

Practical Analysis of the Dynamic Characteristics of JavaScript

Shiyi Wei

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Barbara G. Ryder, Chair
Laurie J. Hendren
Dennis G. Kafura
Eli Tilevich
Danfeng Yao

August 31, 2015
Blacksburg, Virginia

Keywords: Dataflow Analysis, JavaScript, Context Sensitivity
Copyright 2015, Shiyi Wei

Practical Analysis of the Dynamic Characteristics of JavaScript

Shiyi Wei

ABSTRACT

JavaScript is a dynamic object-oriented programming language, which is designed with flexible programming mechanisms. JavaScript is widely used in developing sophisticated software systems, especially web applications. Despite of its popularity, there is a lack of software tools that support JavaScript for software engineering clients. Dataflow analysis approximates software behavior by analyzing the program code; it is the foundation for many software tools. However, several unique features of JavaScript render existing dataflow analysis techniques ineffective.

Reflective constructs, generating code at runtime, make it difficult to acquire the complete program at compile time. Dynamic typing, resulting in changes in object behavior, poses a challenge for building accurate models of objects. Different functionalities can be observed when a function is variadic; the variance of the function behavior may be caused by the arguments whose values can only be known at runtime. Object constructors may be polymorphic such that objects created by the same constructor may contain different properties. In addition to object-oriented programming, JavaScript supports paradigms of functional and procedural programming; this feature renders dataflow analysis techniques ineffective when a JavaScript application uses multiple paradigms. Dataflow analysis needs to handle these challenges.

In this work, we present an analysis framework and several dataflow analyses that can handle dynamic features in JavaScript. The first contribution of our work is the design and instantiation of the JavaScript Blended Analysis Framework (JSBAF). This general-purpose and flexible framework judiciously combines dynamic and static analyses. We have implemented an instance of JSBAF, blended taint analysis, to demonstrate the practicality of the framework.

Our second contribution is a novel context-sensitive points-to analysis for JavaScript that accurately models object property changes. This algorithm uses a new program representation that enables partial flow-sensitive analysis, a more accurate object representation, and an expanded points-to graph. We have defined parameterized state sensitivity (i.e., k -state sensitivity) and evaluated the effectiveness of 1-state-sensitive analysis as the static phase of JSBAF.

The third contribution of our work is an adaptive context-sensitive analysis that selectively applies context-sensitive analysis on the function level. This two-staged adaptive analysis extracts function characteristics from an inexpensive points-to analysis and uses learning-based heuristics to decide on an appropriate context-sensitive analysis per function. The

experimental results show that the adaptive analysis is more precise than any single context-sensitive analysis for several programs in the benchmarks, especially for those multi-paradigm programs.

The research in this thesis was supported by National Science Foundation CCF-0811518 and IBM Open Collaborative Research Program.

Dedicated to
my parents, Chengqian Wei and Min Fu
my wife, Yilei Cao
my son, Simon Wei
for their endless love and support.

Acknowledgments

My exciting and fruitful experience during graduate study would not have been possible without the support from many people. First, I would like to express the deepest gratitude to Dr. Barbara G. Ryder, who has been my extraordinary advisor for the past six years. Her priority as a supervisor has always been my development, providing countless guidance to develop me becoming an independent researcher. She always takes care of my needs with patience, encouragement and support. I remembered writing papers and the meetings with Barbara after completing her day-long administrative duties. Closely working together for years, I was influenced by her passion and determination towards life. I could not have imagined a better mentor.

I would like to thank my committee members. Chatting with Dr. Eli Tilevich is always delightful experience and he provided advice that was invaluable to my graduate study. Dr. Laurie Hendren made extremely helpful suggestions on improving this thesis and I appreciate the time she spent reading my work. Drs. Dennis Kafura and Danfeng Yao provided additional domain expertise in system and security to my advisory committee and they were always responsive to my needs.

I also would like to thank my collaborators at IBM T. J. Watson Research Center, Drs. Julian Dolby and Omer Tripp. The summer intern with Julian has envisioned me deeper understanding of program analysis research and was valuable experience on research collaboration. He provided a lot of help on the details of WALA in this thesis and other work. Discussions with Omer always resulted in wonderful research ideas and concrete plans to make progress in a research project. Omer is very passionate about exciting research and it is enjoyable to work with him.

Several members of PROLANGS research group have helped with my research. The discussions with Drs. Marc Fisher II and Ben Wiedermann formed multiple early ideas in this thesis. The collaboration with Franceska Xhakaj resulted in an empirical study with insightful observations. Several other undergraduate researchers have contributed to the benchmarks used in this thesis. Their commitments as young researchers were inspiring to me.

Needless to say, family is the focus and motivation of my life. My parents are always behind me, supporting all the decisions I made. I am very thankful to my wife, Yilei, for her

understanding, caring and love. Nowadays, the time with my son, Simon, is the highlight of my life.

Contents

1	Introduction	1
1.1	Contributions	2
1.1.1	JavaScript Blended Analysis Framework	3
1.1.2	State-sensitive Points-to Analysis	3
1.1.3	Adaptive Context-sensitive Analysis	4
1.2	Thesis Organization	5
2	Background and Terminology	6
2.1	Background	6
2.1.1	JavaScript Dynamic Features	6
2.1.2	Points-to Analysis	7
2.2	Terminology	10
2.2.1	Blended Analysis	10
2.2.2	JavaScript Object-Reference State	10
3	JavaScript Blended Analysis Framework	12
3.1	JSBAF Design	12
3.1.1	Framework Overview	12
3.1.2	JSBAF for Dynamic Features	14
3.1.3	Soundness of JSBAF	18
3.2	An Instantiation: Blended Taint Analysis	20
3.2.1	Blended Taint Analysis	21

