the example):

(1) Object o_1 is created on line 35.
(2) Function f_2 is defined on lines 38–43.
(3) Function f_2 is assigned to the `get` property of object o_1 (on line 38).
(4) Function f_3 is defined on lines 46–49.
(5) Function f_3 is assigned to the `listen` property of object o_1 (on line 46).
(6) Function f_4 is defined on lines 14–16.
(7) Function f_2 is assigned to the `get` property of function f_4 (on line 29).
(8) Function f_3 is assigned to the `listen` property of function f_4 (on line 29).

Loading a module and observing such events is conceptually straightforward, but it is not so easy to execute arbitrary functions. In normal executions, the `createApplication` function in the `express` module is not run until the application code calls it. This particular function has free variables (`mixin` and `proto`), but these can be resolved by first executing the enclosing top-level module code. If `createApplication` had parameters, we would also need to provide sensible values for those. In the simple Express web application shown in Figure 1a it is trivial to reach the call to `createApplication` simply by a single run of the application code, but in real-world Express web applications, it may be more difficult. Instead of trying to come up with actual program inputs that will exercise all the relevant functions, we use a form of forced execution named approximate interpretation. We explain in Section 3 how this works and how the practical challenges can be addressed.

The information from the observations shown above can subsequently be consumed by the static analysis. Assuming that objects (including function objects) are modeled using allocation-site abstraction [Chase et al. 1990], which is a standard approach in static analysis, each observation about a dynamic property access or similar operation (like those from lines 38 and 29) can directly be incorporated into the static analysis. For example, for observation (7), the static analysis already knows the two function objects created on lines 14–16 and 38–43, so we simply add the dataflow fact that the function from lines 38–43 is written to the `get` property of the function object from lines 14–16, which allows the analysis to correctly resolve the call edge from the call `app.get(...)` on line 3 to the function defined on lines 38–43.

The use of approximate interpretation has significant advantages compared to alternative approaches. The most closely related technique is dynamic determinacy analysis [Schäfer et al. 2013], which in principle is able to infer that the values of the `method` variable on line 38 and the `name` variable on line 29 are determinate (meaning that the value is the same whenever execution reaches that point), but only if qualifying the program points according to the specific call contexts and loop iterations. For example, `name` has the value `"get"` on line 29 only when the `merge` function is called from line 18 and only in a specific iteration of the `forEach` loop. Inferring such context and loop specific determinacy facts is difficult, and it requires the static analysis to similarly be context and path sensitive to be able to exploit the inferred information. In contrast, the dynamic analysis in our approach is much simpler, and the inferred information is immediately useful also for context and path insensitive static analysis.

3 APPROXIMATE INTERPRETATION

We present our approximate interpretation technique for a core subset of JavaScript that consists of the language features shown in Figure 2 that involve basic operations on functions and objects. (Our implementation supports the full JavaScript language and works for real programs; see Section 5.) Some of the operations are labeled such that ℓ denotes the location (file, line and column) of the

Operation	Informal meaning
$\{\}_\ell$	Object construction
$x \Rightarrow_\ell E$	Function definition
$E(E')$	Function call
$E.p$	Static property read
$E.p = E'$	Static property write
$E[E']_\ell$	Dynamic property read
$E[E'] = E''$	Dynamic property write

Fig. 2. Operations in the core language and their informal meaning.

operation in the program code. As mentioned in the introduction, object fields (called properties) in JavaScript can be added and overwritten anytime after the objects have been created. In JavaScript, functions are themselves objects and can as such also have properties. Source files act as modules that can be loaded with the `require` function from the standard library.[2] As an example, the `express` module shown in Figure 1b exports the `createApplication` function (line 11), which is loaded into the main application module in Figure 1a (line 1). Each module is implicitly wrapped into a function, which we call a *module function*, that is executed the first time the module is loaded.

To avoid ambiguity, we use the term *function value* to refer to a function that exists as a runtime value, and the term *function definition* refers to a syntactic definition of a function in the source code. The distinction is important because function values may have free variables, so multiple function values can exist for the same function definition. (For modules this distinction is not necessary since module functions do not have free variables.)

Approximate interpretation uses a worklist algorithm to iteratively explore the modules and functions in a given program. In this process the following sets and maps are used:

- *Worklist* is a list of modules and function values that remain to be processed. It is initialized with a collection of JavaScript modules from the program to be analyzed, for example each application-code module or a single designated main module.
- *Visited* is a set of modules and function definitions that have been processed. (Note that it contains function definitions, not function values, as explained later.)
- $\mathcal{H}_R \colon \mathit{Loc} \to \mathcal{P}(\mathit{Loc})$ is a map of *read hints*, which maps source locations to sets of locations. A read hint $\ell' \in \mathcal{H}_R(\ell)$ indicates that an object created at ℓ' has been read at a dynamic property read operation at location ℓ.
- $\mathcal{H}_W \subseteq \mathit{Loc} \times \mathit{String} \times \mathit{Loc}$ is a set of *write hints* that have been collected. Each write hint is a triple (ℓ, p, ℓ') where the labels ℓ and ℓ' denote program locations and p is a property name (i.e., a string). A write hint $(\ell, p, \ell') \in \mathcal{H}_W$ indicates that an object created at location ℓ' has been written to the property p of an object created at location ℓ using a dynamic property write or similar operation.
- $\mathit{loc} \colon \mathit{Object} \to \mathit{Loc}$ is a map from objects (including function values) to the source code locations where they have been created.
- $\mathit{this} \colon \mathit{Object} \to \mathit{Object}$ is a map from function values to objects.

Here, the sets *Loc*, *String*, and *Object* are, respectively, the set of all program locations, the set of all strings (which act as property names), and the set of all runtime objects (including functions and modules). *Visited*, $\mathcal{H}_R$, $\mathcal{H}_W$, *loc*, and *this* are initially empty, except that *loc* contains the names of

[2]The example uses the CommonJS module system; the approach also works for ES modules.

the initial modules in the worklist. The output of approximate interpretation consists of $\mathcal{H}_R$ and $\mathcal{H}_W$; the remaining sets and maps are only used in the process of creating those components.

In each iteration of the worklist algorithm, a module or function value f is selected and removed from *Worklist*, and $loc(f)$ is added to *Visited*. We now wish to execute f to explore what dynamic property accesses and similar operations it may perform. If f is a module, we simply load it (using `require` or `import`, depending on which module system is used). If f is a function, it is not entirely trivial to execute it, because it may have parameters, and because it may access the special variables `arguments` and `this`.[3] Note that free variables are a lesser concern, because function values are closures that contain bindings of the free variables.

We use JavaScript's proxy object mechanism to represent unknown values.[4] A special global proxy object $\mathbf{p}^\star$ is created. The behavior of operations on this object is explained below. The execution of f is then initiated by invoking $f.\texttt{apply}(w, \mathbf{p}^\star)$ where $w = this(f)$ if $this(f)$ is defined and otherwise $w = \mathbf{p}^\star$, which has the effect of binding the value of `this`, `arguments`, and also all the explicitly declared parameters of f in the execution.[5]

During the execution of f (including all the functions called directly or transitively from f), approximate interpretation performs the following actions when different kinds of operations are encountered.

Object constructions At an object construction $\{\}_\ell$ where v is the new object, we simply collect the allocation site by adding ℓ to $loc(v)$.

Function definitions At a function definition $x \texttt{=>}_\ell E$ where v is the new function value, ℓ is added to $loc(v)$ (as done at constructions of ordinary objects). Also, we add v to *Worklist* if $loc(v) \notin Visited$. Importantly, we do not schedule the function value for later execution if a function value from the same function definition has already been executed.

