Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity

Changhee Park and Sukyoung Ryu

Department of Computer Science, KAIST, Daejeon, Republic of Korea
{changhee.park,sryu.cs}@kaist.ac.kr

Abstract

The numbers and sizes of JavaScript applications are ever growing but static analysis techniques for analyzing large-scale JavaScript applications are not yet ready in a *scalable* and *precise* manner. Even when building complex software like compilers and operating systems in JavaScript, developers do not get much benefits from existing static analyzers, which suffer from mutually intermingled problems of scalability and imprecision.

In this paper, we present Loop-Sensitive Analysis (LSA) that improves the analysis scalability by enhancing the analysis precision in loops. LSA distinguishes loop iterations as many as needed by automatically choosing loop unrolling numbers during analysis. We formalize LSA in the abstract interpretation framework and prove its soundness and precision theorems using Coq. We evaluate our implementation of LSA using the analysis results of main web pages in the 5 most popular websites and those of the programs that use top 5 JavaScript libraries, and show that it outperforms the state-of-the-art JavaScript static analyzers in terms of analysis scalability. Our mechanization and implementation of LSA are both publicly available.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages

Keywords and phrases JavaScript, static analysis, loops

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.735

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.1.1.12

1 Introduction

The popularity of JavaScript has extended its application areas beyond simple scripts, but analyzing JavaScript applications statically is still a challenging problem. While web application developers use JavaScript to build large-scale software including games, compilers, and even operating systems, tool supports for developing them are still in a primitive stage compared to those for statically typed languages such as C and Java. Building development tools that aid programmers understand and debug programs often requires scalable and precise static analysis techniques, but extremely functional and dynamic features of JavaScript make static analysis impractical. For a function call "$o[e]()$", for example, because JavaScript provides first-class functions and dynamic property accesses in objects, statically estimating possible values of e often leads to an imprecise result, which in turn results in many false-positive function calls. Imprecise analysis produces false execution flows to analyze incurring much performance overhead, which makes analysis results even more imprecise.

Researchers have proposed various techniques to improve analysis precision for JavaScript web applications such as specializing specific programming patterns [23], using run-time information for determinate values [20], and combining multiple heuristic specialization methods [1]. They show that improving analysis precision significantly improves analysis

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 735–756

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

performance as well. However, their static analyzers are not yet scalable enough to analyze simple applications using major libraries while they do not have any soundness proofs. Simply loading one of major libraries such as jQuery[1], Mootools[2], and Prototype[3] involves many dynamic features of JavaScript, and thus existing static analysis techniques suffer from the scalability problem due to imprecise analysis results.

In this paper, we present a novel analysis technique, *Loop-Sensitive Analysis* (LSA), which dramatically improves the analysis scalability of JavaScript applications by enhancing the analysis precision in loops. LSA distinguishes each iteration of loops with determinate loop conditions using loop contexts during analysis. It may look similar to the traditional loop unrolling but it selectively unrolls loops with determinate loop conditions and it decides the unrolling number for each loop differently and automatically during analysis. While existing loop specialization techniques [1, 23] are applicable to only special forms of loops, LSA is applicable to any forms of loops.

We formalize LSA in the abstract interpretation framework [7, 8], prove its soundness and precision theorems, and evaluate its implementation with top 5 JavaScript libraries and main web pages in the 5 most popular websites. While loop-sensitivity can be used with any form of context-sensitivity, we formally present LSA as an extension of k-CFA (Control Flow Analysis) [21] to show their relationship rigorously for language-independent programs represented by Control Flow Graphs (CFGs). We prove that LSA is sound if its base k-CFA is sound and that LSA provides more precise than or at least as precise as the analysis results of k-CFA using the proof assistant tool Coq [5]; the mechanized proofs are publicly available [12]. We implement LSA on top of an open-source JavaScript analysis framework, SAFE [13, 15]. The LSA implementation demonstrates that LSA significantly improves the analysis scalability and precision so that it can analyze all versions of jQuery, the most widely used JavaScript library, 4 of top 5 libraries, and 3 of the 5 main web pages of the most popular websites in a reasonable practical time, which outperforms the state-of-the-art JavaScript static analyzers, TAJS [1] and WALA [20], in terms of scalability.

The contributions of this paper are as follows:

- We present a novel analysis technique, LSA, which improves analysis *precision* by distinguishing each iteration of loops as many as needed during analysis. The technique is language independent and it is applicable to analysis of programs in other languages than JavaScript.
- We *formalize* LSA in the abstract interpretation framework and show how to extend k-CFA to use the technique. The formalization specifies the technical details of LSA and it is usable for formal proofs and verification.
- We provide *mechanized proofs* of the soundness and precision of LSA using the proof assistant tool, Coq [12].
- We make an LSA *implementation publicly available*; the artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). Using the implementation, we demonstrate that LSA outperforms the state-of-the-art JavaScript static analyzers in analyzing top 5 JavaScript libraries and the main web pages of the 5 most popular websites in a *scalable* way by improving analysis precision.

[1] `http://jquery.com`
[2] `http://mootools.net`
[3] `http://prototypejs.org`

```
1  jQuery.extend = function() {
2     var options, name, copy, ...
3         target = arguments[0] ...
4         i=1, length = arguments.length ...
5     if(i === length) {
6        target = this;
7        i--;
8     } ...
9     for(; i < length; i ++) { ...
10        options = arguments[i] ...
11        for (name in options) { ...
12           copy = options[name] ...
13           target[name] = copy ...
14        } ...
15     }
16 }
17 jQuery.extend({expendo: ... ,
18                    each: ... });
19 jQuery.each(...);
```

Figure 1 An excerpt from jQuery 2.1.0.

The rest of the paper is organized as follows. In Section 2, we provide a high-level idea of LSA using a motivating example. Sections 3 and 4 formally describe the concrete collecting semantics of our target programs and k-CFA, respectively. In Section 5, we present LSA as an extension of k-CFA and show its soundness and precision theorems. We evaluate a prototype implementation of LSA in terms of scalability and precision in Section 6, discuss related work in Section 7, and conclude in Section 8.

2 Motivation

In this section, after showing a running example that represents typical code patterns in JavaScript applications, we describe the scalability and precision problems in statically analyzing them. Then, we present a high-level idea of our solution, LSA.

Figure 1 shows our running example, an excerpt from the latest jQuery 2.1.0 library omitting irrelevant parts for presentation brevity. It first defines the method `jQuery.extend` (line 1) and calls it (line 17) with one object argument that has 25 properties: 4 fields and 21 methods including the method `each`. At this call site, the `extend` method extends the `jQuery` object with new properties in the argument object. Note that, in JavaScript, the value of the `arguments` object in a callee is an array object including the arguments passed by a caller [9]. In this case, because `arguments.length` is 1, the value of `target` becomes the value of `this`, the `jQuery` object, (line 6) and `i` becomes 0 (line 7) before getting into the `for` loops. Then, the subsequent loops extend the `jQuery` object by copying all the properties in the argument object one by one, and the subsequent call of `jQuery.each` (line 19) calls the function added by the `jQuery.extend` function.

While jQuery is the most widely used JavaScript library with a market share of more

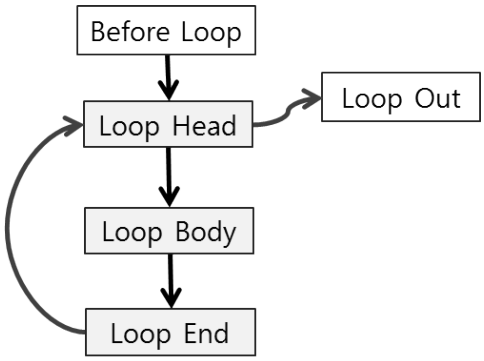

Figure 2 Control flow graph for a loop.

than 90%[4], we found that the state-of-the-art static analyzers for JavaScript applications such as SAFE [13], TAJS [18], and WALA [11] are not capable of analyzing all versions of jQuery. For example, with flow sensitivity (distinguishing different program points) [6] and call-context sensitivity (distinguishing different call sites) techniques, analysis of a simple program that just loads jQuery 2.1.0 using SAFE does not terminate in 5 hours; the analysis does not terminate either with varying sensitivities like object-sensitivity (distinguishing calls by the addresses of receiver objects) [22] and k-CFA with 1 to 10 for k (distinguishing calls by k-length call strings that represent call sequences).

We observed that this scalability problem arises due to the combination of imprecise analysis results in loops and the dynamic nature of object property names in JavaScript. Most static analyzers represent a loop as a CFG illustrated in Figure 2 and simply join the analysis results from all incoming edges into a loop-head node, which is the same as combining analysis results of all iterations. Let us revisit the code example in Figure 1. When the SAFE static analyzer analyzes the call of `jQuery.extend` on line 17, it estimates that the possible values of `options` on line 10 include the argument object passed on line 17, which approximates the possible values of `name` on line 11 as all 25 property names of the argument object. Therefore, the nested loop body on lines 12 and 13 copies the joined value of the values of 25 properties to all 25 properties in the `jQuery` object by `target[name] = copy`. Then, the analyzer estimates that the subsequent `jQuery.each` call may invoke all possible 21 functions. Because jQuery calls such methods frequently at loading time, the imprecise analysis results lead to state explosion and incur large performance overhead. We found such patterns in various applications including 9 of top 10 popular websites[5] as well as in major JavaScript libraries.

