2015 30th IEEE/ACM International Conference on Automated Software Engineering

Practically Tunable Static Analysis Framework for Large-Scale JavaScript Applications

Yoonseok Ko
KAIST
mir597@kaist.ac.kr

Hongki Lee
KAIST
petitkan@kaist.ac.kr

Julian Dolby
IBM Research
dolby@us.ibm.com

Sukyoung Ryu
KAIST
sryu.cs@kaist.ac.kr

***Abstract*—We present a novel approach to analyze large-scale JavaScript applications statically by *tuning* the analysis scalability possibly giving up its soundness. For a given sound static *baseline analysis* of JavaScript programs, our framework allows users to define a sound approximation of *selected executions* that they are interested in analyzing, and it derives a *tuned static analysis* that can analyze the selected executions practically. The selected executions serve as parameters of the framework by taking trade-off between the scalability and the soundness of derived analyses. We formally describe our framework in abstract interpretation, and implement two instances of the framework. We evaluate them by analyzing large-scale real-world JavaScript applications, and the evaluation results show that the framework indeed empowers users to experiment with different levels of scalability and soundness. Our implementation provides an extra level of scalability by deriving *sparse* versions of derived analyses, and the implementation is publicly available.**

I. Introduction

JavaScript applications are prevalent these days and their sizes become larger and their program logics become more complex. One can find various kinds of applications written in JavaScript including games [1], compilers [2], and even operating systems [3], [4]. The extensive uses of multiple large-scale libraries[1] such as jQuery[2], MooTools[3], Prototype[4], and YUI[5] are one of the sources of the problem.

However, statically analyzing large-scale JavaScript applications is very challenging. Extremely dynamic features of JavaScript like run-time code generation and heavy uses of first-class functions complicate static analysis of program flows [5]. In addition to normal control flows in programs, JavaScript involves diverse exception flows, which add complications to control flow graphs of programs and, in turn, to their static analysis. While most C program analyses ignore infrequent uses of exception behaviors by `setjmp` and `longjmp`, and ML-like program analyses use strongly-typed language features to analyze exception flows and to detect uncaught exceptions [6], [7], dynamic changes of object properties including prototypes and the lack of static type system in JavaScript make the static analysis of its exception flows much more complex and unscalable.

[1]JavaScript libraries are often called frameworks.
[2]http://jquery.com
[3]http://mootools.net
[4]http://prototypejs.org
[5]http://yuilibrary.com

For large-scale C programs, a general sparse analysis framework [8], [9], [10] has been very successful in implementing practical analyzers, but it may not be applicable to other languages like JavaScript. The authors discussed it as an open issue as follows:

> Applying our framework for other languages may be more difficult than that for C. For instance, for dynamic languages such as JavaScript, our simple flow-insensitive pre-analysis may not be effective, since too imprecise analyses can take much time for those languages. Designing a pre-analysis that is cheap yet precise for other languages remains an open problem.

Indeed, our experimental results show that simply applying the general sparse analysis framework to JavaScript is not effective.

Unfortunately, even state-of-the-art JavaScript analysis techniques are often sound but impractical, or practical but unsound. Most JavaScript analysis frameworks like SAFE [11], [12], TAJS [13], [14], and WALA [15], [16], [17] provide traditional sound static analyzers, which are not yet practically scalable. In addition, WALA provides a new *intentionally unsound* static analysis especially designed for scalability. Since WALA was originally designed for sound static analysis of Java programs, it provides a sound *Propagation-Based analysis (PB)* of JavaScript programs similarly for Java program analysis. However, because such a sound analysis has shown to be impractical not being able to analyze the most widely used JavaScript library, jQuery, WALA provides yet another JavaScript analysis, an unsound *Field-Based analysis (FB)*, by constructing lightweight and unsound call graphs [18], which has been successfully used for a commercial product IBM Security AppScan [19].

Given that the problem of scalable sound static analysis of large JavaScript applications is still an open problem, users may get more benefits from focusing on a subset of program executions. Instead of attempting to analyze the entire program flows, analyzing program executions selectively may be a reasonable option. As a recently introduced term *soundiness* [20] denotes, "attempting to capture the balance, prevalent in practice, of over-approximated handling of most language features, yet deliberately under-approximated handling of a feature subset well recognized by experts" may be a desirable

978-1-5090-0025-8/15 $31.00 © 2015 IEEE
DOI 10.1109/ASE.2015.28

approach to take, and many modern analysis applications like IDEs, security analyses, and bug detectors do not require soundness. As we describe in more detail later in this paper, one may prefer to use a sound and elaborate analysis to understand every possible execution flow of a given program. Or, one may want to use an efficient analysis for a selected subset of program executions that they are interested in.

In this paper, we present a novel approach to analyze large-scale JavaScript applications statically by tuning sound *baseline analyses* for scalability using approximation of *selected executions* of interest to users possibly giving up soundness. Our framework takes a static baseline analysis that over-approximates the JavaScript semantics and a *pre-analysis* that over-approximates a subset of program executions, and produces a *tuned static analysis* that can analyze the selected executions practically. The framework allows users to identify a subset of program execution flows that they are interested in analyzing to take trade-off between the scalability and the soundness of derived analyses.

Users can tune the trade-off between the scalability and the soundness of derived analyses by using pre-analyses of selected executions as parameters. The fewer program execution flows a pre-analysis covers, the more a derived analysis would be scalable but unsound. We formally describe our framework in the abstract interpretation setting [21], [22] and evaluate the framework by using two instances: 1) SWP derived from SAFE as a sound baseline analysis and WALA's PB as a sound pre-analysis; and 2) SWF derived from SAFE as a sound baseline analysis and WALA's FB as an unsound pre-analysis. Our experiments with real-world JavaScript applications show that SWF may miss some analysis results that SWP can obtain, but it is indeed more scalable and still precise. Users may want to find an appropriate balance between the soundness and the scalability by tuning the coverage of pre-analyses. Our implementation also provides an extra level of scalability in terms of analysis performance by deriving *sparse* versions of derived analyses, and the implementation is publicly available [23].

The contributions of this paper are as follows:

- We present a novel approach that allows users to decide *the trade-off between the scalability and the soundness* of static analysis.
- We *formally present* the framework in the abstract interpretation setting. It clearly describes assumptions and requirements to use the framework, and it proves the safety of derived analyses.
- We show how to use the general framework with *two concrete instances* using the state-of-the-art JavaScript analyzers, SAFE and WALA. The instance SWP shows that simply applying the general sparse analysis framework to JavaScript may not be effective, and the instance SWF shows that focusing on selected program executions enables analysis of large-scale JavaScript applications.
- We make our implementation *publicly available* [23]. Our experimental results show that the framework derives tuned static analyses that can analyze real-world JavaScript applications practically.

The rest of the paper is organized as follows. We illustrate complications of statically analyzing JavaScript libraries using a concrete code example from the YUI library (Section II). We formally present the framework that takes a baseline analysis and a pre-analysis as inputs and derives a tuned static analysis in the abstract interpretation setting, and we prove the safety of the derived analyses (Section III). To show how to instantiate the general framework, we describe two concrete instances of the framework using SAFE as a baseline analyzer and two analyses of WALA as pre-analyses (Section IV). Then, we describe our implementation (Section V) and evaluate the framework using two instances (Section VI). We discuss related work (Section VII) and conclude (Section VIII).

II. Motivating Example

While much progress has been made in analyzing modern JavaScript applications that rely heavily on multiple libraries, significant challenges still remain for achieving a good balance of the analysis cost and precision. This is largely due to library code making extensive use of dynamic JavaScript features, especially first-class property accesses. We use Figure 1 taken from the YUI library to illustrate the situation.