Function calls At a function call $E(E')$ where v is the value of E, the following rules apply.

(1) If $v = \mathbf{p}^\star$, we use the proxy mechanism to treat the call as a no-op and use $\mathbf{p}^\star$ as return value. This naturally ignores any side effects that may happen at the call site in real executions, which we discuss later in this section.

(2) If v is a function in the Node.js standard library,[6] it is treated as in a normal execution, except that all functions that may interact with the outside world (`fs`, `net`, `exec`, etc.) are replaced by a mock function. This provides a simple sandboxing mechanism that avoids possibly harmful effects on the host system running the approximate interpretation. Many of these standard library functions involve callbacks; the mock function simply invokes any callbacks that are given as arguments, and $\mathbf{p}^\star$ is used as return value. Nothing special is done for calls to `require` (and occurrences of `import`); as mentioned earlier the ordinary runtime system will run the module function the first time it is encountered.

(3) If v is a native function in the ECMAScript standard library,[7] it is also treated as in a normal execution, except that certain functions additionally construct new objects and perform dynamic property write operations. Specifically, the function `Object.create` is a form of object construction (as described above), and `Object.defineProperty`, `Object.defineProperties`, and `Object.assign` can be modeled as dynamic property write operations (described below).

(4) If v is any other function, i.e., it is defined in the program being analyzed, it is also executed as usual, except that two extra steps are performed: Before entering the function body, v

[3] In JavaScript, `arguments` is an array-like object that contains all the arguments from the call, and `this` refers to the receiver object at method calls.

[4] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy

[5] In JavaScript, `f.apply(x, a)` invokes the function `f` with `x` as receiver object and arguments from the array `a`.

[6] https://nodejs.org/api/

[7] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects

is added to *Visited* (if not already there) and removed from *Worklist* (if present). To avoid infinite recursion and long running code, execution is aborted if the stack size or the total number of loop iterations reaches a predefined limit.

Static property reads At a static property read $E.p$ where v is the value of E, if $v = \mathbf{p}^\star$ then $\mathbf{p}^\star$ is used as result value via the proxy mechanism to keep execution going, otherwise a normal read operation is performed.

Static property writes At a static property write $E.p = E'$ where v is the value of E and v' is the value of E', if $v = \mathbf{p}^\star$ then the write is simply ignored, but otherwise the operation proceeds as a usual write. In addition, if v' is a function that does not come from the standard library, $this(v')$ is set to v if not already defined. This has the consequence that if v' is later processed by the approximate interpretation mechanism, v will be used as receiver object (i.e., as the value of `this`) as explained earlier. Since v may not be the correct receiver object in all cases, we wrap it into a proxy object that delegates to $\mathbf{p}^\star$ for absent properties.

Dynamic property reads At a dynamic property read $E[E']_\ell$ where v is the resulting value, a read hint is collected by adding $loc(v)$ to $\mathcal{H}_R(\ell)$, and the read then proceeds as usual. In case $loc(v)$ is not defined, which may happen, for example, because v was created via a function call that was ignored, no hint is added. Note that the values of E and E' are ignored in this case, only the result value and the location of the operation are used.

Dynamic property writes At a dynamic property write $E[E'] = E''$ where v, p, and v'' are the values of E, E', and E'', respectively, a write hint is collected by adding $(loc(v), p, loc(v''))$ to $\mathcal{H}_W$, and the actual write is performed as in normal execution. Again, if $loc(v)$ or $loc(v'')$ is undefined, no hint is added. For this kind of operation, its location is ignored.

Continuing the example from Section 2, performing approximate interpretation of the `express` module in Figure 1b, which is the main module of the Express library, the other library modules including `merge-descriptors` (Figure 1c) and `application` (Figure 1d) are also explored because they are loaded via `express`. The function `createApplication` is not executed if only loading the library modules; it requires that a call is simulated by the mechanism described above, or that it is reached via the application code. At the end of the approximate interpretation of the modules and functions from the library code shown in Figures 1b–1d, the following hints have been produced (where ℓ_n is the source location corresponding to line n in the example code):

$$\mathcal{H}_W = \{(\ell_{35}, \texttt{get}, \ell_{38}), (\ell_{35}, \texttt{listen}, \ell_{46}), (\ell_{14}, \texttt{get}, \ell_{38}), (\ell_{14}, \texttt{listen}, \ell_{46}), \dots\}$$

$$\mathcal{H}_R = [\ell_{41} \mapsto \ell_X, \dots]$$

The four write hints come from observations (3), (5), (7) and (8) in Section 2. The single read hint shown here comes from the execution of the function on lines 38–43 (ℓ_X here denotes the location of a function definition that is outside the program fragments shown in the figure).

So far we have not discussed dynamically generated code, which is not uncommon in JavaScript programs [Richards et al. 2011]. With the native function `eval` (and the related function named `Function`), a dynamically generated string can be parsed and executed as JavaScript code. JavaScript code that uses `eval` is notoriously hard to analyze statically [Jensen et al. 2012], but it is pleasantly easily handled by dynamic analysis. When approximate interpretation encounters calls to `eval`, the generated program code is simply executed in the same way as the statically known code, except that we disable the recording of allocation sites at object constructions and function definitions while inside dynamically generated code, since the program locations cannot be mapped to anything meaningful in the static analysis. However, hints may still be generated from such code. At dynamic property writes in dynamically generated code, it is not unusual that both the object being written and the object being written to originate from statically known code, in which case their source

locations are being recorded in *loc* and are therefore available for construction of write hints. A similar situation applies to dynamic read operations. Other opportunities for handling dynamically generated code are discussed in Section 6.

The approach can easily be extended to also improve analysis of code that uses dynamic module loading. At calls to `require` (and at `import` expressions when using the ES module system), the name of the module to be loaded is usually a constant string in real-world JavaScript programs, but it can be a dynamically computed string, which makes it difficult for a static analysis to find out which modules may be loaded. A similar challenge exists in static analysis for Java where techniques like TamiFlex [Bodden et al. 2011] are often used for resolving custom class loading. Instead of relying on existing test suites for the programs under analysis, approximate interpretation can be used for automatically inferring which modules are likely to be loaded. It is straightforward to extend the approximate interpretation algorithm to also collect such hints, and we conjecture that most real-world occurrences of dynamic module loading in JavaScript are determinate, similar to what has been observed for Java.

Approximate interpretation is a simple and fully automatic method for exploring the behavior of complex program code that involves dynamic object manipulation. It is "approximate" for several reasons. It may deviate from real executions of the given program and thereby produce incorrect hints due to the modeling of unknown function arguments. By essentially ignoring method calls and property write operations on the proxy objects, the side effects that happen at those operations in real executions are missed. This is a well-known phenomenon in forced execution [Hu et al. 2017; Kim et al. 2017; Peng et al. 2014]. Another potential cause of incorrect hints is that functions may be executed in contexts that are not realizable in ordinary executions, due to the way approximate interpretation handles free variables and the special variable `this`. The values provided for those variables are essentially guesses that may not always correspond to any real executions. Nevertheless, it is not critical that the produced hints are perfectly correct (unlike the determinacy facts in dynamic determinacy analysis [Schäfer et al. 2013]) since some amount of imprecision occurs anyway in the static analysis that consumes them, as explained in the following section.

The strategy of simply skipping unknown function calls and other operations involving unknown values using the proxy object allows approximate execution to infer hints for fragments of code that might otherwise be difficult to reach. As an example, this mechanism is able to reach the method call on line 41 in the example in Figure 1d even if the function containing that call is nested within another function and is only reached in real executions of the web server application if HTTP requests appear.