To alleviate the scalability problem, we improve analysis precision in loops by distinguishing each iteration of loops as many as needed during analysis with different loop contexts for each iteration depending on the analysis results of loop conditional expressions. It is similar to unrolling loops during analysis by finding precise unrolling counts for each loop automatically as far as the loop conditions keep definite. As for the example in Figure 1, precise unrolling counts for two loops on lines 9 and 11 should be 1 and 25, respectively. Our analysis technique indeed creates 1 and 25 different loop contexts for the loops, respectively, as long as the analysis results of loop conditional expressions are determinate for each iteration.

[4] `http://w3techs.com/technologies/overview/javascript_library/all`
[5] `http://www.alexa.com`

$$
\begin{array}{ll}
s & \in \mathbf{S} \\
n_{normal} & \in \mathbf{N_{normal}} \\
n_{call} & \in \mathbf{N_{call}} \\
n_{afterCall} & \in \mathbf{N_{afterCall}} \\
n_{entry} & \in \mathbf{N_{entry}} = \{\mathsf{n}^{\mathsf{global}}_{entry}, \ldots\} \\
n_{exit} & \in \mathbf{N_{exit}} \\
n & \in \mathbf{N} = (\mathbf{N_{normal}} \uplus \mathbf{N_{call}} \uplus \mathbf{N_{afterCall}} \uplus \mathbf{N_{entry}} \uplus \mathbf{N_{exit}}) \\
e = \langle n_1, s, n_2 \rangle & \in \mathbf{E} \subseteq \mathbf{N} \times \mathbf{S} \times \mathbf{N} \\
p & \in \mathbf{P} = (\mathbf{N} \uplus \mathbf{E}) \\
callNode & \in \mathbf{N_{afterCall}} \to \mathbf{N_{call}} \\
origin(\langle n_1, s, n_2 \rangle) & = n_1 \\
stmt(\langle n_1, s, n_2 \rangle) & = s \\
target(\langle n_1, s, n_2 \rangle) & = n_2 \\
callEdge(\langle n_1, s, n_2 \rangle) & = n_1 \in \mathbf{N_{call}} \ \wedge \ n_2 \in \mathbf{N_{entry}} \\
returnEdge(\langle n_1, s, n_2 \rangle) & = n_1 \in \mathbf{N_{exit}} \ \wedge \ n_2 \in \mathbf{N_{afterCall}} \\
normalEdge(\langle n_1, s, n_2 \rangle) & = \neg callEdge(\langle n_1, s, n_2 \rangle) \ \wedge \ \neg returnEdge(\langle n_1, s, n_2 \rangle)
\end{array}
$$

Figure 3 Program representation as a CFG.

Then, it propagates analysis results only in the loop contexts where the analysis results of the loop conditional expressions are not true to outside of loops. In this way, we can analyze loop iterations more precisely, which enables to analyze the value of `jQuery.each` as exactly the function on line 18 and consequently to analyze the following `jQuery.each` call on line 19 more precisely. In Section 6, we demonstrate that this technique can dramatically improve the analysis scalability.

3 Collecting Semantics of Programs

In this section, we formally describe the concrete collecting semantics of programs. We represent a program as a language-independent Control Flow Graph (CFG) and define its collecting semantics as a map from each *program point* (a CFG node or a CFG edge) to a set of all reachable concrete states at the point. Our formalization uses notations from Mangal *et al.*'s [17].

3.1 Program Representation

We represent a program as an interprocedural CFG $G = (\mathbf{S}, \mathbf{N}, \mathbf{E}, \mathsf{n}^{\mathsf{global}}_{entry})$ as summarized in Figure 3; $\mathbf{S}$ is a set of statements in the program; $\mathbf{N}$ is a set of nodes consisting of normal nodes $\mathbf{N_{normal}}$, call nodes $\mathbf{N_{call}}$, after-call nodes $\mathbf{N_{afterCall}}$, entry nodes $\mathbf{N_{entry}}$, and exit nodes $\mathbf{N_{exit}}$; $\mathbf{E}$ is a set of edges where an edge is a 3-tuple of a node, a statement, and another node; $\mathsf{n}^{\mathsf{global}}_{entry} \in \mathbf{N_{entry}}$ is a special entry node for the program. We use $\uplus$ to denote a disjoint union of sets.

We assume that CFGs are well-formed: a CFG has an entry node and an exit node for each function; it also has a call edge from a call node to an entry node and its corresponding return edge from an exit node to an after-call node for each function call. The function *callNode* takes an after-call node and returns its corresponding call node. The helper functions

$$
\begin{array}{lrcl}
\text{(set of states)} & \tau & \in & \wp(\mathbf{State}) = \{\tau_{\mathsf{init}}, ...\} \\
\text{(summary map)} & \kappa_c & \in & \mathbf{A_{con}} = \mathbf{P} \to \wp(\mathbf{State}) \\
& [\![s]\!]_c & \in & \wp(\mathbf{State}) \to \wp(\mathbf{State}) \\
& F_{con}(\kappa_c)(n) & = & \bigcup \{\kappa_c(e) \mid n = target(e)\} \\
& F_{con}(\kappa_c)(e) & = & [\![stmt(e)]\!]_c(\kappa_c(origin(e)))
\end{array}
$$

Figure 4 Concrete domain and transfer functions.

origin, *stmt*, and *target* are accessors of an edge, and *callEdge*, *returnEdge*, and *normalEdge* are predicates that identify the kind of an edge.

3.2 Concrete Domain and Transfer Functions

Figure 4 shows the concrete domain and transfer functions of our collecting semantics. The concrete domain is a powerset of concrete states $\wp(\mathbf{State})$, where τ_{init} is a set of initial states. The collecting semantics repeatedly updates a summary map κ_c from program points $\mathbf{P}$ to sets of states using the transfer function F_{con}. For each node n, the transfer function collects the concrete states of all the incoming edges of n. For each edge e, the transfer function performs a concrete execution of the statement $stmt(e)$ using the statement transfer function $[\![\bullet]\!]_c$ on each state at the origin node of e and returns resulting states.

We assume that the set domain of concrete states is a finite complete lattice with the subset relation ordering. Then, the domain $\mathbf{A_{con}}$ of summary maps is also a complete lattice using the following order relation:

$$\forall \kappa_{c_1}, \kappa_{c_2} \in \mathbf{A_{con}} : (\kappa_{c_1} \sqsubseteq \kappa_{c_2} \iff \forall p : \kappa_{c_1}(p) \sqsubseteq \kappa_{c_2}(p)).$$

We also assume that the statement transfer function is monotone:

$$\forall s, \tau_1, \tau_2 : \tau_1 \sqsubseteq \tau_2 \implies [\![s]\!]_c(\tau_1) \sqsubseteq [\![s]\!]_c(\tau_2).$$

Then, the final collecting semantics of a program is defined as a least fixpoint of F_{con} as follows:

$$
\begin{array}{l}
\kappa_{con} = \mathsf{leastFix}\ \lambda\kappa_c.(\kappa_I \sqcup F_{con}(\kappa_c)) \\
\text{where } \kappa_I(p) = \text{if } p = \mathsf{n}^{\mathsf{global}}_{entry} \text{ then } \tau_{\mathsf{init}} \text{ else } \bot.
\end{array}
$$

An initial summary map κ_I maps only the global entry node to the set of initial states τ_{init} and others to $\bot$ (empty set). We can easily prove that F_{con} is monotone on summary maps :

$$\forall \kappa_{c_1}, \kappa_{c_2} : \kappa_{c_1} \sqsubseteq \kappa_{c_2} \implies F_{con}(\kappa_{c_1}) \sqsubseteq F_{con}(\kappa_{c_2})$$

and the unique least fixpoint exists by the Tarski theorem [24].

4 k-CFA Formalization

Before describing our LSA in the next section, we formalize k-CFA with the program representation of Figure 3 in this section.

4.1 Abstract Interpretation

Abstract interpretation [7, 8] is a theoretical foundation of various static analyses; it guarantees the soundness of an analysis when the analysis satisfies some required conditions. In abstract interpretation, a program analysis is a computation of monotone transfer functions on an abstract domain $\hat{D}$, which represents an approximation of a concrete domain D on which concrete programs execute. When a pair of functions α and γ satisfies the following Galois connection condition:

$$\forall x \in D, \hat{x} \in \hat{D} : \alpha(x) \sqsubseteq \hat{x} \iff x \sqsubseteq \gamma(\hat{x})$$

they provide relationships between elements in the domains: an abstraction of a concrete value x is $\alpha(x)$, and concrete values denoted by an abstract value $\hat{x}$ is $\gamma(\hat{x})$. With this condition, a concrete and monotone transfer function F and its corresponding abstract transfer function $\hat{F}$ should satisfy the following:

$$\alpha \circ F \sqsubseteq \hat{F} \circ \alpha$$

where $\circ$ denotes function composition. Once an analysis meets all the conditions above, abstract interpretation guarantees that a concrete program execution result by a least fixpoint of F and its abstract program execution result by a least fixpoint of $\hat{F}$ satisfy the following soundness property:

$$\alpha \circ \mathsf{leastFix}\ F \ \sqsubseteq\ \mathsf{leastFix}\ \hat{F}$$

which means that the analysis result soundly approximates the concrete program result.