3.2.2	Evaluation	24
4	State-sensitive Points-to Analysis	29
4.1	Imprecision of Points-to Analysis	29
4.2	Empirical Study of JavaScript Object Behavior	32
4.2.1	Experimental Design	33
4.2.2	Metrics	36
4.2.3	Object Behavioral Patterns	39
4.2.4	Summary	43
4.3	State-sensitive Points-to Analysis	44
4.3.1	State-Preserving Block Graph	44
4.3.2	Points-to Graph Representation	46
4.3.3	Transfer Functions	48
4.3.4	State Sensitivity	54
4.3.5	Block Sensitivity	55
4.3.6	State-sensitive Analysis as an Instantiation of JSBAF	56
4.4	Evaluation	58
4.4.1	Experimental Design	58
4.4.2	Experimental Results	59
5	Adaptive Context-sensitive Analysis	63
5.1	Empirical Study of JavaScript Context Sensitivity	63
5.1.1	Experimental Design	64
5.1.2	Results	65
5.2	Adaptive Context-sensitive Analysis	67
5.2.1	Overview	68
5.2.2	Function Characteristics	69
5.2.3	Heuristics	71
5.2.4	Selection Workflow	77

5.3	Evaluation	78
5.3.1	Experimental Setup	78
5.3.2	Experimental Results	79
5.3.3	Discussion	83
6	Related Work	84
6.1	Related Analyses of Dynamic Languages	84
6.1.1	Empirical Studies of JavaScript Applications	84
6.1.2	JavaScript Analyses of Dynamic Features	85
6.1.3	JavaScript Security Analyses	88
6.1.4	Studies and Analyses of Other Dynamic Languages	88
6.2	Context-sensitive Analysis	89
6.2.1	Object-sensitive Analysis	90
6.2.2	Selective Context-sensitive Analysis	90
7	Conclusions and Future Work	92
7.1	JavaScript Blended Analysis Framework	92
7.2	State-sensitive Points-to Analysis	93
7.3	Adaptive Context-sensitive Analysis	93
7.4	Future Work	93
	Bibliography	96

List of Figures

2.1	obj-ref state example	11
3.1	JavaScript Blended Analysis Framework	13
3.2	An alternative design of JSBAF static phase	13
3.3	Instrumentation heuristics and cost model of Execution Collector	14
3.4	JavaScript <code>eval</code> example	15
3.5	Simplified jQuery <code>extend</code> function	16
3.6	JavaScript polymorphic constructor example	17
3.7	JavaScript type-based dynamic dispatch example	18
3.8	Soundness of blended analysis	19
3.9	Blended taint analysis for JavaScript web applications	21
4.1	JavaScript example of dynamic object behavior	30
4.2	Run-time points-to graph at line 10	30
4.3	Run-time points-to graph at line 15	31
4.4	Flow- and context-insensitive points-to graph	32
4.5	Percentage of object instances in each object category	33
4.6	Percentage of operations associated with each object category	34
4.7	Sample history information of an individual object from <i>yahoo</i>	35
4.8	Read vs. write and delete operation distribution	36
4.9	Read-local and read-inherit operation distribution	36
4.10	Write and delete operation distribution	36

4.11	Local sizes of user objects at their constructed and last operations	37
4.12	Inherited vs. local properties in user objects	38
4.13	Accessible vs. inaccessible properties from prototype objects	38
4.14	CFG	44
4.15	SPBG	45
4.16	Property write example: input points-to graph	51
4.17	Property write example: updated points-to graph	51
4.18	Optimized object property lookup algorithm: <code>lookup(v, p)</code>	52
4.19	obj-ref state of \mathcal{O}_1	54
4.20	context element of 1-state sensitivity	55
4.21	Blended state-sensitive points-to graph at line 10	57
4.22	Blended state-sensitive points-to graph at line 15	57
5.1	Precision results for the Pts-Size client	66
5.2	Precision results for the REF client	67
5.3	Workflow of two-staged adaptive context-sensitive analysis	68
5.4	Heuristic: baseline vs. 1-call-site	73
5.5	Heuristic: baseline vs. 1-object	73
5.6	Heuristic: baseline vs. 1st-parameter	74
5.7	Heuristic: 1-call-site vs. 1-object	74
5.8	Heuristic: 1-call-site vs. 1st-parameter	75
5.9	Heuristic: 1-object vs. 1st-parameter	76
5.10	Heuristic: ith-parameter vs. jth-parameter	76
5.11	Workflow to select a context-sensitive analysis for a JavaScript function	77
5.12	Analysis precision on Benchmarks I	80
5.13	Analysis precision on Benchmarks II	82

List of Tables

3.1	Benchmarks	25
3.2	Taint analysis time	26
3.3	Taint analysis results	27
4.1	The relationship between statements and operation kinds	35
4.2	Operation occurrence patterns of user objects	39
4.3	Property change patterns of user objects	43
4.4	Expanded points-to graph with annotations	47
4.5	Transfer functions of object creation, direct write and property write statements	49
4.6	Transfer functions of property delete, property read and function invocation statements	50
4.7	Access path edges union rules	56
4.8	Property reference edges union rules	56
4.9	REF analysis precision	59
4.10	REF analysis cost (in seconds) on average per webpage	60
4.11	Benchmark and context statistics	61
5.1	Function characteristics	69
5.2	Selection precision for Benchmarks I	81

Chapter 1

Introduction

A family of object-oriented programming languages shares characteristics that render their run-time behavior unpredictable. These languages are usually designed with flexible mechanisms for loading/generating new code, dynamic typing, etc. The term *dynamic programming language* is used for such languages as JavaScript, PHP, Perl and Ruby. They are widely used in developing sophisticated software systems, especially web and mobile applications. Specifically, JavaScript is the *lingua franca* of client-side web applications, used by 89% of all website software [75] and according to a recent study, JavaScript is becoming the most popular programming language overall [73]. Its dynamic language features enable flexible and interactive website design. In addition, JavaScript is supported by almost all modern browsers.

Despite of its popularity, JavaScript applications present various software engineering challenges such as security and program understanding. For example, the program constructs supporting dynamic code generation provide opportunities for cross-site scripting security exploits. Given the ubiquity of JavaScript, it is crucial to build automated software analysis tools that can handle these problems. Dataflow analysis approximates the software behavior by analyzing program code. The results of dataflow analysis may enable code optimization, assist program understanding and/or detect security vulnerabilities. It has been demonstrated that static program analysis is useful in analyzing other languages such as Java and C++. Unfortunately, the dynamic nature of JavaScript is a double-edged sword, rendering static analysis techniques ineffective in many cases.