A key principle behind the design of the approximate interpretation algorithm is that it aims to infer likely determinate facts of the program being analyzed. For this reason, it is designed to exercise as much program code as possible, but not necessarily visit the same code multiple times. The algorithm has the property that each function definition is selected by the worklist algorithm at most once, which makes it fundamentally different from, e.g., fuzzing and symbolic execution techniques. Many function definitions are skipped because they are reached by approximate interpretation of other functions. Some are unreached because the approximate interpretation algorithm simply fails to reach them for the reasons mentioned above. Still, the exploration strategy manages to discover useful determinacy facts in deep function calls and loops, as demonstrated by the motivating example.

4 REDUCING UNSOUNDNESS OF STATIC CALL GRAPH AND POINTS-TO ANALYSIS

To describe how the hints produced by approximate interpretation can benefit static call graph and points-to analysis, we consider a classic subset-based, flow-insensitive and context-insensitive

Operation	Static analysis constraint rule	
$\{\}_\ell$	$t_\ell \in [\![\{\}_\ell]\!]$	
$x \Rightarrow_\ell E$	$t_\ell \in [\![x \Rightarrow_\ell E]\!]$	
$E(E')$	$\forall t \in [\![E]\!] : [\![E']\!] \subseteq [\![x_t]\!] \wedge [\![E_t]\!] \subseteq [\![E(E')]\!]$	
$E.p$	$\forall t \in [\![E]\!] : [\![t.p]\!] \subseteq [\![E.p]\!]$	
$E.p = E'$	$\forall t \in [\![E]\!] : [\![E']\!] \subseteq [\![t.p]\!]$	
$E[E']_\ell$	$\forall \ell' \in \mathcal{H}_R(\ell) : t_{\ell'} \in [\![E[E']]\!]$	[DPR]
$E[E'] = E''$	$\forall (\ell, p, \ell'') \in \mathcal{H}_W : t_{\ell''} \in [\![t_\ell.p]\!]$	[DPW]

Fig. 3. Static analysis constraint rules for the operations in the core language.

analysis for the core language from Figure 2 (see, e.g., the book chapter by Sridharan et al. [2013]). The analysis rules are presented in constraint-based style in Figure 3. This simple setting suffices for explaining the ideas; the implementation used for experiments in Section 5 supports the full JavaScript language.

The analysis of a given program is expressed as a collection of subset constraints. The object allocation sites[8] and the function definitions constitute a finite set V of abstract values that represent the objects and functions created when the corresponding program object constructions and function definitions are evaluated at runtime. This way of abstracting objects and functions directly matches the use of *loc* in the approximate interpretation. For each program expression E, the constraint variable $[\![E]\!] \subseteq V$ abstractly describes the values E may evaluate to, and similarly, for each abstract object t and property name p, the constraint variable $[\![t.p]\!] \subseteq V$ abstractly describes the values of the properties named p of the objects modeled by t.

The first five rules shown in Figure 3 are standard. The constraint variable for an object construction expression `{}` or a function definition `x => E` at location ℓ contains the corresponding abstract value denoted t_ℓ. The constraint rule for a function call expression $E(E')$ models the flow of arguments and return values. If t is an abstract function value that E may evaluate to, then the arguments flow into its function parameter x_t, and the values that may be returned from the function body E_t flow to the result of the call expression. For a static property read operation $E.p$, if t is an abstract value of E, then the values of the expression include the values of $t.p$. Conversely, for a static property write operation $E.p = E'$ where t is an abstract value of E, the values of $t.p$ include the values of E'. By computing the least solution to the resulting subset constraints, we obtain the points-to sets of all program variables, and we can directly extract a call graph: at every call site $E(E')$, for each abstract function value $t \in [\![E]\!]$ there is a call edge to the function represented by t. With this baseline static analysis, dynamic property reads and writes are ignored, which generally leads to unsoundness, such as missed edges in the call graph. By incorporating the hints from the approximate interpretation phase into the analysis as presented next, the amount of unsoundness is reduced.

The last two rules, named [DPR] and [DPW] in Figure 3, are new. The read hints $\mathcal{H}_R$ are used as follows. Recall from Section 3 that a read hint $\ell' \in \mathcal{H}_R(\ell)$ has been produced during the approximate execution at a dynamic property read operation $E[E']$ at location ℓ if an object

[8]We here assume analysis using allocation-site abstraction [Chase et al. 1990] for objects; the technique also works with field-based analysis [Feldthaus et al. 2013] albeit naturally with lower analysis precision.

originating from location ℓ' has been observed as a result value. For each such hint, we accordingly add the corresponding abstract value $t_{\ell'}$ to the abstract results of the expression, i.e., $t_{\ell'} \in [\![E[E']\!]]$.

For the write hints, $\mathcal{H}_W$, we similarly inject abstract values. A write hint $(\ell, p, \ell'') \in \mathcal{H}_W$ has been produced at a dynamic property write operation $E[E'] = E''$ during the approximate execution if an object originating from ℓ'' has been written to the property p of an object originating from ℓ. Thus, we now simply add the abstract value $t_{\ell''}$ that models the object from ℓ'' to the property p of the abstract value t_ℓ that models the object from ℓ, i.e., $t_{\ell''} \in [\![t_\ell.p]\!]$.

For the example program from Section 2, the five hints produced by approximate interpretation as discussed in Section 3 give rise to these extra constraints in the static analysis (where t_n refers to the abstract value for allocation site n, and E_n refers to the expression E at line n):

$$t_{38} \in [\![t_{35}.\texttt{get}]\!]$$

$$t_{46} \in [\![t_{35}.\texttt{listen}]\!]$$

$$t_{38} \in [\![t_{14}.\texttt{get}]\!]$$

$$t_{46} \in [\![t_{14}.\texttt{listen}]\!]$$

$$t_X \in [\![\texttt{route[method]}_{41}]\!]$$

This enables the static analysis to infer, in particular, the call edge from the call site `app.get(...)` on line 3 to the function at line 38. The baseline analysis infers that $[\![\texttt{app}_2]\!] = \{t_{14}\}$, i.e., that the value of the variable `app` declared on line 2 is the function defined on line 14, so by the [DPW] rule the abstract value t_{38} is added to $[\![t_{14}.\texttt{get}]\!]$, which triggers the generation of the call edge. The other interesting call edge that was missed by the baseline analysis, from the call site `app.listen(...)` on line 7 to the function on line 46, is found similarly now that we have $t_{46} \in [\![t_{14}.\texttt{listen}]\!]$.

Note that the constraint rule for dynamic property writes does not depend on the individual write operations; it does not matter where the write operations have occurred, but only which objects (abstracted by their allocation sites) and property names were involved. Also, at dynamic property read operations the object and property name being read from are not used; instead we use the resulting value and the location of the operation. All this could have been designed differently. For example, we could instead choose to record only the property names and not the origin locations of the involved objects that are encountered during the approximate execution process, and then add subset relations instead of injecting abstract values. For example, at a dynamic property write operation $E[E'] = E''$ where a number of property names $p_1, \ldots, p_n$ are observed as values of E', this would allow us to essentially model the operation as a number of static property write operations, $E.p_1 = E''; \ldots; E.p_n = E''$. Such an alternative approach would, however, cause massive precision losses in many cases, since it would allow abstract dataflow from all the abstract values of E'' to each of the $p_1, \ldots, p_n$ properties of all the abstract values of E. In comparison, the kind of hints and the static analysis rule [DPW] we use for dynamic property write operations have the notable property that they capture more precise relational information between the abstract object being written to, the property name, and the abstract object being written. For example, assume three property writes are observed concretely at a specific dynamic property write operation: $t_1.p_1 = t_1''$; $t_2.p_2 = t_2''$; and $t_3.p_3 = t_3''$. In this situation we have $[\![E]\!] = \{t_1, t_2, t_3\}$, $[\![E']\!] = \{p_1, p_2, p_3\}$, and $[\![E'']\!] = \{t_1'', t_2'', t_3''\}$, so with the alternative approach the analysis would conclude that $[\![t_i.p_j]\!] = \{t_1'', t_2'', t_3''\}$ for all combinations of $i = 1, 2, 3$ and $j = 1, 2, 3$. In contrast, our approach reaches the much more precise conclusion that $[\![t_1.p_1]\!] = \{t_1''\}$, $[\![t_2.p_2]\!] = \{t_2''\}$, and $[\![t_3.p_3]\!] = \{t_3''\}$.