4.2 Formal Description of k-CFA

k-CFA [21] is a call-context sensitive analysis; it distinguishes the same function body from its different call sites using the k number of call strings that represent call history. As a simple example, consider the following function calls:

```
function g() { ... }
function f() { g(); }
f();
f();
```

In 0-CFA, two calls for `f` are indistinguishable; two input states from the calls are joined at the entry of the `f` body. On the contrary, 1-CFA can distinguish the two `f` calls giving more precise analysis results for `f` than the ones from 0-CFA, but it still cannot distinguish the `g` calls at the first and the second `f` calls since it maintains only one length of call strings; likewise, 2-CFA can distinguish the `g` calls but any function calls in a deeper level. In general, k-CFA provides more precise results with the longer length of k at the expense of performance overhead due to more call contexts.

4.2.1 Analysis Domain and Transfer Functions

Figure 5 shows an analysis domain and statement transfer functions for k-CFA. An abstract domain $\hat{\textbf{State}}$ is a finite complete lattice representing a set of abstract states where the state $\hat{\tau}_{\mathsf{init}}$ denotes an initial state when the analysis begins. The statement transfer function $[\![s]\!]$ is monotone on $\hat{\textbf{State}}$:

$$
\begin{array}{lll}
\text{(abstract state)} & \hat{\tau} & \in \hat{\mathbf{State}} = \{\hat{\tau}_{\mathsf{init}}, ...\} \\
\text{(lattice operations)} & \sqcup, \sqcap & \in \wp(\hat{\mathbf{State}}) \to \hat{\mathbf{State}} \\
 & \bot, \top & \in \hat{\mathbf{State}} \\
 & \sqsubseteq & \subseteq \hat{\mathbf{State}} \times \hat{\mathbf{State}} \\
\text{(transfer functions)} & [\![s]\!] & \in \hat{\mathbf{State}} \to \hat{\mathbf{State}} \\
\text{(k-length call string)} & \pi & \in \mathbf{\Pi} = \{\epsilon\} \uplus \biguplus_{k \geq n \geq 1} (\mathbf{N_{call}})^n \\
\text{(k-CFA annotation)} & \hat{\kappa}_c & \in \hat{\mathbf{A}}_{\mathsf{cfa}} = (\mathbf{P} \times \mathbf{\Pi}) \to \hat{\mathbf{State}}
\end{array}
$$

(sequence operations)

$$
n \oplus \pi = n\,\pi
$$

$$
\pi \# k = \begin{cases} \epsilon & \text{if } k = 0 \\ \pi & \text{if } k > 0 \ \wedge \ |\pi| \leq k \\ n_1 \cdots n_k & \text{if } k > 0 \ \wedge \ \pi = n_1 \cdots n_k\, n_{k+1} \cdots \end{cases}
$$

Figure 5 k-CFA domain and statement transfer functions.

$$
\hat{F}_{\mathsf{cfa}}(\hat{\kappa}_c)(\langle n, \pi \rangle) = \bigsqcup \{\hat{\kappa}_c(\langle e, \pi \rangle) \mid n = \mathit{target}(e)\}
$$

$$
\hat{F}_{\mathsf{cfa}}(\hat{\kappa}_c)(\langle e, \pi \rangle) = \begin{cases} [\![\mathit{stmt}(e)]\!](\hat{\kappa}_c(\langle \mathit{origin}(e), \pi \rangle)) & \text{if } \mathit{normalEdge}(e) \\ \bigsqcup \{[\![\mathit{stmt}(e)]\!](\hat{\kappa}_c(\langle \mathit{origin}(e), \pi_1 \rangle)) \mid \pi = (\mathit{origin}(e) \oplus \pi_1)\#k\} & \text{if } \mathit{callEdge}(e) \\ [\![\mathit{stmt}(e)]\!](\hat{\kappa}_c(\langle \mathit{origin}(e), (\mathit{callNode}(\mathit{target}(e)) \oplus \pi)\#k \rangle)) & \text{if } \mathit{returnEdge}(e) \end{cases}
$$

Figure 6 Transfer functions on k-CFA annotations.

$$
\forall s, \tau_1, \tau_2 : \tau_1 \sqsubseteq \tau_2 \Longrightarrow [\![s]\!](\tau_1) \sqsubseteq [\![s]\!](\tau_2)
$$

and it abstractly executes the statement s with an input state and produces an output state. A k-length call string π is a sequence of call nodes with the maximum length k and ϵ denotes the empty sequence. We write $n \oplus \pi$ to denote prepending a new call node n to a sequence π and $\pi \# k$ to denote the k-length prefix of π. Finally, the k-CFA annotation $\hat{\kappa_c}$ maps a pair of a node or an edge and a call string to its corresponding abstract state: $\hat{\kappa_c}(\langle p, \pi \rangle)$ gives an approximate input state at the program point p in the call context represented by the call string π. The domain $\hat{\mathbf{A}}_{\mathsf{cfa}}$ of k-CFA annotations is a complete lattice using the following order relation:

$$
\forall \hat{\kappa}_{c_1}, \hat{\kappa}_{c_2} : (\hat{\kappa}_{c_1} \sqsubseteq \hat{\kappa}_{c_2} \Longleftrightarrow \forall p, \pi : \hat{\kappa}_{c_1}(\langle p, \pi \rangle) \sqsubseteq \hat{\kappa}_{c_2}(\langle p, \pi \rangle)).
$$

4.2.2 Control Flow Analysis

Figure 6 shows the definition of the transfer function $\hat{F}_{\mathsf{cfa}}$ on k-CFA annotations. The transfer function takes a k-CFA annotation and returns an updated k-CFA annotation by transferring the analyzed state of a preceding node or edge to the current node or edge; for call and return edges, it updates call strings appropriately. Note that while a join operator in the transfer function for call edges is necessary to join all analysis results from calls with the

same call strings but different call history at the entry of the target function, the operator is not needed in the transfer function for return edges because analysis results from different call history are already combined at the entry of the function by the corresponding call edges. Then, k-CFA computes the fixpoint of $\hat{F}_{\mathsf{cfa}}$, $\hat{\kappa}_{\mathsf{cfa}}$, which contains final analysis results for all nodes and edges in a CFG with the maximum k-length call contexts:

$$\hat{\kappa}_{\mathsf{cfa}} = \mathsf{leastFix}\ \lambda\hat{\kappa}_c.(\hat{\kappa}_I \sqcup \hat{F}_{\mathsf{cfa}}(\hat{\kappa}_c))$$
$$\text{where } \hat{\kappa}_I(\langle p, \pi\rangle) = \text{if } \langle p, \pi\rangle = \langle \mathsf{n}^{\mathsf{global}}_{entry}, \epsilon\rangle \text{ then } \hat{\tau}_{\mathsf{init}} \text{ else } \bot.$$

The analysis begins with the initial state $\hat{\tau}_{\mathsf{init}}$ at the global entry node in the empty call context, and it propagates the abstract state from the initial node to all reachable nodes and edges via $\hat{F}_{\mathsf{cfa}}$. It is obvious from the order relation of $\hat{A}_{\mathsf{cfa}}$ that $\hat{F}_{\mathsf{cfa}}$ is monotone on k-CFA annotations:

$$\forall \hat{\kappa}_{c_1}, \hat{\kappa}_{c_2} : \hat{\kappa}_{c_1} \sqsubseteq \hat{\kappa}_{c_2} \Longrightarrow \hat{F}_{\mathsf{cfa}}(\hat{\kappa}_{c_1}) \sqsubseteq \hat{F}_{\mathsf{cfa}}(\hat{\kappa}_{c_2}).$$

Then, a unique least fixpoint of $\hat{F}_{\mathsf{cfa}}$ in a complete lattice domain exists [24] and the computation terminates since the domains of $\hat{F}_{\mathsf{cfa}}$ are all finite.

4.3 Soundness

We assume that k-CFA is a sound approximation of the collecting semantics presented in Section 3:

- Galois connection: Functions $\alpha_s \in \wp(\mathbf{State}) \to \hat{\mathbf{State}}$ and $\gamma_s \in \hat{\mathbf{State}} \to \wp(\mathbf{State})$ exist.
- $\alpha_s(\tau_{\mathsf{init}}) \sqsubseteq \hat{\tau}_{\mathsf{init}}$: $\hat{\tau}_{\mathsf{init}}$ is a sound approximation of τ_{init}.
- $\forall s \in \mathbf{S} : \alpha_s \circ [\![s]\!]_c \sqsubseteq [\![s]\!] \circ \alpha_s$: The abstract function $[\![s]\!]$ is a sound approximation of $[\![s]\!]_c$ for a statement s.
- Galois connection: Functions $\alpha_c \in \mathbf{A_{con}} \to \hat{\mathbf{A}}_{\mathbf{cfa}}$ and $\gamma_c \in \hat{\mathbf{A}}_{\mathbf{cfa}} \to \mathbf{A_{con}}$ exist.
- $\alpha_c \circ F_{con} \sqsubseteq \hat{F}_{\mathsf{cfa}} \circ \alpha_c$: The abstract function $\hat{F}_{\mathsf{cfa}}$ is a sound approximation of F_{con}.

Then, by the monotonicity of F_{con} and $\hat{F}_{\mathsf{cfa}}$, abstract interpretation guarantees the soundness of k-CFA:

$$\alpha_c(\kappa_{con}) \ \sqsubseteq\ \hat{\kappa}_{\mathsf{cfa}}.$$

We use the definitions of κ_{con} and $\hat{\kappa}_{\mathsf{cfa}}$, least fixpoints of F_{con} and $\hat{F}_{\mathsf{cfa}}$, respectively defined in previous sections for the same well-formed program p. In the next section, we use the above assumptions to show that our new analysis technique is also sound when we apply it to k-CFA.