Static analysis inspects program source code that is visible at compile time. Invocations of reflective constructs¹ (e.g., the `eval` function in JavaScript) at runtime generate new code. The generated code is difficult to model statically because the parameters can be arbitrary values set at runtime. If static analysis ignores the *dynamic code generation* mechanisms in JavaScript applications, the solution of static analysis is unsound. *Function variadic-*

¹Reflection is the ability of a program to observe program state or change program code at runtime [44].

ity occurs when a function can be called with indefinite number of arguments, regardless of its declared arity. Function behavior usually varies based on the arguments provided. Static analysis cannot model a variadic function accurately because sometimes the actual arguments provided can only be known at runtime. JavaScript is also *dynamically typed*, lacking a static type system. This suggests that it is difficult for static analysis to build accurate models of objects in JavaScript, and that objects can exhibit different behavior at different times during execution due to the use of the *delegation* feature in *prototype-based programming* [41, 76]. Moreover, object constructors may be polymorphic so that objects created by the same constructor may have different properties. Static analysis usually builds conservative models for a *polymorphic constructor* approximating all possible behaviors. Finally, JavaScript supports *multiple programming paradigms*. In addition to object-oriented programming, it also supports features of functional (e.g., first-class functions) and procedural programming. A static analysis technique often is designed to be effective on a specific programming paradigm; the flexible programming paradigm of JavaScript renders static analysis more complicated.

Despite the fact that various program analysis approaches have been proposed specifically for JavaScript applications (e.g., [30, 72, 33]), the above language features still lead to the ineffectiveness of static analysis. In this thesis we present new dataflow analyses that accurately handle the dynamic characteristics of JavaScript and we also evaluate their practicality on software engineering problems (e.g., security). We have proposed *blended analysis* that combines dynamic and static analyses as a general-purpose dataflow analysis framework for JavaScript. Blended analysis collects run-time information such as dynamically generated code and variadic functions to expand the capability of static analysis to analyze these dynamic features. We also have presented a new context-sensitive analysis (i.e., *state sensitivity*) that accurately models the dynamic behavior of JavaScript objects. Finally, we have proposed a two-staged *adaptive analysis* that selects the appropriate context sensitivity on the function level based on heuristics. The adaptive analysis aims to handle the multi-programming paradigm nature of JavaScript.

The new dataflow analysis framework and algorithms presented in this thesis seek to improve the state-of-the-art program analysis for JavaScript and to build better software engineering tools. The proposed approaches may also be applicable to other programming languages with dynamic characteristics similar to JavaScript.

1.1 Contributions

The major contributions of the work presented in this thesis are (i) JavaScript blended analysis framework, (ii) state-sensitive points-to analysis, and (iii) adaptive context-sensitive analysis.

1.1.1 JavaScript Blended Analysis Framework

The first contribution of our work is the design of *JavaScript Blended Analysis Framework* (JSBAF), a general-purpose framework for analyzing JavaScript applications. The framework is designed to judiciously combine dynamic and static analyses in a practical but unsound analysis of JavaScript, to account for the effects of dynamic features not seen by static analysis, while providing sufficient accuracy to be useful. The analysis dynamically collects executions and performs static analysis on their calling structures. JavaScript blended analysis captures rich information about dynamic language features. These include dynamically generated (or loaded) JavaScript code (e.g., through `eval` functions or interpreted `urls`) and variadic function usage. JSBAF is general-purpose in that the dynamic phase may collect run-time information in different levels of detail and the static phase may apply various analysis clients to serve different requirements and goals of JavaScript analysis.

We implemented an instance of JSBAF or *blended taint analysis² for JavaScript*, to demonstrate the accuracy and practicality of the framework. In the dynamic phase, we used an instrumented browser to record execution traces of JavaScript websites; in the static phase, we implemented a taint analysis based on the points-to analysis³ solution. We present an empirical comparison of our blended results with two different static analysis approaches (i.e., one static analysis analyzes libraries and one does not). The experimental results demonstrate the practicality of our approach, and its scalability and precision with respect to static analysis on 12 popular websites. Less than half of the 13 true security exploits found by blended analysis were identified by either of the static analyses. Moreover, blended analysis reported only one false positive vulnerability (i.e., false alarm). Additionally, the static phase of our blended analysis ran to completion under a limited time budget of 10 minutes, whereas one static analysis that analyzes JavaScript libraries used by the websites failed to complete on 41% of the webpages analyzed.

1.1.2 State-sensitive Points-to Analysis

Instead of class-based inheritance, JavaScript supports prototype-based inheritance [41, 76] that results in a JavaScript object inheriting properties from a chain of (at least one) prototype objects. The model also allows the properties of a JavaScript object to be added, updated, or deleted at runtime. We have performed an in-depth empirical study to observe the run-time behavior of JavaScript objects.

The second contribution of our work is a novel points-to algorithm that can accurately model JavaScript objects. In our approach, changes to object properties are tracked more accurately to reflect object run-time behavior at different program points. A new graph decomposition for control flow graphs is used to better track object property changes. The

²Taint analysis detects flows of data that violate program integrity.

³Points-to analysis calculates the set of values a reference property or variable may have during execution.

analysis identifies objects by their creation site as well as their local property names upon construction, more accurately than the per-creation-site representation. To distinguish polymorphic constructors, this analysis incorporates dynamic information collected at runtime. Technically, the analysis is *partially flow-sensitive* (on our new control flow graph structure) and *context-sensitive*, using a new form of object sensitivity [47]⁴. We defined a parameterized model of k -state sensitivity. Rather than using the receiver object name as a calling context in the analysis, we use an approximation of the receiver object and its properties at the call site (i.e., *obj-ref state*).