5 EVALUATION

To evaluate the approach we have implemented the approximate interpretation algorithm described in Section 3 and extended an existing static analyzer with the new rules presented in Section 4. The system is designed for Node.js-based JavaScript and TypeScript projects.

Our implementation of approximate interpretation consists of approximately 2,700 lines of TypeScript code. It uses a combination of source-to-source instrumentation (made with Babel [Babel Team 2024]) and manipulation of standard library functions ("monkey-patching") for observing the dynamic operations during execution. Every relevant part of the source code is wrapped with helper functions responsible for recording the results of dynamic operations. The transformed source code is then executed directly in Node.js.

The static analyzer is the open source tool Jelly (https://github.com/cs-au-dk/jelly). At its core, it is a flow-insensitive, context-insensitive points-to analysis with on-the-fly call graph construction implemented in TypeScript. It supports the full JavaScript language (ES2023), including prototype-based inheritance, getters and setters, object/array patterns, generators, iterators, promises, async/await, etc. and includes models of the core ECMAScript standard library functions. TypeScript is supported simply by ignoring type annotations. The analysis tool also contains functionality for producing dynamic call graphs (based on NodeProf [Sun et al. 2018]), which is useful for measuring analysis precision and recall. The extension for implementing the new analysis rules that utilize the hints produced by the pre-analysis consists of less than 200 lines of code.

The evaluation is designed to answer two main research questions:

(1) To what extent does the use of hints produced by approximate interpretation lead to additional call edges being discovered by the static analysis, and how is the analysis accuracy affected?
(2) What is the running time for approximate interpretation per program?

Benchmarks and metrics. To obtain a variety of real-world JavaScript code for the evaluation, we initially collected 100 packages from npm with the highest number of dependents and 122 JavaScript projects from GitHub selected among the top 1,000 highest ranked JavaScript projects which have test suites available and work with the dynamic call graph construction feature of the baseline analyzer. From this initial collection, 13 of the npm packages that only consist of TypeScript type definitions were excluded, and 16 of the npm packages and 52 of the GitHub projects were removed because the baseline static analyzer is unable to finish analysis with 30 GB of memory using the default analysis settings. This leaves us with 141 benchmarks consisting of 71 npm packages and 70 GitHub projects. For each of them, analysis takes less than 10 minutes.

To be able to measure precision and recall of the call graphs computed by the static analysis relative to the dynamically constructed call graphs, we consider a subset of the benchmarks where the dynamic call graphs are of sufficient quality. For 14 projects, the dynamic call graphs contain very few call edges due to low coverage of the test suites or unsupported NodeProf features. For 20 other projects, we encountered incorrect source locations, which prevent a meaningful comparison with the statically computed call graphs. This selection was done through a manual inspection (performed before the actual evaluation of the approximate interpretation technique).

Altogether, this results in a total of 141 JavaScript applications and libraries, including 36 where we also have dynamic call graphs. The programs with dynamic call graphs are listed in Table 1 along with information about their sizes. Note that the static analysis is a whole-program analysis that includes all dependencies of the program being analyzed. Even for relatively small programs, this leads to large amounts of code being analyzed. The median code size for the 141 programs is 1 076 kB (ranging from 2 kB to 6 675 kB).

To measure the impact of the use of hints in the static analysis, we first consider metrics that do not require availability of dynamic call graphs:

Table 1. Node.js benchmarks for which dynamic call graphs are available.

Benchmark	Packages	Modules	Functions	Code Size (kB)
tj/co	5	19	226	36
flitbit/diff	4	32	309	109
leizongmin/js-xss	7	35	253	139
Leonidas-from-XIV/node-xml2js	3	40	603	190
creationix/js-git	5	63	609	327
jaredhanson/passport	11	84	1 984	409
danielstjules/jsinspect	27	76	1 250	511
typestack/typedi	47	119	1 405	604
senchalabs/connect	45	144	984	614
louischatriot/nedb	13	73	2 314	659
expressjs/compression	49	156	1 097	673
expressjs/body-parser	59	185	1 585	898
express-validator/express-validator	3	164	1 717	924
jezen/is-thirteen	41	142	2 062	956
expressjs/cors	83	224	1 731	1 055
hubotio/hubot	98	231	3 014	1 185
posthtml/posthtml	45	187	2 181	1 350
krakenjs/kraken-js	127	296	3 191	1 380
visionmedia/supertest	99	267	2 644	1 390
ternjs/tern	22	233	3 316	1 434
thlorenz/doctoc	96	322	3 212	1 479
gulpjs/gulp	324	603	3 855	1 501
mroderick/PubSubJS	20	145	1 683	1 608
node-schedule/node-schedule	76	247	2 469	1 621
expressjs/multer	141	247	2 348	1 726
nodeca/js-yaml	58	478	4 431	1 852
expressjs/express	117	422	5 395	1 866
digitalbazaar/forge	67	220	3 150	2 189
IonicaBizau/scrape-it	198	390	4 337	2 243
acornjs/acorn	16	87	1 285	2 287
nexe/nexe	215	524	4 946	2 318
zemirco/json2csv	112	428	6 743	2 628
nodeca/pako	96	382	6 449	2 638
share/sharedb	37	226	5 628	3 088
MikeMcl/big.js	1	26	1 718	3 228
twbs/bootlint	110	741	10 022	6 675

- *Number of call edges* in the produced call graph. Call edges that originate from the different call sites within the same function are counted as distinct edges.
- *Number of reachable functions*. Reachability is measured from the module functions of the main package. This metric is particularly useful in vulnerability analyses that rely on call graph reachability [Nielsen et al. 2021].
- *Percentage of resolved call sites*. A call site is considered resolved if the analysis found at least one call edge for it. Some call sites are unresolved because they involve callbacks in unused library code, but for most call sites a good call graph analysis should be able to find one or more call edges, so this metric gives an indication of the analysis recall [Feldthaus et al. 2013].
- *Percentage of monomorphic call sites*. A call site is considered monomorphic if the analysis has found at most one call edge for it, and otherwise it is considered polymorphic. Some call edges are polymorphic due to use of dynamic dispatching or higher-order functions, but most call sites in practice are monomorphic, so this metric gives an indication of the analysis precision [Kastrinis and Smaragdakis 2013].

For the benchmarks where dynamic call graphs are available, we additionally consider these metrics:

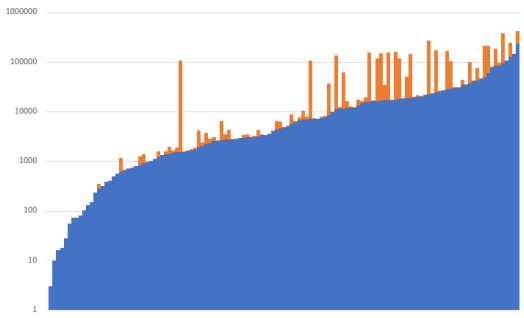

Fig. 4. Call edges.