5 Loop-Sensitive Analysis

Now, we extend k-CFA with loop-sensitivity. In k-CFA, a loop head node is simply a normal node with two incoming and two outgoing edges as shown in Figure 2. According to the node transfer function $\hat{F}_{\mathsf{cfa}}$, the abstract state of a loop head node is a join of abstract states from the two incoming edges, which effectively combines analysis results of all iterations incurring large precision losses. The main idea of LSA is to distinguish each loop iteration using *loop strings* like call strings in k-CFA to reduce precision losses. The novelty of our work lies in applying the sensitivity-based analysis technique to loops using loop strings and formally proving the soundness of the analysis in the abstract interpretation framework. Unlike the traditional loop unrolling, LSA automatically determines the unrolling number of each loop during analysis using the analysis results of loop conditional expressions, which immensely enhances the analysis performance.

$$
\begin{array}{ll}
n_{lhead} & \in \mathbf{N_{lhead}} \\
n_{lend} & \in \mathbf{N_{lend}} \\
n_{out} & \in \mathbf{N_{lout}} \\
n_{break} & \in \mathbf{N_{break}} \\
n_{continue} & \in \mathbf{N_{continue}} \\
n_{return} & \in \mathbf{N_{return}} \\
n & \in \mathbf{N} = (\mathbf{N_{normal}} \uplus \mathbf{N_{call}} \uplus \mathbf{N_{afterCall}} \uplus \mathbf{N_{entry}} \uplus \mathbf{N_{exit}} \uplus \\
 & \qquad \mathbf{N_{lhead}} \uplus \mathbf{N_{lend}} \uplus \mathbf{N_{lout}} \uplus \mathbf{N_{break}} \uplus \mathbf{N_{continue}} \uplus \mathbf{N_{return}})
\end{array}
$$

$$
\begin{array}{l}
\mathit{loopHead} \ \in \ \mathbf{N} \to \mathbf{N_{lhead}} \\
\mathit{loopHeads} \ \in \ \mathbf{N} \to \wp(\mathbf{N_{lhead}}) \\
\mathit{loopInEdge}(\langle n_1, s, n_2 \rangle) = n_1 \in \mathbf{N_{normal}} \wedge n_2 \in \mathbf{N_{lhead}} \\
\mathit{loopIterEdge}(\langle n_1, s, n_2 \rangle) = n_1 \in \mathbf{N_{lend}} \wedge n_2 \in \mathbf{N_{lhead}} \\
\mathit{loopIterEdge}(\langle n_1, s, n_2 \rangle) = n_1 \in \mathbf{N_{continue}} \wedge n_2 \in \mathbf{N_{lhead}} \\
\mathit{loopOutEdge}(\langle n_1, s, n_2 \rangle) = n_1 \in \mathbf{N_{lhead}} \wedge n_2 \in \mathbf{N_{lout}} \\
\mathit{loopBreakEdge}(\langle n_1, s, n_2 \rangle) \ = n_1 \in \mathbf{N_{break}} \wedge n_2 \in \mathbf{N_{lout}} \\
\mathit{loopReturnEdge}(\langle n_1, s, n_2 \rangle) = n_1 \in \mathbf{N_{return}} \wedge n_2 \in \mathbf{N_{exit}} \\
\mathit{normalEdge}(\langle n_1, s, n_2 \rangle) = \neg \mathit{callEdge}(\langle n_1, s, n_2 \rangle) \ \wedge \ \neg \mathit{returnEdge}(\langle n_1, s, n_2 \rangle) \wedge \\
\qquad \neg \mathit{loopInEdge}(\langle n_1, s, n_2 \rangle) \ \wedge \ \neg \mathit{loopIterEdge}(\langle n_1, s, n_2 \rangle) \wedge \\
\qquad \neg \mathit{loopOutEdge}(\langle n_1, s, n_2 \rangle) \ \wedge \ \neg \mathit{loopBreakEdge}(\langle n_1, s, n_2 \rangle) \wedge \\
\qquad \neg \mathit{loopReturnEdge}(\langle n_1, s, n_2 \rangle)
\end{array}
$$

Figure 7 Program representation with loop-sensitivity.

5.1 Formal Description

We formally describe how to extend sound k-CFA to LSA while preserving the soundness of the analysis.

5.1.1 Program Representation

Without loss of generality, we assume that we can rewrite all loop constructs in a target program as "`while(`e`)`s" where the evaluation of e does not have any side effects and CFGs for all loops conform to the structure in Figure 2. Note that for-in loops in JavaScript cannot be rewritten into while loops in general since the iteration order is implementation specific according to the ECMAScript standard [9]. However, real-world implementations in major browsers such as Internet Explorer, Safari, Firefox, and Chrome use the same iteration order; for instance, in `for(x in obj)`, they use the same order that the properties were added to the object `obj`. Thus, we use this order for the JavaScript cases. Figure 7 presents the extension of CFGs in Figure 3 for loop-sensitivity. In addition to call, after-call, entry, and exit nodes introduced in Figure 3, we refine nodes further to distinguish nodes in loops as loop-head, loop-end, loop-out, break, continue, and return nodes. The functions *loopHead* and *loopHeads* take a node in a loop and return its innermost loop-head node and a set of all its enclosing loop-head nodes, respectively. We also refine edges further to distinguish loop-related edges as follows:

- loop-in edge: an edge from a normal node outside a loop to a loop-head node
- loop-iter edge: an edge from a loop-end node or a continue node to a loop-head node
- loop-out edge: an edge from a loop-head node to a loop-out node
- loop-break edge: an edge from a break node inside a loop to a loop-out node
- loop-return edge: an edge from a return node inside a loop to an exit node

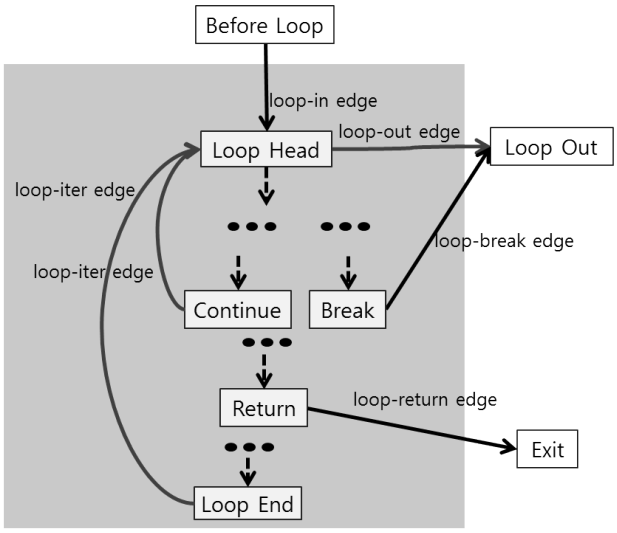

Figure 8 Loop-related nodes and edges for LSA.

(i-depth j-length loop string)
$$\psi \in \mathbf{\Psi} = \{\epsilon\} \uplus \biguplus_{i \geq n \geq 1} (\mathbf{\Pi} \times \mathbf{N_{lhead}} \times \mathbb{N}_{\{0..j\}})^n$$
(abstract value for loop conditional expression)
$$check(s, \hat{\tau}) \in \hat{\mathbf{Bool}} = \{\bot_b, \hat{\mathsf{true}}, \hat{\mathsf{false}}, \top_b\}$$
(context)
$$\phi = \langle \pi, \psi \rangle \in \mathbf{\Phi} = \mathbf{\Pi} \times \mathbf{\Psi}$$
(LSA annotation)
$$\hat{\kappa}_l \in \hat{\mathbf{A}}_{\mathsf{lsa}} = (\mathbf{P} \times \mathbf{\Phi}) \to \hat{\mathbf{State}}$$

Figure 9 LSA domain and transfer functions.

The functions *loopInEdge*, *loopIterEdge*, *loopOutEdge*, *loopBreakEdge*, and *loopReturnEdge* are predicates identifying the kind of an edge. Note that this extension does not change CFG structures but just refine the kinds of nodes and edges; all loop-related nodes and edges are normal nodes and edges in k-CFA. Figure 8 illustrates loop-related nodes and edges: a shaded box denotes a loop, dashed edges denote normal edges, and solid edges denote loop-related edges.

5.1.2 Analysis Domain and Transfer Functions

Figure 9 shows the extension of the analysis domain and transfer functions in Figure 5 for loop-sensitivity. A loop string ψ is a maximum i-length sequence of *loop contexts*; for a loop context $\langle \pi, n_{lhead}, m \rangle$, π is a call context where this loop context is introduced, n_{lhead} is a loop-head node that introduces this loop context, and m is a loop iteration count where $m \in \mathbb{N}_{\{0..j\}}$; $\mathbb{N}_{\{0..j\}}$ is the set of natural numbers between 0 and j. When loop iterations are indistinguishable because the value of the loop conditional expression may be both true and false, we call such contexts *join loop contexts* and denote them by $m = 0$. We call LSA using such "i-depth j-length loop strings" and k-length call strings $\langle i, j, k \rangle$-LSA. The values

of i and j are predefined as k in k-CFA. Intuitively, i and j mean the maximum depth of distinguishable nested loops and the maximum number of distinguishable iterations in a loop, respectively. For instance, if $i = 0$, LSA is the same as k-CFA and $\langle 2, 10, k\rangle$-LSA can distinguish up to 10 iterations in each loop with 2-level nested loops. As a design choice, we require that $j \geq 1$ while $i \geq 0$ and $k \geq 0$. Note that the value of j is effective only when $i \geq 1$.