In order to compare this algorithm with previous techniques, we instantiated the state-sensitive points-to analysis as the static component of the JSBAF. We measured performance and accuracy of our new analysis on a statement-level points-to client (REF analysis) that calculates how many objects are returned by a property lookup, for example, a read of property p on variable x . The experimental results showed a significant improvement in precision from the new analysis. On average over all 12 website benchmarks, 48% of the property lookup statements were resolved to a single abstract object by the state-sensitive analysis, while the blended implementation of an existing JavaScript analysis [72] uniquely resolved only 37% of these statements. Furthermore, although it incurred a 127% overhead, our new analysis was able to analyze each of the webpage in under 5 minutes, attesting to its scalability in practice.

1.1.3 Adaptive Context-sensitive Analysis

The third contribution of our work is an adaptive context-sensitive analysis that accurately analyzes multi-paradigm JavaScript applications by selecting specific context sensitivity on the function level. A recent empirical study on context-sensitive analyses for JavaScript revealed that there was no clear winner context-sensitive analysis for JavaScript across all benchmarks [33]. Because JavaScript features flexible programming paradigms and no single context-sensitive analysis seems best for analyzing JavaScript programs, there are opportunities for an adaptive (i.e., multi-choice) analysis to improve precision. We have performed a fine-grained study that compares the precision of four JavaScript analyses on the function level. We observed that JavaScript functions in the same program may benefit from use of different context-sensitive analyses depending on specific characteristics of the function.

The results of our empirical study guided us to design the novel adaptive analysis that selectively applies a specialized context-sensitive analysis per function chosen from call-site, object and parameter sensitivity. This two-staged analysis first applies an inexpensive points-

⁴Informally, a flow-sensitive analysis follows the execution order of statements in a program; flow-sensitive analysis can perform strong updates, but flow-insensitive cannot. Context-sensitive analysis distinguishes between different calling contexts (i.e., identification of the execution context from which the call is made) of a method, producing different analysis results for each context [60]. Context-insensitive analysis calculates one dataflow solution per method.

to analysis to a JavaScript program to extract function characteristics. We have designed heuristics according to our observations on the relationship between function characteristics and the precision of a specific analysis using the empirical results on the benchmark programs. Finally, an adaptive analysis based on the heuristic-based selection of a context-sensitive analysis per function is performed.

We have performed an evaluation of the adaptive context-sensitive analysis on two sets of benchmark programs. The experimental results show that our adaptive analysis was more precise than any single context-sensitive analysis for several applications in the benchmarks, especially for those using multiple programming paradigms. Our results also show that the heuristics were accurate for selecting appropriate context-sensitive analysis on the function level.

1.2 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 presents the background and general terminology used in this work. Chapter 3 discusses JSBAF and its instantiation. Chapter 4 describes our state-sensitive points-to algorithm. Chapter 5 discusses adaptive context-sensitive analysis. Chapter 6 presents the review of the related work. Finally, Chapter 7 summarizes the thesis and presents possible directions for future work.

Chapter 2

Background and Terminology

In this chapter, we first introduce the related background (i.e., the dynamic features of JavaScript and points-to analysis). We then present the terminology (i.e., blended analysis) and define the key concepts (i.e., JavaScript object-reference state) in this work.

2.1 Background

2.1.1 JavaScript Dynamic Features

JavaScript is a dynamic programming language. Its dynamic features render the run-time behavior of JavaScript applications unpredictable, posing significant challenges for static dataflow analysis. We discuss several JavaScript features that may render static program analysis ineffective, as follows:

Reflective constructs and dynamic code generation. Reflection is a powerful mechanism of programming languages to observe the program behavior (e.g., object properties) and modify its execution at runtime. Although it is a useful programming feature, it is challenging for static analysis to accurately model reflective constructs because reflection often involves unpredictable program behavior. In particular, due to the popularity of JavaScript for developing event-driven client-side web applications, dynamic code generation via reflective constructs is frequently observed in JavaScript websites [58]. JavaScript supports several mechanisms that can generate executable code at runtime. For example, an `eval` function takes a string expression, generates the code, and then executes the result. The `eval` function is often difficult to understand and may result in a large piece of JavaScript code generated at runtime. Other forms of dynamic code generation mechanisms in JavaScript include `setTimeout` and `setInterval` functions that take string parameters. If static analysis models the reflective constructs inaccurately, the solution is imprecise and/or unsound.

Dynamic object behavior. JavaScript is a dynamically-typed object-oriented programming language, whose variables may be bound to different types at different program points. Instead of class-based inheritance, it supports prototype-based inheritance [41, 76] that results in a JavaScript object inheriting properties from a chain of prototype objects. Lacking the notion of class, it is difficult to summarize the type of a JavaScript object at a particular program point. In addition, a JavaScript object property can be added or updated via an indirect assignment statement (e.g., `x.p = y`) and deleted via a delete statement (e.g., `delete x.p`). This means a JavaScript object may exhibit different behaviors at different times during execution because the behavior of an object is defined by its properties. Moreover, object constructors may be polymorphic so that objects created by the same constructor may have distinct properties. These features make it difficult to accurately reason about the behavior of JavaScript objects via static analysis.

JavaScript functions. JavaScript supports first-class functions, which means that functions can be assigned to variables, stored in properties of objects or the elements of arrays, passed as arguments to functions, etc [16]. Also, a JavaScript function can be called without respecting the declared number of arguments; that is, functions may have any degree of variadicity that may exhibit different functionalities at runtime. It is hard to model variadic functions well statically. Furthermore, the `call` and `apply` methods of JavaScript functions may reflectively invoke the represented function. For example, `fun.apply(thisArg, [argsArray])` calls a function with a given `this` value and arguments provided as an array, which provides a flexible mechanism to invoke a variadic function. Reflective function invocations pose challenges for static analysis to accurately analyze the targets of these function calls.

Multi-paradigm programming. We have observed various programming paradigms are often used across JavaScript applications and within a JavaScript application (see Chapter 5), including object-oriented, functional and procedural programming. This multi-paradigm feature of JavaScript renders existing static analysis ineffective because static analysis algorithms are usually designed for accommodating programs exhibiting a specific programming paradigm.