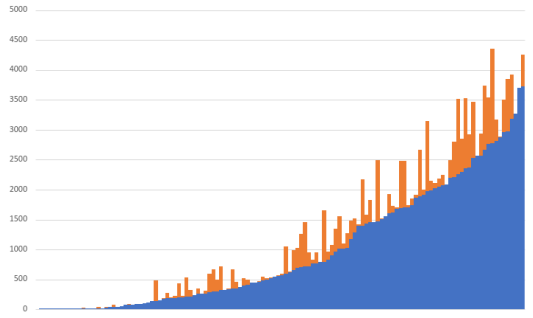

Fig. 5. Reachable functions.

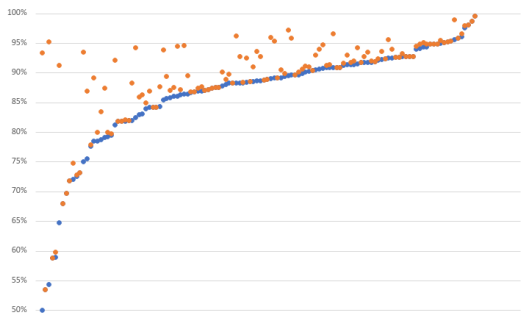

Fig. 6. Resolved call sites.

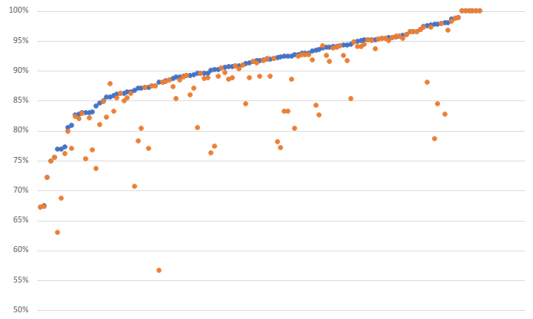

Fig. 7. Monomorphic call sites.

- *Call edge set recall*, i.e., the percentage of call edges in the dynamic call graph that are also in the static call graph [Chakraborty et al. 2022]. For a sound analysis, this would be 100%.
- *Per-call precision*, i.e., the average percentage of the number of call edges in the static call graph that are also in the dynamic call graph for each call site that appears in the dynamic call graph [Feldthaus et al. 2013]. Since the static analysis also produces call edges for code that is not reached in the dynamic executions, this is a more meaningful way of measuring precision than by comparing the sets of call edges for the entire program.

The experiments have been conducted on an Intel(R) Xeon(R) Platinum 8375C CPU @ 2.90GHz machine with memory limit set to 30 GB RAM and analysis time limit set to 10 minutes for each of the two phases per program.

Results. For each of the 141 benchmark programs we have run the approximate interpretation tool and the static analysis with and without the new analysis rules that incorporate the hints. In summary, the results are as follows for the first four metrics described above: The number of call edges for each program is shown in Figure 4. The blue bars indicate the numbers with the baseline static analysis in logarithmic scale; the orange bars indicate the additional call edges added by the new mechanism. The programs are sorted by the former numbers. The number of reachable functions for each program is similarly shown in Figure 5. On average, we see that the use of approximate interpretation leads to 55.1% new call edges being discovered and 21.8% more functions deemed reachable.

The percentage of resolved call sites per program is shown in Figure 6, where the programs are sorted by the percentage for the baseline analysis indicated by the blue dots. The orange dots show that more call sites are resolved with the new mechanism, especially for many of the programs

Table 2. Analysis recall and precision for the Node.js benchmarks for which dynamic call graphs are available.

Benchmark	Recall (%)			Precision (%)		
tj/co	93.1	100.0	(+6.9)	91.2	91.2	(0.0)
flitbit/diff	65.1	100.0	(+34.9)	67.0	61.9	(-5.1)
leizongmin/js-xss	97.3	100.0	(+2.7)	97.2	97.9	(+0.7)
Leonidas-from-XIV/node-xml2js	89.6	89.9	(+0.3)	95.0	95.0	(0.0)
creationix/js-git	95.0	99.1	(+4.1)	96.2	95.7	(-0.5)
jaredhanson/passport	89.0	99.4	(+10.4)	69.9	70.0	(+0.1)
danielstjules/jsinspect	66.3	99.7	(+33.4)	84.5	80.2	(-4.3)
typestack/typedi	70.6	70.6	(0.0)	99.7	99.7	(0.0)
senchalabs/connect	40.1	98.0	(+57.9)	88.2	88.5	(+0.3)
louischatriot/nedb*	88.0	95.3	(+7.3)	78.3	78.2	(-0.1)
expressjs/compression	65.3	78.0	(+12.7)	97.1	92.4	(-4.7)
expressjs/body-parser	51.0	97.0	(+46.0)	92.6	80.3	(-12.3)
express-validator/express-validator	59.1	63.9	(+4.8)	91.4	86.4	(-5.0)
jezen/is-thirteen	100.0	100.0	(0.0)	48.3	48.3	(0.0)
expressjs/cors	66.8	76.5	(+9.7)	91.1	90.1	(-1.0)
hubotio/hubot	58.6	73.8	(+15.2)	94.6	94.6	(0.0)
posthtml/posthtml	90.3	90.3	(+0.0)	79.3	79.3	(+0.0)
krakenjs/kraken-js	37.1	41.6	(+4.5)	96.1	95.9	(-0.2)
visionmedia/supertest	44.6	87.4	(+42.8)	82.2	77.3	(-4.9)
ternjs/tern	66.2	75.2	(+9.0)	91.9	88.3	(-3.6)
thlorenz/doctoc	69.3	95.8	(+26.5)	91.1	90.6	(-0.5)
gulpjs/gulp	83.7	87.4	(+3.7)	81.9	81.3	(-0.6)
mroderick/PubSubJS	88.4	98.5	(+10.1)	88.7	89.2	(+0.5)
node-schedule/node-schedule	96.8	99.5	(+2.7)	97.2	97.2	(0.0)
expressjs/multer	95.6	95.6	(0.0)	94.6	94.6	(0.0)
nodeca/js-yaml	87.9	95.3	(+7.4)	95.5	95.3	(−0.2)
expressjs/express	71.9	99.3	(+27.4)	88.3	86.5	(−1.8)
digitalbazaar/forge	96.3	99.6	(+3.3)	93.9	93.7	(-0.2)
IonicaBizau/scrape-it	80.5	86.9	(+6.4)	94.8	91.3	(-3.5)
acornjs/acorn	99.7	99.9	(+0.2)	100.0	100.0	(0.0)
nexe/nexe	91.6	98.2	(+6.6)	92.4	90.2	(-2.2)
zemirco/json2csv	83.5	83.8	(+0.3)	97.2	97.2	(0.0)
nodeca/pako	100.0	100.0	(0.0)	99.6	99.6	(0.0)
share/sharedb	61.9	63.5	(+1.6)	76.3	75.5	(-0.8)
MikeMcl/big.js	100.0	100.0	(0.0)	96.0	96.0	(0.0)
twbs/bootlint	78.8	80.2	(+1.4)	94.7	93.4	(-1.3)

where the percentage was low with the baseline analysis. The static call graphs ignores method calls on primitive values (e.g., `"foo".toUpperCase()`), so such calls to standard library functions count as unresolved in this metric, which partly explains the remaining gap to 100%. The percentage of monomorphic call sites is similarly shown in Figure 7. As expected since more call edges are discovered, fewer call sites are deemed monomorphic, but on average the decrease is only 1.5%. The outliers where the percentage increases are due to a larger number of getter/setter call sites being found.