The function *check* checks if a boolean expression s evaluates to an error ($\bot_b$), true, false, or both ($\top_b$) in a given abstract state $\hat{\tau}$; we use this function to update loop contexts depending on the values of loop conditionals as we explain later in this section. A context ϕ is a pair of a call string and a loop string, and an LSA annotation $\hat{\kappa}_l$ maps a pair of a node or an edge and a context to its corresponding abstract state: $\hat{\kappa}_l(\langle p, \phi\rangle)$ gives an approximate input state at the program point p in the context ϕ. Note that a call string in a context ϕ denotes different information from a call string in a loop context element $\langle \pi, n_{lhead}, m\rangle$; the former is for k-CFA on which LSA is based, and the latter is to serve as a call context where the loop is introduced, which is necessary to change the current loop context properly when the flow changes by return statements in loops as we present later in this section. The domain $\hat{\mathbf{A}}_{\mathsf{lsa}}$ of LSA annotations is a complete lattice using the following order relation:

$$\forall \hat{\kappa}_{l_1}, \hat{\kappa}_{l_2} : (\hat{\kappa}_{l_1} \sqsubseteq \hat{\kappa}_{l_2} \iff \forall p, \phi : \hat{\kappa}_{l_1}(\langle p, \phi\rangle) \sqsubseteq \hat{\kappa}_{l_2}(\langle p, \phi\rangle)).$$

5.1.3 Loop-Sensitive Analysis

As with k-CFA, $\langle i, j, k\rangle$-LSA computes the least fixpoint of the transfer function $\hat{F}_{\mathsf{lsa}}$ on LSA annotations as follows:

$$\hat{\kappa}_{\mathsf{lsa}} = \mathsf{leastFix}\ \lambda \hat{\kappa}_l.(\hat{\kappa}_I \sqcup \hat{F}_{\mathsf{lsa}}(\hat{\kappa}_l))$$
$$\text{where } \hat{\kappa}_I(\langle p, \langle \pi, \psi\rangle\rangle) = \text{if } \langle p, \langle \pi, \psi\rangle\rangle = \langle \mathsf{n}^{\mathsf{global}}_{entry}, \langle \epsilon, \epsilon\rangle\rangle \text{ then } \hat{\tau}_{\mathsf{init}} \text{ else } \bot$$

where the analysis starts with the initial state $\hat{\tau}_{\mathsf{init}}$ at the global entry node in the empty call and loop contexts. Like $\hat{F}_{\mathsf{cfa}}$ in Figure 6, the definition of $\hat{F}_{\mathsf{lsa}}$ consists of node and edge transfer functions. The transfer functions for nodes and normal, call, and return edges remain the same as in k-CFA except that a context is now a pair of a call string and a loop string instead of a single call string. Due to the space limitation, we present core parts of the transfer functions for loop-related edges, and refer the interested readers to a companion report [12] for the full definition.

Loop-in edge. Figure 10 shows the definition of $\hat{F}_{\mathsf{lsa}}$ for loop-in edges. For presentation brevity, we omit universal quantifiers binding meta variables in obvious cases. For example, $\psi = \langle \pi_1, n_{lhead}, m\rangle \oplus \psi_1$ in the second rule is equal to $\forall \pi_1, n_{lhead}, m, \psi_1 : \psi = \langle \pi_1, n_{lhead}, m\rangle \oplus \psi_1$.

The first rule states that when $i = 0$, LSA is the same as k-CFA; the abstract states from the origin node of the edge propagate through the edge without changing the current loop string: $[\![stmt(e)]\!](\hat{\kappa}_l(\langle origin(e), \langle \pi, \psi\rangle\rangle))$. The function $\hat{F}_{\mathsf{lsa}}$ has similar rules for all the other loop-related edges, and we omit them in this paper for brevity.

The second rule specifies the case when the depth of the current loop string $|\psi|$ reaches the maximum depth i not by the current loop but by an outer loop: it is either from a different call context $\pi \neq \pi_1$ or from a different loop $n_{lhead} \neq target(e)$. Then, LSA simply propagates the abstract states from the origin node of the edge. Other loop-related edges also have similar rules.

The third and fourth rules describe the case with a join loop context, where the iteration count of the innermost loop string is 0. The third rule is when an outer loop creates the

$$\hat{F}_{\mathsf{lsa}}(\hat{\kappa}_l)(\langle e, \langle \pi, \psi \rangle\rangle) \quad \text{if } \mathit{loopInEdge}(e) \;=$$

$$\begin{cases}
[\![\mathit{stmt}(e)]\!](\hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \psi \rangle\rangle)) & \text{if } i = 0 \\
[\![\mathit{stmt}(e)]\!](\hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \psi \rangle\rangle)) & \text{if } |\psi| = i \;\wedge\; \psi = \langle \pi_1, n_{\mathit{lhead}}, m \rangle \oplus \psi_1 \;\wedge \\
 & \quad (\pi \neq \pi_1 \vee\; n_{\mathit{lhead}} \neq \mathit{target}(e)) \\
[\![\mathit{stmt}(e)]\!](\hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \psi \rangle\rangle)) & \text{if } |\psi| < i \;\wedge\; \psi = \langle \pi_1, n_{\mathit{lhead}}, 0 \rangle \oplus \psi_1 \;\wedge \\
 & \quad (\pi \neq \pi_1 \vee n_{\mathit{lhead}} \neq \mathit{target}(e)) \\
[\![\mathit{stmt}(e)]\!](\hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \psi_1 \rangle\rangle)) & \text{if } \psi = \langle \pi, \mathit{target}(e), 0 \rangle \oplus \psi_1 \;\wedge \\
 & \quad \forall \langle \pi_1, n_{\mathit{lhead}}, m \rangle, \psi_2 : \\
 & \qquad \langle \pi_1, n_{\mathit{lhead}}, m \rangle \oplus \psi_2 = \psi_1 \wedge m \neq 0 \\
[\![\mathit{stmt}(e)]\!](\hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \psi_1 \rangle\rangle)) & \text{if } \psi = \langle \pi, \mathit{target}(e), 1 \rangle \oplus \psi_1
\end{cases}$$

Figure 10 Transfer functions for loop-in edges.

$$\hat{F}_{\mathsf{lsa}}(\hat{\kappa}_l)(\langle e, \langle \pi, \psi \rangle\rangle) \quad \text{if } \mathit{loopIterEdge}(e) \;\wedge \psi = \langle \pi, \mathit{target}(e), m \rangle \oplus \psi_1 \;=$$

$$\begin{cases}
[\![\mathit{stmt}(e)]\!](\hat{\tau}_1) \sqcup [\![\mathit{stmt}(e)]\!](\hat{\tau}_2) & \text{if } m = j \;\wedge \\
 & \quad \hat{\tau}_1 = \hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \langle \pi, \mathit{target}(e), m{-}1 \rangle \oplus \psi_1 \rangle\rangle) \;\wedge \\
 & \quad \hat{\tau}_2 = \hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \psi \rangle\rangle) \\
[\![\mathit{stmt}(e)]\!](\hat{\tau}_1) \sqcup [\![\mathit{stmt}(e)]\!](\hat{\tau}_2) & \text{if } 2 \leq m \leq j{-}1 \;\wedge \\
 & \quad \hat{\tau}_1 = \hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \langle \pi, \mathit{target}(e), m{-}1 \rangle \oplus \psi_1 \rangle\rangle) \;\wedge \\
 & \quad \hat{\tau}_2 = \hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \psi \rangle\rangle) \;\wedge \\
 & \quad (\mathit{check}(\mathit{stmt}(e), \hat{\tau}_1) = \top_b \;\vee\; \mathit{check}(\mathit{stmt}(e), \hat{\tau}_2) = \top_b) \\
[\![\mathit{stmt}(e)]\!](\hat{\tau}) & \text{if } 2 \leq m \leq j{-}1 \;\wedge \\
 & \quad \hat{\tau} = \hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \langle \pi, \mathit{target}(e), m{-}1 \rangle \oplus \psi_1 \rangle\rangle) \;\wedge \\
 & \quad (\mathit{check}(\mathit{stmt}(e), \hat{\tau}) = \hat{\mathsf{true}} \;\vee\; \mathit{check}(\mathit{stmt}(e), \hat{\tau}) = \hat{\mathsf{false}})
\end{cases}$$

Figure 11 Transfer functions for loop-iter edges.

join loop context and the fourth rule is when the current loop creates it. The condition $\forall \langle \pi_1, n_{\mathit{lhead}}, m \rangle, \psi_2 : \langle \pi_1, n_{\mathit{lhead}}, m \rangle \oplus \psi_2 = \psi_1 \wedge m \neq 0$ in the fourth rule prevents prepending another join loop context when the outer loop of the current one already creates a join loop context. In a join loop context, LSA joins the analysis results from loop-in and loop-iter edges as the final analysis result of the loop. Note that join loop contexts have the same effect as preventing loops with non-deterministic loop conditions from unrolling and thus avoiding unnecessary fixpoint computation.

The fifth rule addresses the case for analyzing loop bodies. It prepends a new loop context $\langle \pi, \mathit{target}(e), 1 \rangle$ denoting the first iteration of the loop in the same call context to the current loop string, and propagates the abstract states from the origin node of the edge in the context before the loop through the edge: $[\![\mathit{stmt}(e)]\!](\hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \psi_1 \rangle\rangle))$.