2.1.2 Points-to Analysis

Dataflow analysis techniques are frequently used to build automated software tools. Among them, points-to analysis has been the fundamental and enabling analysis for most other inter-procedural dataflow analyses (e.g., taint analysis). Points-to analysis calculates the set of values a reference property or variable may have during execution. Its solution (i.e., points-to graph) represents a model of a program's heap. In a traditional points-to graph, there are two kinds of nodes (i.e., variable node v and abstract object node o) and two kinds of edges (i.e., points-to edge (v, o) and property reference edge $(\langle o_i, p \rangle, o_j)$).

Points-to analysis is interdependent on the inter-procedural program representation (i.e.,

call graph). A call graph represents the calling structure of a program. Specifically, a call graph node represents a function and a call graph edge represents the calling relationship between two functions. For modern object-oriented programming languages (e.g., Java and JavaScript), the call graph construction is often interleaved with points-to analysis because call graph construction affects the points-to solution and vice versa.

There are multiple dimensions that affect the precision and performance of points-to analysis and call graph construction. There have been thorough literature reviews on this topic [26, 60, 71, 67]. Here we discuss the subset of analysis aspects closely related to this work:

Object representation. There are two common object representation choices for points-to analysis [60]: (i) an analysis uses an abstract object to represent all possible instantiations of a class for a class-based language, and (ii) all objects created by the same allocation site are represented by an abstract object. Due to the lack of notion of class in JavaScript, the latter is a more popular choice for analyzing JavaScript applications (e.g., [30, 72]). Nevertheless, because a JavaScript object constructor may be polymorphic, creating objects with different possible behaviors, a more accurate object representation accommodating the JavaScript object model needs to be explored.

Flow sensitivity. A flow-sensitive analysis follows the execution order of statements in a program; a flow-sensitive analysis can perform strong updates, but a flow-insensitive one cannot. An intra-procedural program representation (e.g., control flow graph [2]) enables flow-sensitive analysis. In a control flow graph (CFG), each node represents a basic block (i.e., straight-line piece of code without branch). The header (i.e., first instruction) of a basic block can be (i) entry point of a function, (ii) target instruction of a branch, and (iii) instruction that immediately follows branch.

Although a flow-sensitive analysis is more precise than a flow-insensitive analysis, the scalability problem has resulted in several JavaScript analyses abandoning the use of flow sensitivity (e.g., [72]). However, the nature of JavaScript objects (i.e., property addition, update and deletion) suggests the favor of flow sensitivity. Some recent JavaScript analyses are flow-sensitive (e.g., [30, 4, 53]). It remains an open question to find the balance between performance and precision for JavaScript flow-sensitive analysis.

Context sensitivity. Context sensitivity is a general technique to achieve more precise program analysis by distinguishing between calls to a specific function. Historically, call-strings and functional are the two approaches to enable context sensitivity in an analysis [65].

A call-strings approach distinguishes function calls using information on the call stack. The most widely known call-strings approach is call-site-sensitive (k -CFA) analysis [66]. A k -call-site sensitive analysis uses a sequence of the top k call sites on the call stack as the context element. k is a parameter that determines the maximum length of the call string maintained to adjust the precision and performance of call-site-sensitive analysis. 1-call-site-sensitive analysis separately analyzes each different call site of a function. Intuitively in the

code example below, 1-call-site-sensitive analysis will analyze function `foo` in two calling contexts L1 and L2, such that local variables (including parameters) of `foo` will be analyzed independently for each context element.

```
L1:  x.foo(p1, p3);  
L2:  y.foo(p2, p4);
```

A functional approach distinguishes function calls using information about the computation state at the call. Object sensitivity analyzes a function separately for each of the abstract object names on which this function may be invoked [47]. Milanova *et al.* presented object sensitivity as a parameterized k -object-sensitive analysis, where k denotes the maximum sequence of allocation sites to represent an object name. 1-object-sensitive analysis separately analyzes a function for each of its receiver objects with a different allocation site. Intuitively in the code example above, 1-object-sensitive analysis will analyze function `foo` separately if `x` and/or `y` may point to different abstract objects. If `x` points to objects O_1 and O_2 , while `y` points to object O_3 , 1-object-sensitive analysis will analyze function `foo` for three context elements differentiated as O_1 , O_2 and O_3 .

Other functional approaches presented use the computation state of the parameter instead of the receiver object as a context element. The Cartesian Product Algorithm (CPA) uses tuples of parameter types as a context element for Self [1]. The context-sensitive analysis presented by Sridharan *et al.*, designed specifically for JavaScript programs, analyzes a function separately using the values of a parameter `p` if `p` is used as the property name in a dynamic property access (e.g., `v[p]`) [72]. Andreasen and Møller also designed a context-sensitive analysis for JavaScript using the parameter whose abstract value is a concrete string or a single object address as a context element [4]. To capture these approaches, we define a simplified, parameterized i th-parameter-sensitive analysis, where i means we use the abstract object corresponding to the i th parameter as a context element. Intuitively in the code example above, 1st-parameter-sensitive analysis will analyze function `foo` separately if `p1` and/or `p2` may point to different abstract objects. If `p1` points to object O_4 , while `p2` points to object O_4 and O_5 , 1st-parameter-sensitive analysis will analyze function `foo` for two context elements distinguished as O_4 and O_5 .

Although context sensitivity is a topic that has been thoroughly studied, an accurate context-sensitive analysis for JavaScript still remains to be investigated.

2.2 Terminology

2.2.1 Blended Analysis

Dufour *et al.* presented the blended analysis paradigm for performance diagnosis of framework-intensive Java programs [13, 14]. In the blended analysis paradigm, dynamic analysis is used to obtain the calling structure of a particular execution of interest and then a static analysis is performed in that calling structure to obtain more detailed semantic information relevant for performance understanding. Specifically, this analysis dynamically collected one problematic Java execution and performed a static escape analysis¹ on its call graph. Java features such as reflective calls and dynamically loaded classes were recorded by the dynamic analysis, allowing more precise modeling than by pure static analysis.²

Pruning was an optimization technique applied in Java blended analysis to each executed method’s control flow graph in order to approximate a specialized version of the code executed during a particular call. Pruning was very effective in removing approximately 30% of the statements from Java functions [14]. Essentially, unexecuted statements in functions were removed using run-time information. The analysis noticed which function calls and object creation sites were not recorded and used control dependence information to prune other unexecuted statements.