The static analysis time naturally increases due to the extra dataflow that needs to be propagated. In 9 cases, the extra dataflow causes the static analysis to fail due to out-of-memory; the results shown are for the remaining programs. For 76 of the 141 programs, static analysis time increases by less than 10%, and for 20 programs it increases by more than 2X. The number of hints being produced ranges from 0 to 15 036 with median 1 492.

In these experiments, the use of analysis hints obtained by approximate interpretation leads to, on average, 55.1% more call edges, 21.8% more reachable functions, 17.7% more resolved call sites, and only 1.5% fewer monomorphic call sites.

Table 2 shows the extra recall and precision results for the benchmarks where dynamic call graphs are available. The 'Recall' column shows the call edge set recall before and after the new technique, and 'Precision' similarly shows the per-call precision. For one benchmark (marked with $*$ in the table), the numbers are shown for analysis with rule [DPR] disabled (i.e., only using rule [DPW]) since it otherwise resulted in out-of-memory.

For some of the benchmarks, the technique yields considerable improvements in the analysis recall, in one case from 40.1% to 98.0%. Unsurprisingly, not all JavaScript programs benefit from the technique. For some, the baseline analysis already produces high-quality call graphs, and for some programs the call graphs miss call edges even after applying the new technique for reasons unrelated to dynamic object manipulation, for example, insufficient models of native library functions. A manual inspection of the cases where recall remains low (e.g., krakenjs/kraken-js) reveals that use of dynamically computed module names sometimes causes entire modules to be missed by the static analysis. Also note that these measurements are based on dynamic call graphs produced by the existing test suites of the project, and if their coverage is low, there may be recall improvements that do not show up in these comparisons. For this reason, the results for the other metrics discussed above are also important for answering the research question about analysis accuracy.

The precision results indicate that most hints are correct and do not lead to a large number of spurious call edges in the resulting call graphs.

To study how the improved recall may benefit vulnerability analysis, we have used the inferred call graphs for computing reachability of functions with known vulnerabilities in the dependencies for the benchmarks from Table 1. In total, the dependencies of the 36 projects contain 447 vulnerabilities. According to the call graphs computed by the baseline analysis, only 52 are reachable (it is not uncommon that only a small fraction of the code in dependencies is being used). With the approximate interpretation extension, this number increases to 55. The total number of reachable functions increases from 42 661 to 53 805 thereby increasing confidence that most vulnerabilities are found.

> *In the experiments involving dynamic call graphs, the average analysis recall is improved from 75.9% to 88.1%, and precision is reduced by only 1.5%.*

Regarding the second research question, the experiments show that approximate interpretation is scalable for real-world programs. It takes between 0.6 seconds and 51 seconds per program, which amounts to an average of 4.5 seconds or 3.2ms per function.

Table 3 lists the running times of the baseline static analysis, approximate interpretation, and the extended analysis that uses the produced hints for the benchmarks with dynamic call graphs. The results show that approximate interpretation is scalable to real-world programs. The time spent on approximate interpretation can be reduced by adjusting the thresholds used for aborting long running executions, which may naturally lead to fewer hints being produced, but we have not yet explored this trade-off. For most benchmarks we see only a slight increase in static analysis time when using the hints produced by approximate interpretation, but there are also cases (e.g., share/sharedb) where the analysis time increases substantially due to the extra dataflow.

As expected, not all functions end up being visited during approximate interpretation. Some function definitions are unreached because approximate interpretation is unable to cover all branches, as discussed above. In our experiments, approximate interpretation visits 60% of the functions in the analyzed programs.

> *With the proof-of-concept implementation, approximate interpretation takes between 0.6 seconds and 51 seconds per program, which amounts to an average of 4.5 seconds.*

Table 3. Running times of baseline static analysis, approximate interpretation, and extended static analysis.

Benchmark	Baseline (s)	Approx. (s)	Extended (s)
tj/co	0.5	3.4	0.5
flitbit/diff	0.8	2.0	1.0
leizongmin/js-xss	0.7	1.4	0.8
Leonidas-from-XIV/node-xml2js	1.1	1.8	1.2
creationix/js-git	1.3	2.3	1.5
jaredhanson/passport	2.0	2.4	2.2
danielstjules/jsinspect	2.5	6.8	4.4
typestack/typedi	2.7	1.4	2.7
senchalabs/connect	2.0	2.6	2.2
louischatriot/nedb*	20.1	4.0	26.4
expressjs/compression	2.3	3.0	2.3
expressjs/body-parser	3.1	3.6	3.1
express-validator/express-validator	9.5	10.4	42.3
jezen/is-thirteen	6.5	11.0	6.7
expressjs/cors	3.7	4.5	3.7
hubotio/hubot	4.1	5.4	5.0
posthtml/posthtml	3.9	5.6	5.0
krakenjs/kraken-js	9.1	4.4	10.4
visionmedia/supertest	4.6	5.4	5.3
ternjs/tern	8.0	5.3	11.3
thlorenz/doctoc	8.4	7.1	10.0
gulpjs/gulp	22.1	4.6	25.2
mroderick/PubSubJS	6.7	5.9	15.3
node-schedule/node-schedule	6.2	7.8	6.2
expressjs/multer	6.4	4.2	6.4
nodeca/js-yaml	14.4	4.1	17.8
expressjs/express	6.8	10.1	29.0
digitalbazaar/forge	6.6	12.9	6.9
IonicaBizau/scrape-it	15.8	8.9	81.9
acornjs/acorn	6.8	17.8	7.6
nexe/nexe	10.1	20.9	9.8
zemirco/json2csv	15.1	32.0	15.6
nodeca/pako	23.4	2.4	24.1
share/sharedb	20.7	10.0	102.4
MikeMcl/big.js	6.8	16.5	7.0
twbs/bootlint	31.4	4.4	94.8

Threats to validity. A potential threat to the validity of the conclusions is the choice of benchmarks; the empirical results are not guaranteed to hold for all kinds of JavaScript programs. The benchmarks are popular open source projects, and they are larger than the benchmarks typically used in research on static analysis for JavaScript [Andreasen and Møller 2014; Chakraborty et al. 2022; Feldthaus et al. 2013; Jensen et al. 2009; Kang et al. 2023; Kashyap et al. 2014; Lee et al. 2012; Nielsen et al. 2019; Park et al. 2021; Sridharan et al. 2012; Wei and Ryder 2013]. Antal et al. [2023] recently performed a study of JavaScript call graph extraction tools, showing that many call edges are missed. Their conclusions are based mostly on small performance benchmarks. The results may also be different for other static analysis tools. The Jelly analyzer we use in the experiments is considerably more accurate for real-world JavaScript programs than the static analyzers considered in that study.