The `mix` function in Figure 1 essentially implements a form of mixin-based inheritance [24] to provide object-oriented features within JavaScript code. The mechanism is to copy properties (both code and data) from its second argument `supp` to its first argument `recv` mimicking the effect of inheritance. The other arguments of `mix` determine the details of copying: `overwrite` and `merge` denote whether to overwrite or merge properties; `white`, if specified, lists properties to be copied and excludes all other properties from copying; and `mode` denotes whether and how to copy prototypes of objects. Thus, lines 8 ~ 22 determine the source `from` and the destination `to` of copying depending on `mode`, lines 26 ~ 42 copy only the whitelisted properties, and lines 44 ~ 62 copy all the properties.

This poses a challenge to analyses that have to reason about statements like `to[key] = from[key]` on lines 40 and 56; a naïve analysis that simply approximates the values of `key` with the set of all possible property names will produce an extremely imprecise result that all properties of `from` flow into all properties of `to`. In addition, traditional analyses have also suffered from extreme cost, as the level of imprecision has caused tremendous amounts of propagation. Attempts to provide better results have largely taken two approaches.

One approach [16], [17], [25], [26] has been to apply ever-more-aggressive forms of analysis sensitivity to disambiguate the behavior, such as aggressive flow- and context sensitivity, transformations of `for-in` loop bodies to expose more opportunities for context sensitivity and even exploiting dynamic information to improve static analysis. This approach has shown the ability to handle ever-increasing complexity in JavaScript but nothing has yet been shown to robustly handle modern framework-based code.

For any calls to `mix`, for example, distinguishing the use of different properties in each loop iteration requires multiple applications of techniques such as correlation tracking [16].

```
1  Y.mix = function(recv, supp, overwrite, white,
2                   mode, merge) {
3    var alwaysOverwrite, exists, from, i, key,
4        len, to;
5
6    if (!recv || !supp) return recv || Y;
7
8    if (mode) {
9      if (mode === 2) {
10       Y.mix(recv.prototype,supp.prototype,
11             overwrite, white, 0, merge);
12     }
13     from = mode === 1 || mode === 3 ?
14               supp.prototype : supp;
15     to   = mode === 1 || mode === 4 ?
16               recv.prototype : recv;
17
18     if (!from || !to) return recv;
19   } else {
20     from = supp;
21     to   = recv;
22   }
23
24   alwaysOverwrite = overwrite && !merge;
25   if (white) {
26     for (i=0,len=white.length; i<len; ++i) {
27       key = white[i];
28
29       if (!hasOwn.call(from, key))
30         continue;
31       exists = alwaysOverwrite ?
32                  false : key in to;
33
34       if (merge && exists &&
35           isObject(to[key], true) &&
36           isObject(from[key], true)) {
37         Y.mix(to[key], from[key], overwrite,
38               null, 0, merge);
39       } else if (overwrite || !exists) {
40         to[key] = from[key];
41       }
42     }
43   } else {
44     for (key in from) {
45       if (!hasOwn.call(from, key))
46         continue;
47       exists = alwaysOverwrite ?
48                  false : key in to;
49
50       if (merge && exists &&
51           isObject(to[key], true) &&
52           isObject(from[key], true)) {
53         Y.mix(to[key], from[key], overwrite,
54               null, 0, merge);
55       } else if (overwrite || !exists) {
56         to[key] = from[key];
57       }
58     }
59     if (Y.Object._hasEnumBug) {
60       Y.mix(to, from, overwrite,
61             Y.Object._forceEnum, mode, merge);
62     }
63   }
64   return recv;
65 };
```

Fig. 1: Code excerpt from the YUI library

But more is required, since precision also depends on precise handling of `merge` to avoid potential pollution from extraneous recursive copies on lines 37 and 53. And beyond that, note that calls to `mix` may use different values of `mode`: 0 on lines 10, 37, and 53 but `mode` on line 60. There are four possible options which require understanding constants to distinguish, once again needed to avoid pollution from extraneous copies.

Given this degree of complication, the other approach [18], [27] has gone the other way, making the analysis less precise to address the cost explosion of traditional analysis. The field-based approach [18], for example, abandons object sensitivity entirely, using a single location globally for each field. This approach addresses the issue of statements like `to[key] = from[key]`, since they become no-ops for analysis and can be ignored. While this approach has been shown to scale, it loses precision and/or soundness in many cases.

However, this approach suffers from severe, but different, issues of its own. Even a very simple code like the following:

```
1  var f = { x: function a() { ... } };
2  var g = { x: function b() { ... } };
3  f.x();
```

will be imprecise; because a field-based analysis does not distinguish between fields in different objects with the same name, it concludes that `f.x` on line 3 points to either function `a` or `b`. Thus, there is still no technique in the state of the art that robustly handles modern JavaScript libraries and reliably provides good precision and performance.

This motivates our tunable static analysis framework. Even though a sound analysis alone or the field-based analysis alone may not produce precise analysis results, a combination of two analyses may generate better results than that of each analysis. For example, consider the following code using the `mix` function:

```
1  var f = { x: function a() { ... },
2            y: function b() { ... } };
3  var g = { x: function c() { ... } };
4  var h = Y.mix({}, f);
5  h.x();
```

A sound analysis may conclude that `h.x` on line 5 points to either function `a` or `b` due to the analysis complexity of `mix`. On the contrary, the field-based analysis will conclude that `h.x` points to either function `a` or `c` because it considers only the property names. Now, consider a tuned analysis which performs a sound analysis but only within the results of the field-based analysis; the analysis results will lie inside the intersection of the results of the two analyses. The tuned analysis gets benefits from both analyses, which makes its static analysis simpler, more efficient, and more precise than static analysis of each of them.

Our goal is to present a general framework that automatically derives a tuned analysis from two analyses. The framework takes a sound baseline analysis and a (possibly unsound) pre-analysis generating a boundary of analysis results like the field-based analysis, and it produces a tuned analysis that performs the baseline analysis only within the boundary.

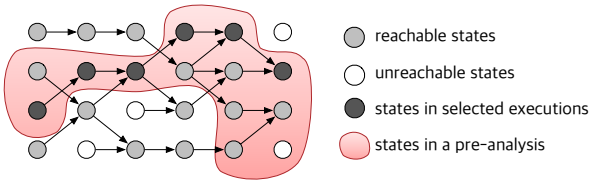

Fig. 2: Program execution flows

III. Tunable Static Analysis Framework

In this section, we describe our framework which enables to tune the scalability of static analysis. Consider the execution flows illustrated in Figure 2. Light gray circles denote reachable states during evaluation, white circles denote unreachable states during evaluation, and dark gray circles denote states in selected execution flows. A sound baseline analysis approximates all the reachable states possibly including some unreachable states (Section III-A), and a pre-analysis approximates the states in the selected execution flows as denoted by a closed loop in Figure 2 (Section III-B). It is a "sound" approximation of the selected execution flows, and, at the same time, it is an "unsound" approximation of the entire execution flows. The framework allows users to tune the derived analysis (Section III-C) by choosing appropriate subsets of program executions. We discuss the impacts of selected program executions in terms of the analysis scalability, soundness, and precision (Section III-D).

A. Baseline Analysis for the Entire Semantics

The first input to the framework is a baseline static analysis that over-approximates the entire semantics of given programs. The framework assumes that the baseline analysis is designed in the abstract interpretation setting [21], [22].

Collecting Semantics. We denote a program $\mathcal{P}$ as a directed graph $\langle \mathbb{C}, \mathbb{E} \rangle$ where $\mathbb{C}$ and $\mathbb{E}$ represent a set of nodes and a set of edges, respectively. A node $c \in \mathbb{C}$ denotes a control point in the program and an edge $(c, c') \in \mathbb{E} \subseteq \mathbb{C} \times \mathbb{C}$ denotes a control flow from c to c'.