Loop-iter edge. Figure 11 shows a partial definition of $\hat{F}_{\mathsf{lsa}}$ for loop-iter edges, which propagate abstract states resulting from loop iterations to a loop-head node $\mathit{target}(e)$ introducing new loop contexts whenever necessary. We omit the rules similar to the ones for loop-in edges for brevity.

$$\hat{F}_{\mathsf{lsa}}(\hat{\kappa}_l)(\langle e, \langle \pi, \psi \rangle\rangle) \quad \text{if } \mathit{loopOutEdge}(e) \ \wedge \ i \neq 0 \ \wedge \ |\psi| < i$$
$$= \bigsqcup \{[\![\mathit{stmt}(e)]\!](\hat{\tau}) \mid 1 \leq m \leq j \ \wedge \ \hat{\tau} = \hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \langle \pi, \mathit{origin}(e), m\rangle \oplus \psi\rangle\rangle) \wedge \hat{\mathsf{false}} \sqsubseteq \mathit{check}(\mathit{stmt}(e), \hat{\tau})\}$$

$$\hat{F}_{\mathsf{lsa}}(\hat{\kappa}_l)(\langle e, \langle \pi, \psi \rangle\rangle) \quad \text{if } \mathit{loopBreakEdge}(e) \ \wedge \ i \neq 0 \ \wedge \ |\psi| < i$$
$$= \bigsqcup \{[\![\mathit{stmt}(e)]\!](\hat{\tau}) \mid 1 \leq m \leq j \wedge \hat{\tau} = \hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \langle \pi, \mathit{loopHead}(\mathit{origin}(e)), m\rangle \oplus \psi\rangle\rangle)\}$$

$$\hat{F}_{\mathsf{lsa}}(\hat{\kappa}_l)(\langle e, \langle \pi, \psi \rangle\rangle) \quad \text{if } \mathit{loopReturnEdge}(e) \ \wedge \ i \neq 0 \ \wedge \ |\psi| < i$$
$$= \bigsqcup \{[\![\mathit{stmt}(e)]\!](\hat{\tau}) \mid \psi_1 \in \mathbf{\Psi} \ \wedge \ n_{lhead_2} \in \mathit{loopHeads}(\mathit{origin}(e)) \wedge \langle \pi, n_{lhead_2}, m_2\rangle \in \psi_1 \wedge 1 \leq m_2 \leq j \ \wedge \ \hat{\tau} = \hat{\kappa}_l(\langle \mathit{origin}(e), \langle \pi, \psi_1 || \psi\rangle\rangle)\}$$

Figure 12 Transfer functions for loop-out, loop-break, and loop-return edges.

The first rule specifies the case when the iteration count m reaches the maximum length j. In this case, all further analysis results from the current iteration in the same loop context are joined with the analysis result from the previous iteration. Note that i and j in $\langle i, j, k\rangle$-LSA with the definition of $\hat{F}_{\mathsf{lsa}}$ ensure that the analysis terminates using a finite loop-string domain.

The second and third rules are for the cases when the iteration count m does not reach the maximum length j. The second rule is when the analysis result of a loop conditional expression $\mathit{stmt}(e)$ in the current iteration is $\top_b$; because all further iterations from the current one are indistinguishable, the analysis results are combined with the analysis result from the previous iteration. In the third rule, because the analysis result of a loop conditional expression in the previous iteration is either true or false, it does not need to join analysis results.

Loop-out edge. Figure 12 presents representative rules for the remaining loop-related edges. The first rule specifies that a loop-out edge propagates an abstract state from a loop to a loop-out node removing the current loop context from the loop string only when the analysis result of the loop conditional expression is false or $\top_b$: $\hat{\mathsf{false}} \sqsubseteq \mathit{check}(\mathit{stmt}(e), \hat{\tau})$. By this condition, when the analysis result of a loop conditional expression is true, a loop-out edge does not propagate the abstract state from a loop to a loop-out node, which improves the analysis precision of nodes after loops.

Loop-break edge. As the second rule in Figure 12 specifies, a loop-break edge propagates an abstract state from a break node inside a loop to a loop-out node. Since the evaluation of the `break` statement breaks out of a loop, the rule reverts the loop string to the one before the current loop by finding the innermost loop-head node with *loopHead* and reverting to the loop context just before it.

Loop-return edge. The third rule in Figure 12 states that a loop-return edge propagates an abstract state from a return node inside a loop to an exit node of a function enclosing the loop. Because the evaluation of the `return` statement breaks out all the enclosing loops and returns from the enclosing function, the rule reverts the loop string to the one before the current function is called by finding all the enclosing loop-head nodes with *loopHeads*

and reverting to the loop context just before the call of the current function. Using the concatenation operator $||$, the rule joins all analysis results of the return node in contexts $\langle \pi, \psi_1 || \psi \rangle$ where ψ_1 contains loop contexts with the current call string π and the enclosing loop-head nodes of the return node: $n_{lhead_2} \in \mathit{loopHeads}(\mathit{origin}(e))$.

We prove that $\hat{F}_{\mathsf{lsa}}$ is monotone on the domain of $\hat{\kappa}_l$.

▶ **Theorem 1.** *(Monotonicity of $\hat{F}_{\mathsf{lsa}}$)*
$\forall \hat{\kappa}_{l_1}, \hat{\kappa}_{l_2} : \hat{\kappa}_{l_1} \sqsubseteq \hat{\kappa}_{l_2} \Longrightarrow \hat{F}_{\mathsf{lsa}}(\hat{\kappa}_{l_1}) \sqsubseteq \hat{F}_{\mathsf{lsa}}(\hat{\kappa}_{l_2})$.

Then, the unique least fixpoint of $\hat{F}_{\mathsf{lsa}}$ exists, and the computation terminates since the domains of $\hat{F}_{\mathsf{lsa}}$ are finite.

5.2 Soundness and Precision

This section shows that $\langle i, j, k \rangle$-LSA extending sound k-CFA is also sound and it is more precise than or as precise as k-CFA. We proved all the theorems in this paper using the proof assistant tool Coq [5] and the Coq library of lattices[6]; the mechanized proofs are publicly available [12].

5.2.1 Soundness

We define translation functions between concrete summary maps and LSA annotations $\alpha : \mathbf{A_{con}} \to \hat{\mathbf{A}}_{\mathbf{lsa}}$ and $\gamma : \hat{\mathbf{A}}_{\mathbf{lsa}} \to \mathbf{A_{con}}$ as follows:

$$\alpha(\kappa_c) = \lambda(\langle p, \phi \rangle).\alpha_s(\kappa_c(p) \sqcap \mathit{statesInCtxt}(\phi))$$
$$\gamma(\hat{\kappa}_l) = \bigsqcup \{\kappa_c \mid \alpha(\kappa_c) \sqsubseteq \hat{\kappa}_l\}$$

where the helper function $\mathit{statesInCtxt} : \mathbf{\Phi} \to \wp(\mathbf{State})$ provides a set of all possible concrete states in a given context. For a given context ϕ, the meet of all reachable states $\kappa_c(p)$ without considering contexts and possibly unreachable states $\mathit{statesInCtxt}(\phi)$ in the context ϕ denotes a set of all reachable states in the context. Then, α and γ satisfy the Galois connection condition:

▶ **Theorem 2.** *(Galois connection)*
$\forall \kappa_c \in \mathbf{A_{con}}, \hat{\kappa}_l \in \hat{\mathbf{A}}_{\mathbf{lsa}} : \alpha(\kappa_c) \sqsubseteq \hat{\kappa}_l \Longleftrightarrow \kappa_c \sqsubseteq \gamma(\hat{\kappa}_l)$.

We assume that all reachable states in a context ϕ are subsumed by the join of execution results from all reachable states in the preceding contexts of ϕ. Then, we can prove the soundness of the transfer function:

▶ **Theorem 3.** *(Soundness of transfer functions)*
$\alpha \circ F_{con} \sqsubseteq \hat{F}_{\mathsf{lsa}} \circ \alpha$.

Finally, with all the theorems above, the abstract interpretation framework guarantees the soundness of LSA:

$$\alpha(\kappa_{con}) \sqsubseteq \hat{\kappa}_{\mathsf{lsa}}.$$

[6] `http://raweb.inria.fr/2006/Raweb/lande/uid20.html`

Table 1 Analysis results of SAFE, SAFE_{lsa}, TAJS, and WALA within the timeout of 5 hours.

Group (Number of programs)	Success			
	SAFE	**SAFE_{lsa}**	**TAJS**	**WALA**
jQuery 1.0.0∼2.1.0 (14)	0	14	11	3
Modernizr 2.8.3 (1)	1	1	1	1
BootStrap 3.3.0 (1)	0	0	0	0
Mootools 1.5.1 (1)	0	1	0	0
Prototype 1.7.2 (1)	0	1	0	0
BENCH (61)	0	37	20	0
SLICE (61)	0	53	33	0
WEBSITE (5)	0	3	0	0

5.2.2 Precision

To compare the analysis results of k-CFA and $\langle i, j, k\rangle$-LSA, we define a translation function from LSA annotations to CFA annotations $\eta : \hat{\mathbf{A}}_{\mathbf{lsa}} \to \hat{\mathbf{A}}_{\mathbf{cfa}}$ as follows:

$$\eta(\hat{\kappa}_l) = \lambda(\langle p, \pi\rangle).\bigsqcup \{\hat{\kappa}_l(\langle p, \langle \pi, \psi\rangle\rangle) \mid \psi \in \Psi\}.$$

It simply translates LSA contexts with the same call string (possibly with different loop strings) to a CFA context by joining the LSA contexts. Then, the analysis results from LSA are more precise than or at least as precise as the ones from CFA at all program points:

▶ **Theorem 4.** *(Precision)* $\eta(\hat{\kappa}_{\text{lsa}}) \sqsubseteq \hat{\kappa}_{\text{cfa}}$

6 Evaluation

In this section, we evaluate our technique in two respects, scalability and precision, using SAFE_{lsa}, an extension of an open-source framework SAFE [13, 15] that statically analyzes JavaScript web applications with modeling of various browser environments. We performed all experiments on a Mac OS X x64 machine with 3.4GHz Intel Core i7 CPU and 16GB Memory, and we used 30-depth 1000-length loop strings and 10-CFA for LSA. Even though we use big numbers for i and j for $\langle i, j, k\rangle$-LSA to ensure termination of analyses, we found that actual analyses create much smaller numbers of loop contexts without incurring much overhead. We also used 10-CFA for the experiment with SAFE for comparison.