Many tools and techniques have been developed for static call-graph analysis of JavaScript programs. As mentioned in Sections 1 and 2, existing approaches that aim for soundness (e.g., TAJS [Andreasen and Møller 2014; Jensen et al. 2009] and SAFE [Lee et al. 2012; Park et al. 2021]) do not scale to real-world programs. More pragmatic analysis designs (e.g. WALA [Chakraborty et al. 2022], JAM [Nielsen et al. 2021] and FAST [Kang et al. 2023]) miss many call edges and often have low precision. The *js-callgraph* [Persper Foundation 2019] implementation of the ACG

algorithm [Feldthaus et al. 2013] provides basic handling of JavaScript modules and other language features that were not supported in the original implementation of ACG, but it has been unmaintained since 2019. It fails to produce call graphs for 31 of the 36 benchmarks listed in Table 2, and for the remaining 5 it achieves significantly lower accuracy than our baseline analyzer. The WALA analyzer was initially designed for Java but has been extended with support for JavaScript. It contains an implementation of the ACG algorithm but targets browser-based applications and lacks support for many modern JavaScript language features including the ESM module system, so it is also unable to produce call graphs for these benchmarks. CodeQL [GitHub 2024] is capable of analyzing large, real-world JavaScript projects. In order to ensure scalability, it only analyzes the code of the application itself, not analyzing its dependencies directly but instead relying on (typically hand-written) library models. The FAST analyzer [Kang et al. 2023]) (which builds on ODGen [Li et al. 2022]) lacks support for commonly used language features, such as, iterators, generators, and getters/setters, and ignores calls to most native functions from the ECMAScript standard library. Like CodeQL, FAST requires hand-written library models (the current version contains simple models for 15 of the 2.7 million libraries available in the npm registry). Running FAST without using the hand-written models of third-party libraries (such that it includes the actual library code in the analysis) on the motivating example from Section 2, it reports only 17 of the 138 actual call edges (recall: 12.3%) after 11 seconds. In comparison, the whole-program analyzer used in this work finds 136 of the call edges (recall: 98.5%) in 3 seconds (including time for approximate interpretation).

6 POTENTIAL IMPROVEMENTS

In this section we discuss opportunities for future work that can potentially improve static analysis accuracy further by extending approximate interpretation to produce more hints.

Dynamically generated code. As mentioned in Section 3, approximate interpretation does work in presence of code that is dynamically generated with `eval` and similar language constructs, in the sense that it is possible to infer useful hints in such code. However, it is possible to go one step further and use approximate interpretation to produce additional hints to the static analysis in the form of strings of program code that appear during the executions at calls to, for example, `eval`. The static analysis could then be extended to treat such hints as additional code to be analyzed. Jensen et al. [2012] have shown that many occurrences of `eval` in practice are determinate, in which case this approach will be effective.

Unknown function arguments. The mechanism described in Section 3 does not produce hints when dynamic property access operations involve the synthetic proxy object that is used for representing unknown function arguments. For example, assume a dynamic property read $\texttt{x[y]}_\ell$ is encountered during approximate interpretation where the value of `x` is $\mathbf{p}^\star$ and the value of `y` is the string `"p"`, in which case no hints are currently generated. We could produce a hint $(\ell,\texttt{"p"})$, which would allow the static analysis to treat the operation as a static property read, `x.p`, instead of ignoring it. This might negatively affect analysis precision if the operation occurs in a polymorphic function, so this kind of hint should only be produced when no hints would otherwise be produced.

Approximate interpretation of function fragments. Approximate interpretation may fail to reach all program code and therefore may miss useful determinacy facts. This may happen, for example, if a branch condition involves the proxy object that we use for representing unknown function arguments. As observed in Section 5, a considerable fraction of the function definitions remain unvisited. For this reason it may be worthwhile to force-execute not only entire functions but also individual branches of code that are otherwise not being covered.

Reusing approximate interpretation results. Studies have shown that more than 90% of the code in a typical Node.js application comes from third-party library code [Koishybayev and Kapravelos 2020]. This provides an opportunity for reusing results of approximate interpretation. Notice that in the example from Section 2, all the interesting hints come from the Express library code, not the application code. This means that once the Express library has been subjected to approximate interpretation, the produced hints can be reused for the thousands of applications that use the library.

Context sensitivity. As shown in Section 4, the idea of recording the origin locations of the objects that are involved in dynamic property accesses provides relational information that is highly useful for static analysis that it not necessarily context sensitive. Although this paper focuses on improving analysis recall, precision is naturally also important, and context sensitivity is one of the main tools for making static analyses more precise. If using approximate interpretation together with context-sensitive analysis, it is possible to increase analysis precision by qualifying the recorded hints with calling context information, much like the use of context-qualified determinacy facts in dynamic determinacy analysis [Schäfer et al. 2013]. If, for example, the static analysis uses 1-call-site-sensitive analysis with context-sensitive heap [Kastrinis and Smaragdakis 2013], the approximate interpretation algorithm could be extended to keep track of the top-most call site in the call stack and then include that information in the hints being produced. Specifically, every location $\ell \in Loc$ thereby becomes a pair of a calling context and a source code location. The static analysis could then apply the hints only for the matching calling contexts and thereby improve precision.

An example where such use of context-sensitive hints could lead to improved precision is the expression `route[method]` in Figure 1d (line 41). In the inner function (lines 38–43), `method` is a free variable that refers to the parameter of the enclosing function. This means that when `app.get` and `app.listen` are invoked, a method of the same name gets invoked on the `route` objects (line 41). Reasoning about this correlation of function calls requires context sensitive analysis. If the read hints produced for `route[method]` are qualified by the calling contexts, it is possible for a context-sensitive static analysis to use them for resolving those calls both soundly and precisely.

7 RELATED WORK

Approximate interpretation is, to our knowledge, the first technique that uses forced execution to reduce static analysis unsoundness. The idea behind forcing executions of JavaScript program fragments has previously been explored by Hu et al. [2017] and Kim et al. [2017]. Their tools, JSForce and J-Force, enumerate the behavior of applications by forcing execution of branches to uncover malicious code otherwise hidden by cloaking or other obfuscation techniques.

Dynamic determinacy analysis by Schäfer et al. [2013] is closely related to our approach but there are some fundamental differences as discussed in the preceding sections. First, instead of exploring the program code automatically, it requires the user to provide one or more initial executions which are then subjected to counterfactual execution to explore alternative branches. Second, it produces individual determinacy facts that do not provide relational information (see Section 4). Third, the inferred determinacy facts are of limited use in static analyses that are not context- and path-sensitive (see Section 2), unlike the approach presented here.

Static determinacy analysis by Andreasen and Møller [2014] instead infers and takes advantage of determinacy facts using a purely static analysis with flow-, context- and path-sensitivity as well as constant propagation. Unlike approximate interpretation, it has not been demonstrated to be scalable to large, real-world JavaScript applications.

The idea of utilizing information obtained from dynamic program executions for improving static analysis has appeared in many variations. The HeapDL technique by Grech et al. [2017] aims to reduce the degree of unsoundness in static analysis for Java by the use of HPROF heap snapshots captured during dynamic execution. Like our approach for JavaScript, HeapDL leads to more call graph edges being discovered by the static analysis. The technique has strong similarities to our approach, but inferring information from snapshots is substantially different from recording relevant information during analysis. Not only is our approach easier to implement for a language like JavaScript; it also allows us to capture relevant operations on objects that may no longer exist at the time of the snapshot. Additionally, HeapDL requires test suites or manually exercising the program under analysis, unlike the approximate interpretation which is fully automatic.

Another analysis technique that combines dynamic and static analysis is blended analysis by Wei and Ryder [2013], which is a fast but unsound form of static taint analysis performed on individual execution traces obtained by running test suites for JavaScript programs. Andreasen et al. [2017] have used dynamic analysis for helping analysis designers investigate causes of soundness and precision problems. The recent work by Chakraborty et al. [2022] uses dynamic analysis to automatically quantify the importance of different root causes of unsoundness in static call graph analysis for JavaScript. As mentioned in the introduction, their results show that dynamic property accesses tend to be the major reason for low recall, which is exactly what our analysis technique is designed to address.