Given a program $\mathcal{P}$, we define its collecting semantics using $\langle \mathbb{C}, \wp(\mathbb{S}), f, \rightarrow_\phi \rangle$ where:

- $\mathbb{C}$: a finite set of control points;
- $\sigma \in \wp(\mathbb{S})$: a powerset of concrete states;
- $f(c) \in \wp(\mathbb{S}) \rightarrow \wp(\mathbb{S})$: a set of local semantic functions at a given control point c;
- $\phi \in \mathbb{C} \rightarrow \wp(\mathbb{S})$: a set of program states mapping control points to sets of concrete states; and
- $\rightarrow_\phi$: a set of control flows for a given program state ϕ

for a set of concrete states $\mathbb{S}$. A local semantic function $f(c)$ takes a set of concrete states and returns a union of sets of resulting states for each of the given states. A set of control flows $\rightarrow_\phi$ is a subset of control flows $\mathbb{E}$ and it is confined by a given program state ϕ. Then, the collecting semantics of a program $\mathcal{P}$ is the least fixpoint of the following semantic function:

$$F \in (\mathbb{C} \rightarrow \wp(\mathbb{S})) \rightarrow (\mathbb{C} \rightarrow \wp(\mathbb{S}))$$
$$F(\phi) = \lambda c \in \mathbb{C}.\ f(c)\left(\bigcup_{c' \rightarrow_\phi c} \phi(c')\right).$$

Baseline Abstraction. For a baseline analysis that our framework tunes for scalability, we require a static analysis designed in the abstract interpretation framework. We assume that it abstracts a set of concrete states σ to an abstract state $\hat{\sigma} \in \hat{\mathbb{S}}$ by the following Galois connection:

$$\wp(\mathbb{S}) \underset{\alpha_\mathbb{S}}{\overset{\gamma_\mathbb{S}}{\leftrightarrows}} \hat{\mathbb{S}}.$$

While the framework does not require any structure on $\hat{\mathbb{S}}$, when $\hat{\mathbb{S}}$ is a function cpo (complete partial order) $\hat{\mathbb{L}} \rightarrow \hat{\mathbb{V}}$ for a finite set of abstract locations $\hat{l} \in \hat{\mathbb{L}}$ and an arbitrary cpo of abstract values $\hat{v} \in \hat{\mathbb{V}}$, the framework provides an extra level of scalability in terms of the analysis performance as we discuss in Section V. Then, the abstraction of a program state ϕ and the concretization of an abstract program state $\hat{\phi} \in \mathbb{C} \rightarrow \hat{\mathbb{S}}$ for the baseline analysis are as follows:

$$\alpha_m(\phi) = \lambda c \in \mathbb{C}.\ \alpha_\mathbb{S}(\phi(c))$$
$$\gamma_m(\hat{\phi}) = \lambda c \in \mathbb{C}.\ \gamma_\mathbb{S}(\hat{\phi}(c)).$$

The abstract semantics of the baseline analysis is the least fixpoint of the following semantic function:

$$\hat{F} \in (\mathbb{C} \rightarrow \hat{\mathbb{S}}) \rightarrow (\mathbb{C} \rightarrow \hat{\mathbb{S}})$$
$$\hat{F}(\hat{\phi}) = \lambda c \in \mathbb{C}.\ \hat{f}(c)\left(\bigsqcup_{c' \hookrightarrow_{\hat{\phi}} c} \hat{\phi}(c')\right)$$

where $\hat{f}(c) \in \hat{\mathbb{S}} \rightarrow \hat{\mathbb{S}}$ is an abstract semantic function at a given control point c and $\hookrightarrow_{\hat{\phi}} \subseteq \mathbb{C} \times \mathbb{C}$ is a set of abstract control flows for a given abstract program state $\hat{\phi}$.

The abstract semantic function $\hat{f}$ is a monotonic function designed in the abstract interpretation framework such that:

- $\forall c \in \mathbb{C}. \forall \hat{\sigma}, \hat{\sigma}' \in \hat{\mathbb{S}}.\ \hat{\sigma} \sqsubseteq \hat{\sigma}' \Rightarrow \hat{f}(c)(\hat{\sigma}) \sqsubseteq \hat{f}(c)(\hat{\sigma}')$
- $\forall c \in \mathbb{C}. \alpha_\mathbb{S} \circ f(c) \sqsubseteq \hat{f}(c) \circ \alpha_\mathbb{S}$

The set of abstract control flows $\hookrightarrow \subseteq (\mathbb{C} \rightarrow \hat{\mathbb{S}}) \rightarrow \mathbb{C} \times \mathbb{C}$ is an abstract counterpart of $\rightarrow \subseteq (\mathbb{C} \rightarrow \wp(\mathbb{S})) \rightarrow \mathbb{C} \times \mathbb{C}$ satisfying the following conditions:

1) $\hookrightarrow_{\alpha_m(\phi)} \supseteq \rightarrow_\phi$
2) $\forall \hat{\phi}, \hat{\phi}' \in \mathbb{C} \rightarrow \hat{\mathbb{S}}.\ \hat{\phi} \sqsubseteq \hat{\phi}' \Rightarrow \hookrightarrow_{\hat{\phi}} \subseteq \hookrightarrow_{\hat{\phi}'}$

The first condition means that the set of abstract control flows $\hookrightarrow_{\alpha_m(\phi)}$ is a sound approximation of the concrete control flows for a given program state ϕ, and the second condition means that $\hookrightarrow$ is monotonic.

B. Pre-Analysis for Selected Semantics

The second input to the framework is a pre-analysis that over-approximates a set of selected execution flows of a given program. In order to analyze only the execution flows of interest, our framework asks for an approximation of the selected subset of the whole program executions. The scalability of the analysis is parameterized by this set of selected executions.

Selected Collecting Semantics. Suppose that we are interested in analyzing selected execution flows specified by a local semantic function f_s such that:

$$\forall c \in \mathbb{C}, \sigma \in \wp(\mathbb{S}).\ f_s(c)(\sigma) \subseteq f(c)(\sigma).$$

Then, we can define a *selected collecting semantics* that subsumes only the selected execution flows denoted by f_s as the least fixpoint of the following semantic function:

$$F_s \in (\mathbb{C} \to \wp(\mathbb{S})) \to (\mathbb{C} \to \wp(\mathbb{S}))$$
$$F_s(\phi) = \lambda c \in \mathbb{C}.\ f_s(c)\left(\bigcup_{c' \to_\phi c} \phi(c')\right).$$

Abstraction for Selected Collecting Semantics. For a pre-analysis that our framework uses to tune the baseline analysis for scalability, we do not require it be designed in the abstract interpretation framework unlike for the baseline analysis. The framework does not assume anything about the pre-analysis except that it produces a *contour* which over-approximates the selected execution flows we are interested in. A *contour* $\dot{C} \in \dot{\mathbb{X}}$ for some poset (partially ordered set) $\dot{\mathbb{X}}$ over-approximates the selected collecting semantics: $\alpha_s(\mathbf{lfp}F_s) \sqsubseteq \dot{C}$ holds for the following Galois connection:

$$\mathbb{C} \to \wp(\mathbb{S}) \underset{\alpha_s}{\overset{\gamma_s}{\leftrightarrows}} \dot{\mathbb{X}}.$$

Note that $\dot{\mathbb{X}}$ may be $\mathbb{C} \to \wp(\mathbb{S})$ like the collecting domain, $\mathbb{C} \to \hat{\mathbb{S}}$ like the abstract domain, or some other domain.