Overall, blended analysis is a tightly coupled dynamic and static analysis paradigm. It focuses a static analysis on a dynamic calling structure collected at runtime, and further refines the static analysis using additional information collected by a lightweight dynamic analysis.

2.2.2 JavaScript Object-Reference State

Recall that JavaScript is a dynamically typed programming language whose object behavior can change as object properties are added or deleted at runtime. In strongly typed programming languages, the notion of *type* is used to abstract the possible behavior of an object (e.g., the class of an object in Java) [64]; however, in dynamically typed languages, the type of an object can change during execution. In order to avoid confusion, we call the type of a JavaScript object its *obj-ref state*.³

Definition 1. The obj-ref state at a program point denotes all of its accessible

¹Escape analysis is a static analysis that determines whether the lifetime of data exceeds its static scope [8].

²When discussing in the context of blended analysis, we use the term pure static analysis referring to an analysis based on monotone dataflow frameworks [45].

³This general notion can be used for other dynamic languages and is related to structured typing for strongly typed languages [64].

properties and their non-primitive values.

The accessible properties of an object conform to the property lookup mechanism implemented in JavaScript. Every JavaScript object includes an internal reference to its prototype object from which it inherits non-local properties. A JavaScript object may have a sequence of prototype objects (i.e., a prototype chain) whose properties it can inherit. When reading a property p of an object o , the JavaScript runtime checks the local properties of o to see if o has a property named p . If not, the JavaScript runtime checks to see if the prototype object of o has a property named p , continuing to check along the prototype chain from object to object until the property is found (or not) [16].

Definition 2. *State-update statements* are: (i) property write statement (i.e., $x.p = y$ or $x[p] = y$), (ii) property delete statement (i.e., `delete x.p` or `delete x[p]`), and (iii) an invocation that directly or indirectly results in execution of (1) and/or (2).

The state-update statements are the set of statements in JavaScript that may affect the obj-ref state. In Figure 2.1, we illustrate the obj-ref state with an example that shows the objects connected to O_1 at a program point. The local properties of object O_1 are named p_1 and p_2 and O_4 is its prototype object. O_7 is visible from O_1 by accessing $O_1.p_4$ while O_6 is not visible from O_1 by accessing $O_1.p_2$ because a local property named p_2 exists for O_1 . To sum up, the shaded nodes (i.e., O_6 and O_9) are not accessible from O_1 and the unshaded nodes constitute O_1 's reference state.

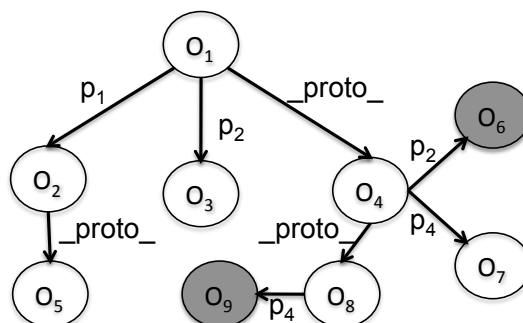


Figure 2.1: obj-ref state example

Chapter 3

JavaScript Blended Analysis Framework

Recall that blended analysis paradigm was designed by Dufour *et al.* [13, 14] for performance diagnosis of framework-intensive Java programs, capturing dynamic aspects of Java such as reflective calls and dynamically loaded classes. Because JavaScript applications exhibit more dynamic behavior than Java programs, it is natural to apply a dynamic analysis to accommodate its dynamic features. We have designed the JavaScript Blended Analysis Framework (JSBAF), a flexible and general-purpose analysis framework that tightly couples dynamic and static analyses. We analyze multiple executions rather than a single one, but the overall algorithm workflow and pruning are both utilized, albeit to handle a more general set of dynamic language features in JavaScript. In this chapter, we propose the detailed design of JSBAF. We then present the first instantiation of JSBAF (i.e., blended taint analysis) for finding security vulnerabilities in JavaScript websites and discuss the experimental results.¹

3.1 JSBAF Design

3.1.1 Framework Overview

JSBAF was designed to judiciously combine dynamic and static analyses, accounting for the effects of dynamic features not seen by pure static analysis. It aims to offer an efficient methodology to obtain practical and accurate analysis solution of JavaScript applications.

Figure 3.1 illustrates an overview of JSBAF that can be applied in the following software testing scenario for a JavaScript program. We assume the presence of a test suite of the JavaScript program as input. Such a test suite may be obtained in the following ways: (i)

¹Part of the contents presented in this chapter was published in [77].

existing tests of JavaScript programs (e.g., the unit and integration test suite of jQuery library), (ii) tests created by automated test generation tools of JavaScript (e.g., *JSEFT* [48] and *Artemis* [5]), and (iii) manually explored tests of JavaScript applications (see Section 3.2). The *Test Selector* chooses a subset of the tests that offer good coverage of the program to obtain an accurate analysis solution at lower cost than using all the tests. Note that despite that fact that each test is likely to cover a different program path, this difference may not be distinguishable depending on the dynamic information collected by JSBAF. The *Execution Collector* then gathers run-time information (e.g., function calls and dynamically generated code) by executing each selected test. The *Static Infrastructure* performs static dataflow analysis on the program represented by the observed calling structure and takes into account other run-time information from the dynamic trace. The *Solution Integrator* combines dataflow solutions from different dynamic traces into a program solution, and decides if there are more traces to analyze.