Park et al. [2021] introduced a technique called dynamic shortcuts, which are concrete executions of selected functions performed during static analysis. The concrete executions use proxy objects to represent abstract values, similar to our use of a proxy object for representing unknown values. The technique aims to improve performance and precision, not recall as our approach.

The Concerto system by Toman and Grossman [2019] combines concrete interpretation and abstract interpretation for analysis of framework-based applications. Application code is analyzed using abstract interpretation, and framework code is analyzed using mostly-concrete execution, based on the observation that framework implementations are difficult to analyze statically but can often be deterministically evaluated at analysis time, which is essentially the same observation used in the work on dynamic and static determinacy as discussed above. To our knowledge, Concerto has so far only been implemented for a subset of Java and evaluated on a minimalistic case study, and it is unclear how the approach could be adapted to JavaScript.

8 CONCLUSION

Approximate interpretation is a simple yet effective technique for learning likely determinate facts about complex dynamic object manipulation in JavaScript programs. By inferring not only the string values that appear as property names in dynamic property accesses but also the allocation sites and function definitions of the involved objects, it provides a powerful alternative to dynamic determinacy analysis. The experimental results demonstrate that the technique works on real-world JavaScript code and succeeds in reducing the degree of unsoundness of an existing state-of-the-art static call graph analysis.

In addition to exploring the ideas outlined in Section 6, for future work it may be interesting to combine approximate interpretation and fuzzing or symbolic execution to address the limitations of the use of proxy objects for handling unknown parameter values more precisely, although it will likely make the dynamic analysis phase slower. Another direction could be adapting the approximate interpretation technique to other languages, for example Java, to provide an alternative to TamiFlex and HeapDL-style approaches for resolving reflective code and dynamic class loading.

DATA AVAILABILITY STATEMENT

The source code of the Jelly program analysis tool including the implementation of the approximate interpretation mechanism is available at https://github.com/cs-au-dk/jelly. An artifact that contains Jelly and the benchmarks and scripts used for the experimental evaluation is archived on Zenodo [Møller et al. 2024].

REFERENCES

Esben Andreasen and Anders Møller. 2014. Determinacy in static analysis for jQuery. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 17–31. https://doi.org/10.1145/2660193.2660214

Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. 2017. Systematic approaches for increasing soundness and precision of static analyzers. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2017, Barcelona, Spain, June 18, 2017*, Karim Ali and Cristina Cifuentes (Eds.). ACM, 31–36. https://doi.org/10.1145/3088515.3088521

Gábor Antal, Péter Hegedüs, Zoltán Herczeg, Gábor Lóki, and Rudolf Ferenc. 2023. Is JavaScript Call Graph Extraction Solved Yet? A Comparative Study of Static and Dynamic Tools. *IEEE Access* 11 (2023), 25266–25284. https://doi.org/10.1109/ACCESS.2023.3255984

Babel Team. 2024. Babel. https://babeljs.io/.

Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 241–250. https://doi.org/10.1145/1985793.1985827

Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. 2022. Automatic Root Cause Quantification for Missing Edges in JavaScript Call Graphs. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany (LIPIcs, Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:28. https://doi.org/10.4230/LIPICS.ECOOP.2022.3

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

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

GitHub. 2024. CodeQL. https://codeql.github.com/.

Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. 2017. Heaps don't lie: countering unsoundness with heap snapshots. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 68:1–68:27. https://doi.org/10.1145/3133892

Salvatore Guarnieri and V. Benjamin Livshits. 2009. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, Fabian Monrose (Ed.). USENIX Association, 151–168. http://www.usenix.org/events/sec09/tech/full_papers/guarnieri.pdf

Xunchao Hu, Yao Cheng, Yue Duan, Andrew Henderson, and Heng Yin. 2017. JSForce: A Forced Execution Engine for Malicious JavaScript Detection. In *Security and Privacy in Communication Networks - 13th International Conference, SecureComm 2017, Niagara Falls, ON, Canada, October 22-25, 2017, Proceedings (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Vol. 238)*, Xiaodong Lin, Ali A. Ghorbani, Kui Ren, Sencun Zhu, and Aiqing Zhang (Eds.). Springer, 704–720. https://doi.org/10.1007/978-3-319-78813-5_37

Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. 2012. Remedying the eval that men do. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, Mats Per Erik Heimdahl and Zhendong Su (Eds.). ACM, 34–44. https://doi.org/10.1145/2338965.2336758

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5673)*, Jens Palsberg and Zhendong Su (Eds.). Springer, 238–255. https://doi.org/10.1007/978-3-642-03237-0_17

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

Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: a static analysis platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 121–132. https://doi.org/10.1145/2635868.2635904

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

Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. 2017. J-Force: Forced Execution on JavaScript. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, Rick Barrett, Rick Cummings, Eugene Agichtein, and Evgeniy Gabrilovich (Eds.). ACM, 897–906. https://doi.org/10.1145/3038912.3052674

Igibek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the Attack Surface of Node.js Applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2020, San Sebastian, Spain, October 14-15, 2020*, Manuel Egele and Leyla Bilge (Eds.). USENIX Association, 121–134. https://www.usenix.org/conference/raid2020/presentation/koishybayev

Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. 2012. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages*. 96.

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

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (2015), 44–46. https://doi.org/10.1145/2644805

Magnus Madsen and Esben Andreasen. 2014. String Analysis for Dynamic Field Access. In *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8409)*, Albert Cohen (Ed.). Springer, 197–217. https://doi.org/10.1007/978-3-642-54807-9_12

Anders Møller, Mathias Rud Laursen, and Wenyuan Xu. 2024. Artifact for "Reducing Static Analysis Unsoundness with Approximate Interpretation", PLDI 2024. https://doi.org/10.5281/zenodo.10796242.

Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. 2019. Nodest: feedback-driven static analysis of Node.js applications. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 455–465. https://doi.org/10.1145/3338906.3338933

Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. 2021. Modular call graph construction for security scanning of Node.js applications. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 29–41. https://doi.org/10.1145/3460319.3464836

Joonyoung Park, Jihyeok Park, Dongjun Youn, and Sukyoung Ryu. 2021. Accelerating JavaScript static analysis via dynamic shortcuts. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 1129–1140. https://doi.org/10.1145/3468264.3468556

Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-Force: Force-Executing Binary Programs for Security Applications. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 829–844. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/peng

Persper Foundation. 2019. js-callgraph. https://github.com/Persper/js-callgraph.

Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications. In *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings (Lecture Notes in Computer Science, Vol. 6813)*, Mira Mezini (Ed.). Springer, 52–78. https://doi.org/10.1007/978-3-642-22655-7_4

Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Dynamic determinacy 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, 165–174. https://doi.org/10.1145/2491956.2462168

Yannis Smaragdakis and George Kastrinis. 2018. Defensive Points-To Analysis: Effective Soundness via Laziness. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands (LIPIcs,*

Vol. 109), Todd D. Millstein (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:28. https://doi.org/10.4230/LIPICS.ECOOP.2018.23

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

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

Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. 2018. Efficient dynamic analysis for Node.js. In *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*, Christophe Dubach and Jingling Xue (Eds.). ACM, 196–206. https://doi.org/10.1145/3178372.3179527

John Toman and Dan Grossman. 2019. Concerto: a framework for combined concrete and abstract interpretation. *Proc. ACM Program. Lang.* 3, POPL (2019), 43:1–43:29. https://doi.org/10.1145/3290356

Shiyi Wei and Barbara G. Ryder. 2013. Practical blended taint analysis for JavaScript. In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, Mauro Pezzè and Mark Harman (Eds.). ACM, 336–346. https://doi.org/10.1145/2483760.2483788

Received 2023-11-16; accepted 2024-03-31