Because the contour is an element in the domain $\dot{\mathbb{X}}$, our framework requires an "interpretation" of the contour in terms of the abstract domain by an interpretation function:

$$\mathcal{I} \in \dot{\mathbb{X}} \to (\mathbb{C} \to \hat{\mathbb{S}}) \text{ such that } \alpha_m \circ \gamma_s \sqsubseteq \mathcal{I}.$$

Then, $\mathcal{I}(\dot{C})$ denotes an over-approximation of the selected executions in the abstract domain. When $\dot{\mathbb{X}}$ is $\mathbb{C} \to \wp(\mathbb{S})$ or $\mathbb{C} \to \hat{\mathbb{S}}$, $\mathcal{I}$ is α_m or the identity function, respectively. In summary, the relations between the collecting domain, the abstract domain, and $\dot{\mathbb{X}}$ are as follows:

$$\begin{array}{ccc} \mathbb{C} \to \wp(\mathbb{S}) & \underset{\alpha_m}{\overset{\gamma_m}{\leftrightarrows}} & \mathbb{C} \to \hat{\mathbb{S}} \\ & \overset{\gamma_s}{\underset{\alpha_s}{\swarrow\!\!\!\searrow}} & \uparrow_{\mathcal{I}} \\ & & \dot{\mathbb{X}} \end{array}$$

C. Tuned Static Analysis

Finally, using the baseline analysis $\hat{F}$ and the contour $\dot{C}$, our framework derives a tuned static analysis which over-approximates the selected set of program executions approximated by $\dot{C}$. Using the sound abstract semantic function $\hat{f}$ from the baseline analysis and the interpretation of the contour $\mathcal{I}(\dot{C})$, we can define a tuned analysis which over-approximates selected executions as the least fixpoint of the following abstract semantic function:

$$\hat{F}_s(\hat{\phi}) = \lambda c \in \mathbb{C}.\ \hat{f}(c)\left(\bigsqcup_{c' \overset{s}{\hookrightarrow}_{\hat{\phi}} c} \hat{\phi}(c')\right) \sqcap \mathcal{I}(\dot{C}),$$

where the abstract semantic function $\hat{f}$ considers only the control flows restricted by the given contour:

$$\overset{s}{\hookrightarrow}_{\hat{\phi}} \overset{\text{def}}{=} \hookrightarrow_{\hat{\phi}} \cap \hookrightarrow_{\mathcal{I}(\dot{C})}.$$

Thus, the derived analysis $\hat{F}_s$ considers only the control flows restricted by the contour, and the resulting abstract states from $\hat{f}$ remain within the interpretation of the contour because $\hat{F}_s$ performs the meet operation with $\mathcal{I}(\dot{C})$.

The derived analysis $\hat{F}_s$ is an over-approximation of the selected program executions approximated by the contour.

Theorem 1 (Correctness of $\hat{F}_s$): $\alpha_m(\mathbf{lfp}F_s) \sqsubseteq \mathbf{lfp}\hat{F}_s$

Also, the interpretation of the contour $\mathcal{I}(\dot{C})$ is an over-approximation of the derived tuned analysis of the selected executions:

Theorem 2: $\mathbf{lfp}\hat{F}_s \sqsubseteq \mathcal{I}(\dot{C})$

Proofs of both theorems are straightforward.

D. Discussion

Our framework allows users to tune the analysis scalability by choosing appropriate contours. Because a derived tuned analysis performs its analysis within the boundary confined by its contour, the contour determines the scalability of the tuned analysis. Good contours trim spurious execution flows from consideration of tuned analyses leading to reducing false positives from the analysis results.

Depending on a selected contour, the derived analysis may not be scalable enough. In such cases, choosing an unsound but scalable contour that contains a strict subset of all execution flows may be an option. By choosing specific cases to focus on, tuned analyses can get benefits for the scalability while giving up the soundness. Of course, one can focus on other cases one by one or collectively by selecting different contours.

As the selection of contours explicitly tunes the analysis scalability by confining the scope of tuned analyses, it also improves the precision of tuned analyses. No matter what contours we select, the tuned analysis has less false alarms than that of the baseline analysis. At the same time, because contours trim any spurious execution flows not contained in them by the meet operation with $\mathcal{I}(\dot{C})$, the tuned analyses become more precise than the baseline analysis.

Note that the tuned analysis is more scalable than the baseline analysis because it trims the analysis target with the given contour by performing the meet operation with $\mathcal{I}(\dot{C})$. In addition, when the abstract state domain $\hat{\mathbb{S}}$ is a function cpo (complete partial order) $\hat{\mathbb{L}} \to \hat{\mathbb{V}}$ as we discussed in Section III-A, we can improve the analysis scalability further in terms of the analysis performance by applying the sparse analysis framework [9], [8], [10]. We describe how we apply it in our implementation of the framework in Section V.

IV. Instances

To demonstrate how to use the general framework in practice, this section shows two instances, SWP and SWF. They both use the SAFE analysis [11], [12] as a baseline analysis, and SWP and SWF use WALA's propagation-based analysis (PB) [15], [16], [17] and WALA's field-based analysis (FB) [18] as pre-analyses that construct contours, respectively.

A. Instance: SWP

Selected Collecting Semantics. Because PB is a sound analysis, using PB as a pre-analysis effectively means that SWP uses all the possible execution flows as selected executions. Thus, the selected collecting semantics $\langle \mathbb{C}, \wp(\mathbb{S}), f_P, \to_\phi \rangle$ for some f_P is the same as the collecting semantics in Section III-A.

Abstraction for Selected Collecting Semantics. Since PB is a flow-insensitive analysis, which does not consider control points during analysis, it abstracts a set of concrete states for each control point to a singleton abstract state:

$$\mathbb{C} \to \wp(\mathbb{S}) \underset{\alpha_s}{\overset{\gamma_s}{\leftrightarrows}} \hat{\mathbb{S}} \qquad \begin{array}{rcl} \alpha_s(\phi) & = & \bigsqcup_{c \in \mathbb{C}} \ \alpha_{\mathbb{S}}(\phi(c)) \\ \gamma_s(\hat{\sigma}) & = & \lambda c \in \mathbb{C}.\ \gamma_{\mathbb{S}}(\hat{\sigma}) \ . \end{array}$$

Then, a contour $\dot{\mathcal{C}}_{\mathsf{WP}} \in \hat{\mathbb{S}}$ is the least fixpoint of the following semantic function $\hat{F}_P \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$:

$$\hat{F}_P(\hat{\sigma}) = \bigsqcup_{c \in \mathbb{C}} \hat{f}_P(c)(\hat{\sigma})$$

where $\hat{f}_P \in \mathbb{C} \to \hat{\mathbb{S}} \to \hat{\mathbb{S}}$ is an abstract counter part of f_P. We assume that the abstract semantic function $\hat{f}_P(c)$ is monotonic designed in the abstract interpretation framework. Therefore, the following condition holds:

$$\alpha_s(\mathbf{lfp}\ F_P) \sqsubseteq \mathbf{lfp}\ \hat{F}_P.$$

Since we use the same abstraction for abstract states, the definition of $\mathcal{I}_P$ is simply as follows:

$$\mathcal{I}_P(\hat{\sigma}) = \lambda c \in \mathbb{C}.\ \hat{\sigma}$$

which satisfies the condition $\alpha_m \circ \gamma_s \sqsubseteq \mathcal{I}_P$. For a given abstract state, it produces a function that returns the abstract state for any control point. Then, we can design a tuned analysis $\hat{F}_{SWP}$ for the contour $\dot{\mathcal{C}}_{\mathsf{WP}}$ as follows:

$$\hat{F}_{SWP}(\hat{\phi}) = \lambda c \in \mathbb{C}.\ \hat{f}(c) \left(\bigsqcup_{c' \overset{P}{\hookrightarrow}_{\hat{\phi}} c} \hat{\phi}(c') \right) \sqcap \mathcal{I}_P(\dot{\mathcal{C}}_{\mathsf{WP}})$$

where $\overset{P}{\hookrightarrow}_{\hat{\phi}} \overset{\mathsf{def}}{=} \hookrightarrow_{\hat{\phi}} \cap \hookrightarrow_{\mathcal{I}_P(\dot{\mathcal{C}}_{\mathsf{WP}})}$.