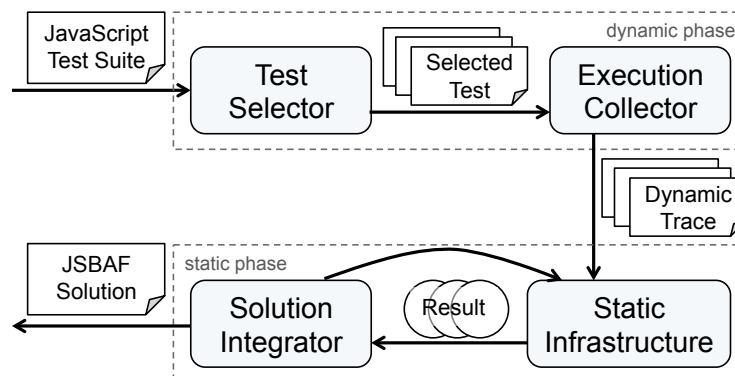


Figure 3.1: JavaScript Blended Analysis Framework

By design, JSBAF in Figure 3.1 performs static analysis on each dynamic trace and combines the results. An alternative design of the static phase would be to combine the dynamic information from all the executions (i.e., *Trace Integrator*) and then apply static analysis once to the combined executions (Figure 3.2). Intuitively, this alternative approach is straightforward and may save the cost of multiple static analyses. It also may lose precision by introducing possibly infeasible inter-procedural paths when multiple calling structures are combined. In our instantiations of JSBAF (Section 3.2 and Chapter 4), we have chosen the design of JSBAF shown in Figure 3.1.

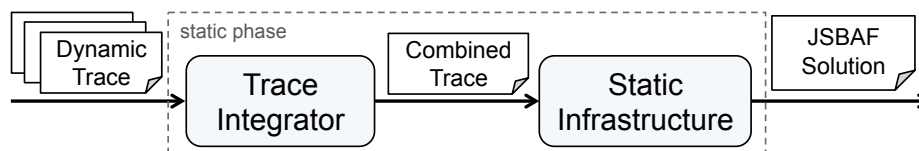


Figure 3.2: An alternative design of JSBAF static phase

In addition, JSBAF is a flexible framework in that techniques and heuristics applied to

individual components can be substituted. For example, Figure 3.3 shows various instrumentation choices for the Execution Collector. Gathering more dynamic information (e.g., sequence of all executed instructions) enables a more accurate representation of an execution, but it also results in additional cost for the Execution Collector. The Execution Collector is designed to flexibly apply instrumentation heuristics depending on the requirements and budget of the analysis. In our implementations, we have adapted a relatively lightweight Execution Collector that collects the dynamically generated code, function calls and object allocations. Similarly for the Static Infrastructure, different static analysis choices (e.g., flow and context sensitivity) that affect the precision and performance can be applied. JSBAF is also general-purpose in that by replacing the Static Infrastructure, we can change the specific analysis applied to the JavaScript program. We will demonstrate the flexibility and generality of JSBAF by showing its instantiations that apply different flow- and context-sensitive policies as well as different analysis clients (Section 3.2 and Chapter 4).



Figure 3.3: Instrumentation heuristics and cost model of Execution Collector

3.1.2 JSBAF for Dynamic Features

We now discuss the benefits of JSBAF on handling several dynamic features of JavaScript.

Eval. Pure static analysis can analyze JavaScript source code that is statically visible; however at runtime, invocations of reflective constructs such as `eval` may result in new JavaScript code being generated and executed. This generated code is difficult to model accurately via static analysis because `eval` parameters may contain complicated string values being set at runtime. Because inaccurate models of reflective constructs often result in intractable (i.e., unscalable and/or overly approximate) analyses, most JavaScript static analyses (e.g., [30, 21, 72]) give up analysis soundness by ignoring the dynamic code generation mechanisms. Nevertheless, there have been efforts on transforming `eval` functions to other JavaScript constructs and thus enabling static analysis on the transformed code. Jensen *et al.* presented a static analysis to eliminate specific calls to `eval` [28] and Meawad *et al.* proposed a semi-automated approach via classification techniques to replace `eval` call sites with safer JavaScript idioms [46]. However, there still are inevitable cases where these approaches cannot transform some calls to `eval` (e.g., Jensen *et al.* reported that 11 out of 44 `eval` call sites in their experiments could not be successfully transformed by their approach [28]).

Figure 3.4 shows an example of `eval` functions from *xing.com*. Jensen *et al.* reported that this piece of code containing multiple `eval` calls not resolvable by their approach [28]. In

```

1  for (n = 1; n < 20; n++) {
2      xe = "s.prop" + n + "=myUe(s.prop" + n + ")";
3      ex = "s.eVar" + n
4          + "=myCp(s.prop" + n + ", 'D=c" + n + "')";
5      to = "typeof(s.prop" + n + ")";
6      if (eval(to) != "undefined") {
7          eval(xe);
8          eval(ex)
9      }
10 }

```

Figure 3.4: JavaScript `eval` example

JSBAF, `eval` calls can be monitored by the Execution Collector which gathers any code generated thusly, making it available during analysis of the dynamic calling structure. For the code example in Figure 3.4, the Execution Collector observed 19 calls to `eval(to)` (line 6) with the actual code recorded as

`typeof(s.propX)` where $X \in \{1, 2, \dots, 19\}$.

The Execution Collector also observed 2 calls to `eval(xe)` (line 7) and `eval(ex)` (line 8) with the actual code

`s.propY=myUe(s.propY)` and `s.eVarY=myCp(s.propY, 'D=cY')` where $Y \in \{1, 2\}$.

The Static Infrastructure then analyzes the code including the effects of `eval` calls because the above generated code is visible for the Static Infrastructure and the dynamic calling structure contains calls to functions `myUe` and `myCp`.

In addition to `eval`, other JavaScript mechanisms that generate code at runtime such as `setTimeout` and `setInterval` functions can also be collected and analyzed by JSBAF.

Function Variadicity. Function variadicity occurs when a function can be called with an arbitrary number of arguments, regardless of its declaration. If fewer arguments are provided than in the declaration, the values of the rest of the declared arguments are set to be *undefined*. If more arguments are provided than in the declaration, the arguments can be accessed through an associated *arguments* object. Sometimes, branch conditions within a function can be differentiated by its number of arguments; thus, the function may exhibit different behaviors.