6.1 Scalability

We evaluate the scalability of LSA by comparing the analysis results of SAFE_{lsa} with those of state-of-the-art static analyzers, SAFE, TAJS, and WALA; we refer the interested readers to Section 7 for more detailed explanation of the analyzers. For fair comparison, we used the latest versions of SAFE and WALA from their open-source repositories except for jQuery as we explain below and the specialized version for TAJS [18]. Similar to the experiments of TAJS [1], we measured how many target subjects each analyzer successfully analyzes within the timeout of 5 hours with normal analysis results. Table 1 summarizes the experimental results.

The first column in Table 1 shows target subject groups we experimented with and the numbers of programs in each group. The first 5 groups are simple programs that just load

one of the top 5 JavaScript libraries according to W3Techs[7]. Because jQuery has a more than 90% market share and many web applications still use old versions of jQuery even with newer versions, we analyze all 14 released versions of jQuery (1.0.0 $\sim$ 1.11.0, 2.0.0 $\sim$ 2.1.0) while we analyze only the latest versions for other libraries. The BENCH and SLICE groups are benchmarks from the experiments of TAJS [1]; the authors collected 71 programs from a jQuery tutorial[8] that perform simple operations using jQuery 1.10.0, and they compared analysis results of using the entire jQuery (BENCH) and using its sliced versions (SLICE). Note that our experiments have excluded 10 programs from the original 71 benchmarks because some soundness bugs in TAJS affected analysis results on the 10 programs. One such bug is that the loop specialization mechanism in TAJS misses analysis flows in the presence of the `return` statement in loops, which formal verification like our Coq mechanization can surely detect. The last group WEBSITE contains main web pages of the 5 most popular websites, `google.com`, `facebook.com`, `youtube.com`, `baidu.com`, and `yahoo.com`, according to the Alexa website[9].

The second to last columns in Table 1 show the successful analysis results. While SAFE analyzes only one program that loads Modernizr[10], SAFE_{lsa} performs the best: it analyzes all programs that load 14 versions of jQuery and the latest versions of Modernizr, Mootools, and Prototype; it analyzes 37, 53, and 3 programs in the BENCH, SLICE, and WEBSITE groups, respectively. Note that because TAJS does not support ES5 getters and setters, it cannot analyze jQuery version 2.x. Also, while WALA can analyze 3 versions of jQuery using the dynamic determinacy technique [20], the technique is not available from its latest open-source repository. Thus, Table 1 presents that WALA can analyze 3 versions of jQuery but it does not hold for the latest open-source version. TAJS and WALA analyze 11 and 3 versions of jQuery, respectively, but our experiments showed that they fail to analyze Mootools and Prototype unlike SAFE_{lsa}. For the BENCH and SLICE groups, TAJS analyzes 20 and 33 programs, respectively, but WALA analyzes none of them; for the WEBSITE group, both TAJS and WALA fail to analyze any of 5 programs in the group.

We believe that SAFE_{lsa} is more scalable than the other analyzers because of its ability to distinguish loops more precisely without much overhead. From the experiments with jQuery, we observed that LSA distinguishes at most 4 nested loops (4-depth) across function boundaries and maximum 36 iterations (36-length). This implies that LSA can keep small numbers of loop contexts in practice even when we use big numbers for i and j in $\langle i, j, k\rangle$-LSA. Note that any combinations of the i and j values bigger than 4 and 36 in $\langle i, j, 10\rangle$-LSA give the same analysis results in the jQuery cases with the same numbers of distinguished loop contexts. Moreover, while LSA can precisely analyze loops of any forms such as `for`, `for-in`, `while`, and `do-while`, loop specialization techniques in TAJS and WALA are applicable to only special forms of loops by choosing contexts in heuristic ways; we found that Mootools and Prototype used various forms of loops, which include those that the techniques in TAJS and WALA cannot handle.

We investigated reasons why SAFE_{lsa} fails to analyze BootStrap[11] and some programs in the BENCH and SLICE groups within the timeout of 10 minutes. One reason is state explosion by statically indeterminate values. For example, jQuery provides the `jQuery.now()` method that returns a number representing the current time; a sound static analysis result of the

[7] `http://w3techs.com/technologies/overview/javascript_library/all`
[8] `http://www.jquery-tutorial.net/`
[9] http://www.alexa.com/topsites
[10] `http://modernizr.com`
[11] `http://getbootstrap.com/javascript/`

Table 2 Analysis results of SAFE and SAFE_{lsa} within the timeout of 5 hours.

Target	SAFE				SAFE_{lsa}			
	time (s)	MaxCALL (#)	CALL (%)	PROP (%)	**time (s)**	MaxCALL (#)	CALL (%)	PROP (%)
jQuery 2.1.1	timeout	32	88.57	36.36	27.56	2	99.65	83.93
Modernizr 2.8.3	60.02	49	95.42	41.67	5.92	2	96.85	100.00
BootStrap 3.3.0	timeout	37	87.10	53.33	timeout	254	87.85	73.02
Mootools 1.5.1	timeout	76	39.47	9.09	226.95	3	99.13	93.88
Prototype 1.7.2	timeout	30	91.49	28.57	35.74	2	99.41	87.72
`google.com`	timeout	22	72.22	45.45	6,433.98	10	95.22	77.56
`facebook.com`	timeout	2	99.41	42.22	timeout	3	99.60	88.14
`youtube.com`	timeout	45	94.51	42.25	3,583.09	2	99.54	89.53
`baidu.com`	timeout	37	92.00	56.52	timeout	41	93.12	72.37
`yahoo.com`	timeout	64	94.31	88.44	7,244.78	2	99.21	97.34
Average	–	39	85.45	44.39	–	32	96.95	86.34

method call should be any number. We observed that such statically indeterminate values flow into loops making analysis results of loop conditional expressions also indeterminate, which prohibits LSA from analyzing loops precisely.

Another reason we found in BootStrap is also state explosion in loops due to a sound event modeling that considers all possible event-dispatch scenarios. Event handlers for an event can access the target DOM element where the event was initially fired by the `event.target` property. Because of event bubbling and capturing [25] (event propagation through the path from the root of a DOM tree to a target element), the target element may not be `event.currentTarget` for which the current event handler has been registered. Therefore, a sound static analysis result for `event.target` should be all DOM elements on the subtree of `event.currentTarget`, and we observed that such imprecise analysis results cause state explosion by flowing into loops.

We believe that other analysis techniques not particularly related to loops may alleviate the state explosion problems. Using random constant values as in TAJS [1] or user inputs for statically indeterminate values may lessen the former problem, and more sophisticated event modeling may mitigate the latter problem. Our future work includes these directions.

6.2 Precision

For precision, we compare the analysis results of SAFE and SAFE_{lsa} for programs that just load the latest versions of the top 5 libraries and main web pages of the 5 most popular websites. To measure the analysis precision, we compute MaxCALL, CALL, and PROP that can be critical in the analysis scalability: (1) MaxCALL indicates the maximum number of possible function calls resolved during analysis for each call; bigger numbers denote more imprecise analysis results leading to more false function calls, which harm the analysis

scalability. (2) CALL indicates the ratio of definite function calls resolved to exactly one function calls to all function calls; bigger numbers denote more precise analysis results with less spurious function calls. (3) PROP indicates the ratio of dynamic property accesses resolved to constant names to all dynamic property accesses without considering direct constant accesses like `o["name"]`; bigger numbers denote more precise analysis results with more exact property accesses. Thus, the bigger MaxCALL and the smaller CALL and PROP an analysis has, the more likely it suffers from the scalability problem.

Table 2 shows the result; we averaged all figures in the table from 3 runs and normalized them to per program point to directly compare SAFE and SAFE_{lsa}; for those with timeout, we used pre-fixpoint analysis results that are still useful for precision comparison. The table shows that SAFE_{lsa} significantly improves analysis results of SAFE; it analyzes more programs than and provides more precise analysis results than SAFE. On average, LSA reduces MaxCALL from 39 to 32 and it improves CALL and PROP from 85.45% and 44.39% to 96.95% and 86.34%, respectively. In the analysis of Mootools for example, LSA dramatically improves the analysis precision by reducing MaxCALL from 76 to 3 and improving CALL and PROP from 39.47% and 9.09% to 99.13% and 93.88%, respectively. Interestingly, SAFE_{lsa} has bigger MaxCALL than SAFE for BootStrap, `facebook.com`, and `baidu.com`, which SAFE_{lsa} fails to analyze within the timeout. We found that SAFE_{lsa} reaches more program points to analyze and performs more fixpoint computation at the same program points than SAFE within the same timeout thanks to improved scalability, which increases MaxCALL at program points with imprecise analysis results. We expect that when analyzing the target programs without setting a timeout, final analysis results would show smaller MaxCALL in SAFE_{lsa} than SAFE.