Note that the derived analysis $\hat{F}_{SWP}$ is as precise as SAFE, and because it is tuned by the contour $\dot{\mathcal{C}}_{\mathsf{WP}}$, it may be more precise than SAFE. In addition, since the contour over-approximates the collecting semantics, the tuned analysis over-approximates all the possible program executions by Theorem 1.

B. Instance: SWF

Selected Collecting Semantics. To lessen the analysis complexity, FB abandons elaborate object-sensitive analysis entirely using a single location globally for each field. Thus, we can denote the selected collecting semantics as $\langle \mathbb{C}, \wp(\mathbb{S}), f_F, \rightarrow_\phi \rangle$ for some local semantic function f_F, where $f_F(c)$ specifies the selected collecting semantics at a given control point c, which is a subset of the collecting semantics $f(c)$:

$$\forall c \in \mathbb{C}, \sigma \in \wp(\mathbb{S}).\ f_F(c)(\sigma) \subseteq f(c)(\sigma).$$

Then, we can define the selected collecting semantics as follows:

$$\begin{array}{c} F_F \in (\mathbb{C} \to \wp(\mathbb{S})) \to (\mathbb{C} \to \wp(\mathbb{S})) \\ F_F(\phi) = \lambda c \in \mathbb{C}.\ f_F(c)\left(\bigcup_{c' \to_\phi c} \phi(c')\right). \end{array}$$

Abstraction for Selected Collecting Semantics. Because FB is a flow-insensitive and object-insensitive analysis, it does not consider control points nor objects during analysis. Thus, it basically abstracts a set of concrete states for each control point to a global single abstract object $\hat{\mathbb{O}}$:

$$\mathbb{C} \to \wp(\mathbb{S}) \underset{\alpha_s}{\overset{\gamma_s}{\leftrightarrows}} \hat{\mathbb{O}} \qquad \begin{array}{rcl} \alpha_s(\phi) & = & \bigsqcup_{\hat{l} \in \hat{\mathbb{L}}} (\bigsqcup_{c \in \mathbb{C}} \ \alpha_{\mathbb{S}}(\phi(c)))(\hat{l}) \\ \gamma_s(\hat{o}) & = & \lambda c \in \mathbb{C}.\gamma_{\mathbb{S}}(\lambda \hat{l} \in \hat{\mathbb{L}}.\hat{o}) \end{array}$$

Then, a contour $\dot{\mathcal{C}}_{\mathsf{WF}} \in \hat{\mathbb{O}}$ is the least fixpoint of the following semantic function $\hat{F}_F \in \hat{\mathbb{O}} \to \hat{\mathbb{O}}$:

$$\hat{F}_F(\hat{o}) = \bigsqcup_{c \in \mathbb{C}} \hat{f}_F(c)(\hat{o})$$

where $\hat{f}_F(c)$ is an abstract counter part of $f_F(c)$. We assume that the abstract semantic function $\hat{f}_F(c)$ is monotonic designed in the abstract interpretation framework. Therefore, the following condition holds:

$$\alpha_s(\mathbf{lfp} F_F) \sqsubseteq \mathbf{lfp} \hat{F}_F.$$

In this case, we can define $\mathcal{I}_F \in \hat{\mathbb{O}} \to (\mathbb{C} \to \hat{\mathbb{S}})$ as follows:

$$\mathcal{I}_F(\hat{o}) = \lambda c \in \mathbb{C}.(\lambda \hat{l} \in \hat{\mathbb{L}}.\hat{o}),$$

and we can design a tuned analysis that subsumes the selected collecting semantics as follows:

$$\hat{F}_{SWF}(\hat{\phi}) = \lambda c \in \mathbb{C}.\ \hat{f}(c) \left(\bigsqcup_{c' \overset{F}{\hookrightarrow}_{\hat{\phi}} c} \hat{\phi}(c') \right) \sqcap \mathcal{I}_F(\dot{\mathcal{C}}_{\mathsf{WF}})$$

where $\overset{F}{\hookrightarrow}_{\hat{\phi}} \overset{\mathsf{def}}{=} \hookrightarrow_{\hat{\phi}} \cap \hookrightarrow_{\mathcal{I}_F(\dot{\mathcal{C}}_{walaq})}$.

To see the effects of the tuned analysis more clearly, let us revisit the code example presented in Section II:

```
1  var f = { x: function a() { ... },
2            y: function b() { ... } };
3  var g = { x: function c() { ... } };
4  var h = Y.mix({}, f);
5  h.x();
```

While the actual evaluation of the code calls the function `a` on line 5, the sound baseline analysis SAFE will conclude that either function `a` or `b` will be called, and an unsound FB will conclude that either function `a` or `c` will be called. However, the tuned analysis will precisely conclude that only the function `a` will be called by performing the meet operation with the contour.

C. Discussion

While both PB and FB, two pre-analyses used in the instances, are standalone static analyzers, the framework does not require a pre-analysis be a separate static analysis. Instead, one can use a collection of dynamic execution traces that contain the states during particular executions as a pre-analysis result. For such a collection of dynamic execution traces, we do not even need to abstract a set of concrete states to construct a contour. Thus, we can use the following Galois connection:

$$\mathbb{C} \to \wp(\mathbb{S}) \underset{\alpha_s}{\overset{\gamma_s}{\leftrightarrows}} \mathbb{C} \to \wp(\mathbb{S})$$

where both α_s and γ_s are the identify function. The interpretation function $\mathcal{I}$ is the same as α_m in this case.

Indeed, we can formalize the blended analysis [27] in this setting where a tuned analysis performs the baseline SAFE analysis on only extracted execution flows. Also, when a contour contains information about dynamically generated code, the corresponding tuned analysis can get benefits from the extracted code. Although we could not experiment with such a pre-analysis that collects concrete execution traces because the tool is not publicly available, we plan to pursue this direction.

V. Implementation

This section describes main technical challenges in integrating two independently developed analyzers and deriving sparse analyses for JavaScript programs, and explains how we solve them.

A. Interpretation of WALA Analysis Results in SAFE

We implemented the framework using SAFE as the baseline analysis and two versions of WALA analyses PB and FB as pre-analyses. The analysis results of PB and FB correspond to their contours $\dot{\mathcal{C}}_{\mathsf{WP}}$ and $\dot{\mathcal{C}}_{\mathsf{WF}}$, respectively. To use the contours in tuned analyses, we develop concrete versions of the interpretation function $\mathcal{I}_P$ and $\mathcal{I}_F$ for PB and FB, respectively. In other words, to use the pre-analysis results in the SAFE analysis, we should transform program states in WALA to program states in SAFE, which requires various mappings between WALA domains and SAFE domains. For example, we should transform an abstract location in WALA to that in SAFE, and an abstract value in WALA to that in SAFE. Here are some representative transformation due to the differences between WALA and SAFE in the implementation of JavaScript static analysis.

User objects. For user-defined objects generated by function declarations, array literals, object literals, or object construction, we use their source locations to map the same user objects in WALA and SAFE. For built-in functions that do not have source locations of their definition sites, we use their names.

Lexical environments. JavaScript maintains variable information in lexical environments; each function call creates a new lexical environment to keep the information of the variables in the function. While SAFE creates a lexical environment for each function call site as the ECMAScript standard specifies, WALA creates a lexical environment for each function declaration. Thus, for each lexical environment for a function f in WALA, we map it to the lexical environments of the call sites of f in SAFE.

Arguments objects. Each JavaScript function except for the global function has an implicitly declared `arguments` object. While SAFE generates `arguments` objects in function call sites as the ECMAScript standard specifies, WALA generates them in function definition sites. Similarly for lexical environments, we map each `arguments` object for a function f in WALA to the `arguments` object of the call sites of f in SAFE.

Variables. Both WALA and SAFE deal with user-defined variables from input JavaScript programs and analyzer-defined temporary variables to keep intermediate results of complex expressions. For example, when SAFE translates a complex expression like "`x = obj.prop1.prop2`" into its own intermediate representation, it generates a temporary variable for `obj.prop1`. While it is straightforward to map user-defined variables using their source locations, mapping analyzer-defined variables is not trivial because WALA and SAFE have different strategies to create temporary variables. For temporary variables, we use source locations of their corresponding complex expressions.

Implicit type conversions. JavaScript has implicit type conversions for primitive values such as strings, numbers, and booleans. When primitive values are used as built-in objects like `String`, `Number`, and `Boolean`, they implicitly convert to corresponding built-in objects. While SAFE models implicit type conversions as the ECMAScript standard defines, WALA does not support them. Thus, we transform implicit type conversions in JavaScript applications to explicit type conversions.

B. Sparse Analysis for JavaScript Applications

To make tuned analyses even more scalable, we applied the general sparse analysis framework [9], [10] in our implementation after addressing two issues. The first issues is that while the sparse analysis framework requires a pre-analysis be a conservative approximation of a given baseline analysis, contours in our framework may not be a conservative approximation of the baseline analysis. However, because the formalization presented in Section III guarantees that contours support conservative approximation of tuned analyses by Theorem 2, we can safely apply the sparse analysis framework to tuned analyses.

The second issue is that the general sparse analysis framework is often inefficient in JavaScript analysis because rebuilding data dependencies whenever analysis results change is costly [10]. The proposed approach that postpones rebuilding data dependencies may work well for C-like languages, but it may not work for languages with prevalent exception flows like JavaScript. A reasonable solution may be an incremental algorithm that newly builds data dependencies only for parts that have been changed [28]. However, the algorithm is impractically expensive for JavaScript analysis because whenever the algorithm creates a new join (ϕ) node during analysis, it requires elimination of related edges to the join node. To address this problem of prevalent exception flows, we improved the incremental algorithm. Instead of eliminating existing edges and adding ϕ nodes on the fly as in the traditional incremental algorithm, we devised an improved algorithm that pre-computes all the possible ϕ nodes and constructs data dependencies using the pre-computed ϕ nodes. Our algorithm may create more ϕ nodes than the original incremental algorithm, but it certainly reduces the costs of data dependency construction, which is especially necessary for JavaScript analysis.

VI. Evaluation

In this section, we evaluate the proposed tunable analysis framework using four analyzers. We first describe research questions, evaluation methodology, and evaluation subjects, and we show the evaluation results.

A. Research Questions

We design our evaluation to address the following research questions:

- **RQ1. Scalability:** Given the same timeout, how many subjects does each analyzer finish analyzing? An analyzer finishing analysis of more subjects is more scalable.
- **RQ2. Precision:** For the callsites in each subject, how many called function objects does each analyzer approximate on average? An analyzer that finds fewer function objects for each function call on average is more precise.
- **RQ3. Coverage:** Given the same timeout, how many basic blocks does each analyzer finish analyzing? An analyzer finishing analysis of more basic blocks has higher coverage.
- **RQ4. Global sparse analysis:** Is the global sparse analysis framework applicable to JavaScript?

B. Evaluation Methodology and Subjects

To answer the research questions, we performed various experiments using four analyzers: 1) SAFE, a sound baseline analyzer, 2) SWP, a tuned analyzer with SAFE as a baseline analyzer and WALA's PB as a pre-analyzer 3) SWF, a tuned analyzer with SAFE as a baseline analyzer and WALA's FB as a pre-analyzer, and 4) FB, an unsound WALA's FB analysis. For all the experiments, SAFE used 5 call-context sensitivity and location cloning [29]. The pre-analyses, PB and FB, are context-insensitive, and they do not perform string analysis. In addition, PB used correlation tracking [16] for `for-in` loops. While the quality of analysis results relies on the quality of the underlying analyzers, because it is not the main focus of this paper, we refer the interested readers to relevant papers [12], [26], [16], [17].

For evaluation subjects, we used 3 categories–benchmark, library, and website–and used 5 subjects from each category. We distinguished benchmarks and websites because they have different behaviors [30], and we included libraries because they have been the major targets for JavaScript analysis [16], [17], [31]. For the benchmark category, we used RayTrace, Richards, Splay, and NavierStokes from the V8 benchmark suite version 7[6] and Box2dWeb[7]. For the library category, we used jQuery, MooTools, Prototype, YUI, and Underscore[8]. For the website category, we used `live.com`, `wikipedia.org`, `facebook.com`, `youtube.com`, and `baidu.com`. For each category, we selected subjects that show the differences between analyzers most explicitly. We conducted the experiments on a machine with 3.4GHz Intel Core i7 CPU and 32GB Memory.

[6]https://v8.googlecode.com/svn/data/benchmarks/v7/run.html
[7]http://code.google.com/p/box2dweb/
[8]http://underscorejs.org

TABLE I: Scalability of the analyzers. Entries marked ✗ denote that the analyzers do not finish analysis within the timeout of 10 minutes. The analysis time is in seconds, and parenthesized numbers (m/n) denote that a pre-analysis took m seconds and a baseline analysis took n seconds. The analysis time for FB is the same as the pre-analysis time for SWF.

	Subject (LOC)	SAFE	SWP	SWF
Benchmark	RayTrace (679)	✗	576.3 (4.8/571.5)	600.3 (2.9/597.4)
	Richards (288)	2.1	3.7 (2.4/1.3)	3.9 (2.2/1.7)
	Splay (205)	9.2	8.6 (2.1/6.5)	15.8 (1.8/14.0)
	NavierStokes (331)	1.0	4.4 (3.1/1.3)	3.2 (2.0/1.2)
	Box2dWeb (10918)	45.0	58.5 (15.6/42.9)	48.2 (45.4/2.8)
Library	jQuery (6206)	✗	✗	28.6 (23.9/4.7)
	MooTools (4022)	✗	✗	27.6 (25.7/1.9)
	Prototype (5914)	✗	✗	26.2 (24.2/2.0)
	YUI (7181)	✗	8.6 (6.9/1.7)	18.7 (13.7/5.0)
	Underscore (1065)	✗	✗	5.2 (4.6/0.6)
Website	`live.com` (6325)	240.8	✗	128.0 (32.2/95.8)
	`wikipedia.org` (291)	0.1	5.8 (5.8/0.0)	5.8 (5.8/0.0)
	`facebook.com` (5584)	✗	✗	64.1 (62.1/2.0)
	`youtube.com` (5061)	✗	✗	382.6 (247.6/135.0)
	`baidu.com` (9739)	✗	✗	119.0 (100.2/18.8)

C. Scalability

To compare the scalability of four analyzers, we show which analyzer finishes analysis of each subject in Table I with the timeout of 10 minutes. We measured the analysis time in seconds. For tuned analyzers, we show how much time a pre-analysis and a baseline analysis each took in parentheses. Because the analysis time for FB is the same as the pre-analysis time for SWF, we do not show FB in the table. While all SAFE, SWP, SWF, and FB can finish analyzing most benchmarks, they show differences in analyzing libraries and websites.

Comparing SAFE and SWP in analyzing libraries, they both cannot analyze most of them; SWP finishes analysis of the YUI library. To analyze YUI in a sound manner, SAFE analyzes many (possibly spurious) execution flows of YUI, which causes much (possibly unnecessary) analysis computation. On the contrary, SWP uses a contour produced by PB to trim unnecessary execution flows, which makes a set of execution flows to analyze smaller than SAFE. Therefore, it decreases the amount of analysis computation leading to finish analysis of YUI by SWP. Because SWP takes sound analysis results of PB to generate contours, our framework guarantees that SWP provides a sound analysis result of YUI.