6.3 Threats to Validity

A possible threat to validity of our results is that, since our target programs are simple programs that use top 5 libraries and main web pages of the 5 most popular websites, the results may not hold for some real-world JavaScript programs in different domains. Another threat is that the comparison results with other static analyzers are not independent of the capabilities of their base analyzers. As SAFE, WALA, and TAJS use different analysis techniques and different modeling of JavaScript built-in functions and DOM APIs, the better analysis results of SAFE_{lsa} may not be purely due to LSA.

7 Related Work

Sharir and Pnueli [21] introduced two approaches of call-context sensitivity, k-CFA and Summary-Based Analysis (SBA), to statically analyze functions more precisely. While k-CFA distinguishes function calls by call strings of the maximum k-length that represent call histories, SBA distinguishes function calls by input states to functions. They proved that k-CFA with the unbound length of k and SBA have the same analysis precision in domains with precision-lossless join operations. Mangal *et al.* [17] extended this result by showing that the result still holds in the presence of precision-lossy join operations. Using their notation, we formalized k-CFA and LSA in the abstract interpretation framework to prove the soundness and precision theorems of LSA. Note that although loop-sensitivity can be used with any call sensitivity techniques, LSA that we present in this work subsumes k-CFA; while k-CFA distinguishes call contexts only by some abstraction of call history denoted by call strings, LSA distinguishes loop contexts not only by abstraction of loop history but also

by abstraction of loop condition values. Thus, LSA is also similar to SBA in that it uses value abstraction for loop conditions.

Trace partitioning [10, 19] is a general theoretical framework that supports systematic abstractions on trace-based concrete semantics. While trace partitioning can improve the analysis precision of loops as well, each loop should be annotated with an unrolling number before the analysis just like the conventional loop unrolling technique. On the contrary, LSA finds precise unrolling counts for loops automatically during analysis as far as loop conditions keep determinate. Furthermore, our LSA formalization provides a detailed explanation about loop context updates in the presence of tricky program points such as `break`, `continue`, and `return` statements in loops, and our Coq mechanization proves its soundness.

SAFE [13, 15] is an extensible analysis framework for JavaScript web applications. SAFE performs sophisticated data flow analyses for JavaScript applications in a flow-sensitive and context-sensitive way producing heap information that contains pointer and value information for all variables at each program point as analysis results. In addition to the data flow analysis, it supports various extensions such as the ES6 module system [4] and detection of Web API misuses [3]. Our experiments showed that SAFE_{lsa} significantly improves the scalability and the precision of SAFE, which may, in turn, improve those of the various extensions of SAFE.

TAJS [18] is a flow and context sensitive static analyzer for JavaScript applications similar to SAFE. In addition to the object sensitivity [22], TAJS extended its context-sensitivity to distinguish more functions and loops using the values of function parameters and loop variables selected in heuristic ways, and the extension enabled TAJS to analyze most versions of jQuery [1]. However, the technique does not accompany any formalization nor soundness proof, and it is not general in that it can distinguish loops only when they conform to some specific forms; for instance, the technique does not distinguish loops where variables in loop conditional expressions are not involved in object property updates. Indeed, we found some soundness bugs and observed that TAJS cannot analyze 3 libraries among the top 5 ones.

WALA [11] is a general analysis framework originally for Java, and it analyzes JavaScript web applications as well. To address JavaScript-specific scalability problems, WALA developed the correlation tracking technique [23] that rewrites `for-in` loops in the following forms with the same property reads and writes:

```
for(x in src) des[x] = src[x];
```

to the following code:

```
for(x in src) (function (p) {des[p] = src[p]}) (x);
```

Then, the analysis distinguishes the function call in each iteration using field sensitivity (distinguishing function calls by fields of objects) [16]. Later, WALA presented the dynamic determinacy analysis [20], which first performs a dynamic information flow analysis [2] to track ever-determinate values in all concrete executions, propagates such constant values in a program, and then performs a static analysis on the specialized program. In particular for loops, the technique uses such determinate values to find the maximum unrolling numbers. However, even with the correlation tracking and dynamic determinacy techniques, WALA can analyze only 3 versions of jQuery. In contrast, LSA can apply to any forms of loops and it is fully static and automatic for finding precise numbers of distinguishable iterations for loops.

Kashyap *et al.* [14] presented JSAI, a static analysis platform to support sound analysis for JavaScript applications similar to SAFE and TAJS. One main feature of JSAI is the

configurability of sensitivity techniques including call-context sensitivity. The authors evaluated JSAI applying various call-context sensitivities such as k-CFA and object sensitivity to various benchmarks. Their experimental results showed that static analysis with higher sensitivity (as greater k for k-CFA) is far better than its counterpart with low sensitivity in terms of performance in various kinds of JavaScript programs giving more precise analysis results. It implies that higher precision may be a key to higher scalability of static analysis at least for JavaScript programs. In Section 6, We also showed that higher precision by LSA significantly improves the analysis performance for some JavaScript web applications. Because JSAI does not support the comprehensive modeling of JavaScript built-in functions including browser APIs that are essential for analyzing JavaScript web applications, we could not compare JSAI with our implementation. However, we conjecture that JSAI would fail to analyze many JavaScript web applications including JavaScript libraries unless it supports a loop specialization technique such as LSA; in Section 2, we showed that call-context sensitivity alone is not enough to analyze the most widely used JavaScript library in a scalable way due to great loss of precision in loops.

8 Conclusion

We presented a novel analysis technique, Loop-Sensitive Analysis (LSA), which distinguishes loop iterations as many as needed during analysis using loop contexts. We formalized LSA in the abstract interpretation framework and showed how to extend k-CFA to LSA. We proved that LSA is more precise than or at least as precise as k-CFA while it remains sound if the base k-CFA is sound. We provide the mechanized proofs of the soundness and precision theorems by the proof assistant tool, Coq. We have implemented LSA as an extension of an open-source JavaScript static analysis framework, SAFE. Our mechanized proofs and the LSA implementation are publicly available. We demonstrated that LSA dramatically improves the scalability of the state-of-the-art JavaScript static analyzers by enhancing analysis precision when analyzing the main web pages of the 5 most popular websites and applications that use the top 5 JavaScript libraries. Because we presented LSA as language independent, it is applicable to analysis of programs in other programming languages.

Acknowledgments. This work is supported in part by Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grants NRF-2014R1A2A2A010 03235 and NRF-2008-0062609), Samsung Electronics, and Google.

References

1 Esben Andreasen and Anders Møller. Determinacy in static analysis for jQuery. In *OOPSLA'14: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2014.

2 Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS'09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*. ACM, 2009.

3 SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. SAFE$_{\text{WAPI}}$: Web API misuse detector for web applications. In *ESEC/FSE'14: Proceedings of the 22nd ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM, 2014.

4 Junhee Cho and Sukyoung Ryu. JavaScript module system: Exploring the design space. In *Modularity'14: Proceedings of the 13th International Conference on Modularity*, 2014.

5 The Coq Proof Assistant. `http://coq.inria.fr/`.

6 Patrick Cousot. Semantic foundations of program analysis. In *Program Flow Analysis: Theory and Applications, Chapter 10*, pages 303–342. Prentice-Hall, Inc., 1981.

7 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, 1977.

8 Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL'79: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, 1979.

9 ECMA. ECMA-262: ECMAScript Language Specification. Edition 5.1, 2011.

10 Maria Handjieva and Stanislav Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *Static Analysis, 5th International Symposium, SAS'98, Pisa, Italy, September 14-16, 1998, Proceedings*, pages 200–214. Springer-Verlag, 1998.

11 IBM Research. T.J. Watson Libraries for Analysis (WALA). `http://wala.sf.net`.

12 KAIST PLRG. Research material. `http://plrg.kaist.ac.kr/pch`.

13 KAIST PLRG. SAFE: Scalable analysis framework for ECMAScript. `http://safe.kaist.ac.kr`, 2014.

14 Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben 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.

15 Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *FOOL'12: International Workshop on Foundations of Object Oriented Languages*, 2012.

16 Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using SPARK. In *CC'03: Proceedings of the 12th International Conference on Compiler Construction*. Springer-Verlag, 2003.

17 Ravi Mangal, Mayur Naik, and Hongseok Yang. A correspondence between two approaches to interprocedural analysis in the presence of join. In *ESOP 2014: Proceedings of the 23rd European Symposium on Programming*. Springer, 2014.

18 Anders Møller, Simon Holm Jensen, Peter Thiemann, Magnus Madsen, Matthias Diehn Ingesman, Peter Jonsson, and Esben Andreasen. TAJS: Type analyzer for JavaScript. `https://github.com/cs-au-dk/TAJS`, 2014.

19 Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM TOPLAS*, 29(5), 2007.

20 Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Dynamic determinacy analysis. In *PLDI'13: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2013.

21 Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications, Chapter 7*. Prentice-Hall, 1981.

22 Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *POPL'11: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2011.

23 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. In *ECOOP'12: Proceedings of the 26th European Conference on Object-Oriented Programming*. Springer-Verlag, 2012.

24 Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 1955.

25 W3C. Document Object Model Events. `http://www.w3.org/TR/2003/NOTE-DOM-Level-3-Events-20031107`, 2003.