For the analysis of websites, both SAFE and SWP cannot analyze most of them except for the simple `wikipedia.org`; SAFE can finish analysis of `live.com` in addition. We observed that SWP did not finish analysis of `live.com` because its pre-analysis PB did not finish analyzing it.

Not surprisingly, SWF is the most scalable one; it successfully finishes analysis of all the benchmarks, libraries, and websites within 8 minutes. Because contours derived from the FB pre-analysis are deliberately unsound, SWF can analyze large-scale programs more effectively than SAFE and SWP.

The experimental results show that the analysis scalability is tunable by contours. We believe that reasonably designed contours will enable analysis of large and sophisticated programs.

D. Precision

To evaluate how precise results each tuned analysis produces by confining analysis targets with its contour, we measured the average number of callees the analyzer estimates over the number of call sites as shown in Table II. We also present the maximum number of calls in parentheses. Note that comparing analysis precision between sound analyses SAFE and SWP and unsound analyses SWF and FB is not trivial. Although the numbers in the table do not represent the analysis precision of whole programs, it shows speculative analysis precision for the analysis targets of each analyzer.

Splay and NavierStokes in benchmarks show the same numbers for all the analyzers except for FB. Richards in benchmarks has the same precision in SAFE and SWP, and YUI in libraries shows the same precision between SWP and SWF. We confirmed that they have the same call sites and each call site has the same callees, which means that they have the same precision at each call site. For Richards, Box2dWeb, and `live.com`, SWF shows smaller numbers than SAFE and SWP due to its unsound pre-analysis, FB. As the FB column for precision in Table II suggests, FB may miss some call sites (unsound) and, at the same time, it may include unreachable call sites (imprecise) as well. Because FB ignores property accesses with imprecise property names, it may miss actual execution flows. Also, because FB is *context-insensitive*, it over-approximates functions to be called at call sites. Thus, numbers of call sites analyzed by FB are smaller than those by SWF for some cases (RayTrace, Richards, Splay, and NavierStokes) but larger in most cases. However, since SWF performs the meet operation with the contours derived from FB, SWF presents the best results. Note that FB has been successfully used in finding security vulnerabilities in the AppScan product. Because the analysis results from SWF subsumes the concrete execution flows analyzed by FB, SWF can detect the same security vulnerabilities as AppScan. Moreover, because SWF produces more precise analysis results, AppScan may be able to reduce false positives by using SWF instead of FB. The experimental results show that the analysis precision is tunable by contours. When analysis results are not precise enough, one may shrink contours to achieve better analysis precision.

E. Coverage

Finally, we compare the analysis coverage of the analyzers using numbers of reachable basic blocks that they analyze. We can expect that SAFE and SWP subsume all the possible execution flows but they may not terminate. On the contrary, SWF and FB may subsume partial execution flows but, because they are more scalable, they may cover more lines of source code even when SAFE and SWP do not terminate. Table II shows that SWF and FB indeed cover more subjects. Splay and NavierStokes in benchmarks cover the same basic blocks in SAFE, SWP, and SWF. We can speculate from this result that contours generated in SWP and SWF include SAFE analysis results, so that each analysis result is sound for the whole program. Richards shows that SAFE covers more basic blocks than SWP, and SWP covers more basic blocks than SWF. Indeed, because both SAFE and SWP are sound, their analysis targets should include that of SWF. At the same time, because SWP performs the meet operation with the contours derived from PB, the analysis targets of SWP should be smaller than that of SAFE. Even though FB shows the largest coverage for all subjects, FB may miss some actual basic blocks (unsound) and, simulataneously, it may include unreachable basic blocks (imprecise) as we discussed in Section VI-D. While FB analyzes more than thousand basic blocks for most websites, SWF does not analyze that many basic blocks by the meet operation with the contours from FB. The experimental results show that generating smaller contours helps analysis of large-scale programs while giving up some analysis coverage. To improve the analysis coverage, one may enlarge contours to cover more program execution flows.

F. Sparse

As we discussed in Section V-B, SWP and SWF are instances of the global sparse analysis framework [10]. We consider SWP as an application of the global sparse analysis framework as it is. Table I shows that applying the global sparse analysis as is to JavaScript analysis helps in analyzing YUI; while SAFE did not finish analysis of YUI, SWP did. However, it does not work well in general in the sense that SWP does not outperform SAFE noticeably. We believe that a naïve application of the general sparse analysis framework to analysis of JavaScript programs, represented by SWP, is likely to be ineffective. For languages with frequent changes in control flow graphs like JavaScript, additional techniques are necessary as in SWF.

G. Threats to Validity

We identify the following threats to the validity:

- **The subjects used in the experiments may not be representative.** We selected 5 widely used subjects for 3 different categories that have different behaviors [30] each from real-world JavaScript applications. However, they may not be representative of all JavaScript programs.
- **The experimental results on precision may not represent the precision of the whole program analysis results.** Because we measured only average numbers of callees and numbers of call sites in analysis targets, the analysis precision results may not apply to whole program analysis results. Therefore, direct comparison between average numbers of callees is meaningful if analyses have the same call sites. If a contour of SWF excludes call sites that have small numbers of callees, then SWF may have larger average numbers of callees than other sound analyses, SAFE and SWP.

VII. Related Work

Because JavaScript applications are prevalent these days, researchers have proposed various techniques to statically analyze JavaScript programs. Similarly for statically typed languages like Java, most analysis techniques are sound but impractical [16], [17]. Moreover, static analysis of JavaScript

TABLE II: Precision and coverage. Each entry for precision denotes the average number of callees that the analyzer estiates (the maximum number of calls) / the number of call sites. Each entry for coverage denotes the number of reachable basic blocks that the analyzer analyzes.

	Subject	Precision				Coverage			
		SAFE	SWP	SWF	FB	SAFE	SWP	SWF	FB
Benchmark	RayTrace	✗	1.05 (3)/1425	1.05 (3)/1425	1.38 (16)/ 168	✗	813	813	1079
	Richards	1.04 (4)/ 82	1.04 (4)/ 82	1.04 (4)/ 76	1.15 (5)/ 53	327	319	303	387
	Splay	1.00 (1)/ 193	1.00 (1)/ 193	1.00 (1)/ 193	1.02 (2)/ 44	292	292	292	329
	NavierStokes	1.00 (1)/ 77	1.00 (1)/ 77	1.00 (1)/ 77	1.04 (2)/ 57	340	340	340	465
	Box2dWeb	1.03 (22)/ 698	1.03 (22)/ 698	1.00 (1)/ 111	2.90 (260)/1929	922	922	117	11728
Library	jQuery	✗	✗	1.03 (4)/ 95	14.82 (187)/1432	✗	✗	502	11847
	MooTools	✗	✗	1.00 (1)/ 36	17.06 (207)/1290	✗	✗	134	9432
	Prototype	✗	✗	1.00 (1)/ 32	4.10 (159)/1513	✗	✗	248	10881
	YUI	✗	1.11 (6)/ 47	1.11 (6)/ 47	3.12 (73)/ 864	✗	324	312	8036
	Underscore	✗	✗	1.00 (1)/ 28	3.45 (36)/ 194	✗	✗	137	1766
Website	`live.com`	1.02 (16)/ 884	✗	1.00 (1)/ 766	3.55 (186)/2382	411	✗	336	11741
	`wikipedia.org`	0.00 (0)/ 0	0.00 (0)/ 0	0.00 (0)/ 0	1.21 (4)/ 75	6	6	6	647
	`facebook.com`	✗	✗	1.00 (1)/ 12	5.93 (231)/1282	✗	✗	132	9546
	`youtube.com`	✗	✗	1.00 (1)/ 107	10.85 (215)/2137	✗	✗	532	15028
	`baidu.com`	✗	✗	1.04 (6)/ 125	20.54 (443)/3872	✗	✗	674	27729

is much more impractical than static analysis of Java due to its extremely dynamic and functional features. Some analysis like correlation tracking [16] keeps track of relationships between object properties, but it applies to only special patterns. Some sound analysis like dynamic determinacy [17] utilizes determinate values at compile time, but it loses the precision gain by determinate values as soon as it reaches indeterminate values, which is not scalable enough.

Instead, recent studies have reported unsound analyses of JavaScript programs. Because bug detection does not need to guarantee absence of bugs, bug detection may not need to use sound analysis. Similarly, a simple taint analysis may not require sound analysis; it may be enough to analyze unsound call graphs of target programs as long as the unsound analysis is scalable enough to finish analyzing large-scale JavaScript applications in the wild [18]. Once unsound analyses are acceptable, various combinations of dynamic and static analyses would be possible by performing such analyses on only extracted execution flows [27].

Building on top of the static analysis techniques for JavaScript programs, researchers have developed open-source JavaScript static analyzers. SAFE [11] is an abstract interpretation based static analysis framework. It supports a rich set of modeled libraries, DOM APIs, and platform APIs for analyzing real-world JavaScript web applications [12]. It also provides various analysis techniques like multiple sensitivities and loop-sensitive analysis [26]. Like SAFE, TAJS [13] supports a static analysis in the abstract interpretation framework. It provides several analysis techniques to improve the precision and scalability of JavaScript analysis [32], [33], [31], [25], but it is not yet scalable for analyzing libraries like Mootols and Prototype. WALA [15] is originally developed for Java program analysis, and now it also provides JavaScript program analysis. Using existing analysis supports for Java, it equips with a variety of tools for JavaScript analysis including a sound propagation-based pointer analysis. However, due to the significant differences between Java and JavaScript, WALA provides another unsound field-based analysis [18] for JavaScript, which has been successfully used in academia [27], and applied to a commercial product IBM Security AppScan [19]. JSAI [34] is a JavaScript analysis framework with configurable sensitivities. While JSAI supports sound analyses, our approach handles even unsound analyses.

For large-scale C programs, several scalable analyses are available. A sparse analysis framework for C-like programs [8] has shown very effective in making existing sound static analyses scalable. Its extension to a general sparse analysis framework [10] proposes a mechanism to apply it to other languages like JavaScript. However, unfortunately, our experiments using the propagation-based WALA analysis as its "simple flow-insensitive pre-analysis" to make the SAFE analysis scalable showed that the framework as it is may not be practically applicable to analysis of JavaScript programs.

VIII. Conclusion

We present a general framework that enables designers of analyses to tune the analysis scalability and precision. Given that the extremely dynamic and functional features of large-scale JavaScript applications make scalable static analysis almost impossible, designers may get more benefits from experimenting with the analysis scalability and precision than pursuing for the ultimate sound and scalable static analysis. Our framework is parameterized by a sound baseline analysis that analyzes a given program elaborately and a pre-analysis that specifies only a subset of execution flows; designers can tune the scalability and precision of the resulting analysis by changing the pre-analysis. Our framework is theoretically well founded in the abstract interpretation framework and practically evaluated with state-of-the-art JavaScript analyzers. We formally presented stepwise instructions to use our framework, and exemplified differences resulted from different choices of pre-analyses. Using open-source JavaScript analyzers, SAFE and WALA, we implemented two instances of the framework and showed effectiveness of the framework.

Acknowledgment

This work is supported in part by National Research Foundation of Korea (Grant NRF-2014R1A2A2A01003235), Samsung Electronics, and Google Faculty Research Award.

REFERENCES

[1] "100 JavaScript online games," http://www.lutanho.net/stroke/online.html.

[2] "List of languages that compile to JS," https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js.

[3] "Node OS," https://node-os.com.

[4] "Runtime.JS," http://runtimejs.org.

[5] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[6] K. Yi and S. Ryu, "A cost-effective estimation of uncaught exceptions in Standard ML programs," *Theoretical Computer Science*, vol. 277, no. 1-2, pp. 185–217, 2002.

[7] X. Leroy and F. Pessaux, "Type-based analysis of uncaught exceptions," *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 2, pp. 340–377, Mar. 2000.

[8] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi, "Design and implementation of sparse global analyses for C-like languages," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012, pp. 229–238.

[9] Y. Ko, K. Heo, and H. Oh, "A sparse evaluation technique for detailed semantic analyses," *Computer Languages, Systems & Structures*, 2014.

[10] H. Oh, K. Heo, W. Lee, W. Lee, D. Park, J. Kang, and K. Yi, "Global sparse analysis framework," *ACM Transactions on Programming Languages and Systems*, vol. 36, no. 3, 2014.

[11] KAIST PLRG, "SAFE: JavaScript analysis framework," http://safe.kaist.ac.kr, 2013.

[12] S. Bae, H. Cho, I. Lim, and S. Ryu, "$\mathsf{SAFE}_{\mathsf{WAPI}}$: Web API misuse detector for web applications," in *Proceedings of the ACM SIGSOFT Symposium and the European Conference on Foundations of Software Engineering*, 2014.

[13] A. Møller, S. H. Jensen, P. Thiemann, M. Madsen, M. D. Ingesman, P. Jonsson, and E. Andreasen, "TAJS: Type analyzer for JavaScript," https://github.com/cs-au-dk/TAJS, 2014.

[14] M. Madsen and A. Møller, "Sparse dataflow analysis with pointers and reachability," in *Proc. 21st International Static Analysis Symposium (SAS)*, 2014.

[15] IBM Research, "T.J. Watson Libraries for Analysis (WALA)," http://wala.sf.net.

[16] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip, "Correlation tracking for points-to analysis of JavaScript," in *Proceedings of the European Conference on Object-Oriented Programming*, 2012.

[17] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Dynamic determinacy analysis," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[18] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for JavaScript IDE services," in *Proceedings of the International Conference on Software Engineering*, 2013.

[19] "IBM Security AppScan," http://www-03.ibm.com/software/products/en/appscan.

[20] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: A manifesto," *Communications of the ACM*, pp. 44–46, 2015.

[21] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1977, pp. 238–252.

[22] ——, "Systematic design of program analysis frameworks," in *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1979, pp. 269–282.

[23] "Implementation of practically tunable static analysis framework for large-scale JavaScript applications," Omitted for anonymizing authors, 2015.

[24] G. Bracha and W. Cook, "Mixin-based inheritance," in *Proceedings of the European Conference on Object-oriented Programming / Object-oriented Programming Systems, Languages, and Applications*, 1990, pp. 303–311.

[25] E. Andreasen and A. Møller, "Determinacy in static analysis for jQuery," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2014.

[26] C. Park and S. Ryu, "Scalable and precise static analysis of JavaScript applications via loop-sensitivity," in *Proceedings of the European Conference on Object-Oriented Programming*, 2015.

[27] S. Wei and B. G. Ryder, "Practical blended taint analysis for JavaScript," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2013, pp. 336–346.

[28] J.-D. Choi, V. Sarkar, and E. Schonberg, "Incremental computation of static single assignment form," in *Proceedings of the International Conference on Compiler Construction*, 1996.

[29] C. Lattner, A. Lenharth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[30] W. T. Cheung, S. Ryu, and S. Kim, "Development nature matters: An empirical study of code clones in javascript applications," *Empirical Software Engineering*, 2015.

[31] A. Feldthaus and A. Møller, "Checking correctness of TypeScript interfaces for JavaScript libraries," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2014.

[32] S. H. Jensen, P. A. Jonsson, and A. Møller, "Remedying the eval that men do," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2012.

[33] S. H. Jensen, M. Madsen, and A. Møller, "Modeling the HTML DOM and browser API in static analysis of JavaScript web applications," in *Proceedings of the European Conference on Foundations of Software Engineering*, 2011.

[34] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, "JSAI: A static analysis platform for JavaScript," in *FSE '14: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.