Figure 3.5 shows a shortened version of the important `extend` function of jQuery v1.11.2. Overall, this function is designed to extend the properties of the `target` object, whose reference is dependent on the number as well as the types of arguments passed to the function. If

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

Figure 3.5: Simplified jQuery extend function

one argument is passed, the `this` object is assigned to `target` at line 9; thus, the functionalities of jQuery are extended in lines 12 to 18. Otherwise, the values of the `target` object may also be `arguments[0]` (line 2) and `arguments[1]` (line 5). Static analysis needed model function variadicity to accurately reason about the references of the `target` object so that it can produce precise results on this function. Unfortunately, pure static analyses for JavaScript normally ignore this feature because (i) the actual number of arguments provided during the call may only be known at runtime (e.g., calling a variadic function using the reflective `apply` function may result in the length and values of the arguments not knowable statically), and (ii) specific techniques need to be applied to separate the behaviors of a variadic function. In contrast, the Execution Collector can capture the actual number of arguments for each call so that the dynamic calling structure can contain separate nodes for instances of the same function called with different numbers of arguments, introducing context sensitivity. Executing a simple program that loads jQuery library, we observed 29, 2 and 4 calls to the `extend` function with 1, 2 and 3 arguments, respectively. In addition, when branches of a variadic function are determined by its number of arguments, using the concrete value of `arguments.length` as a context element, the Static Infrastructure can prune the unexecuted branches from the analysis and result in a more accurate approximation of the code in the function variant.

Constructor Polymorphism. JavaScript object constructors may be polymorphic so that objects created by the same constructor may have distinct properties. For the example shown

in Figure 3.6, JavaScript objects created by this constructor `X` may have a local property `p` or `q` depending on the value of the parameter passed. In an empirical study, Richards *et al.* reported the existence of polymorphic constructors in JavaScript web applications [59]. Recall that a static analysis usually represents all objects created by the same allocation site by an abstract object and assumes all possible properties created by the constructor (e.g., both `p` and `q` by the constructor `X` in Figure 3.6). This approximation is inaccurate to summarize JavaScript objects because objects with different properties often exhibit different behaviors. The Execution Collector can collect the run-time information of properties associated with a created object by (i) instrumenting the property write statements, or (ii) instrumenting the object allocations and pruning unexecuted statements. The Static Infrastructure then uses this dynamic information to distinguish objects created by the same constructor with distinct properties (see Chapter 4).

```
1 function X(b) {  
2     if(b) { this.p = new Y(); }  
3     else this.q = new Z();  
4 }
```

Figure 3.6: JavaScript polymorphic constructor example

Dynamic Property Accesses. JavaScript object properties can be accessed as associative arrays, where the names of the properties are expressed as strings. In addition, object properties are not declared and can be changed at any program point. Dynamic property accesses have been identified by Sridharan *et al.* [72] and Park *et al.* [53] as an important issue that causes the inaccuracy of JavaScript static analyses. Line 15 in Figure 3.5 demonstrates a use case of dynamic property accesses. For each iteration of the embedded loops, the property `name` of `target` is written as the value of the property `name` of `options`. It is difficult for static analyze to resolve and distinguish the property names for the dynamic property accesses because the values are associated with function arguments and loops. The Execution Collector can instrument the concrete names of property accesses as well as the loop conditions and the Static Infrastructure can then perform specialized static analysis on these concrete values.

Other Features. In addition to the above important dynamic features, there are other features in JavaScript that require special treatment. Although some features can be modeled by a static analysis (e.g., lexical scoping and type-based dynamic dispatch), JSBAF may further improve the precision of these static models.

For example, Figure 3.7 illustrates an example of dynamic dispatch based on the object constructor. Because of the nature of dynamic typing, the variable `v` can refer to objects whose constructors are unrelated by inheritance (e.g., the objects created by constructors `A` and `B`). The actual function being called in line 6 depends on the constructors of `v`, which is determined by the predicate in line 1. Pure static analysis normally assumes that either branch in this code can be taken, and makes a conservative approximation that `v` can be

```
1  if (b) {  
2    v = new A();  
3  else  
4    v = new B();  
5  }  
6  v.bar();
```

Figure 3.7: JavaScript type-based dynamic dispatch example

created by either A or B, making the target in line 6 ambiguous. In JSBAF, the Execution Collector can collect the functions that are called and constructors that are used to create JavaScript objects. The Static Infrastructure can use pruning to eliminate the code that was not executed to preserve the dynamic information. In case of Figure 3.7, one of the branches (e.g., line 4) will be pruned if *v* is created only by A so that the Static Infrastructure knows the actual type of *v* in line 6. Thus JSBAF performs static analysis on accurate presentations of the executions.

3.1.3 Soundness of JSBAF

Overall, JSBAF presents an unsound [45] analysis framework for JavaScript. In blended analysis, the program input to the Static Infrastructure is an over-approximation of the actually executed code and the Static Infrastructure, instead of reasoning over all possible behavior of the program, analyzes the program representations collected by the Execution Collector. Specifically, JSBAF produces overly-approximated results (i.e., solution of the Static Infrastructure) on under approximations of the whole program behavior (i.e., traces collected by the Execution Collector). In this section, we discuss the theoretical relationship between the solutions of blended and pure static analyses, in terms of soundness.

A sound pure static analysis is expected to approximate all possible behavior of a program, including all language constructs. Due to the conservative approximations, a pure static analysis solution may also contain the behavior that cannot be exhibited in any execution. The *precision* of an analysis measures if it produces many of these spurious results. In addition, a pure static analysis is often performed under limited time and memory resources. The *performance* of an analysis measures if it produces results under reasonable resources. An analysis is practical if it achieves good precision as well as performance analyzing the target programs. For example, the green rectangle in Figure 3.8 represents the solution of a sound pure static analysis and the region in the dashed circle represents all possible behavior of the program (i.e., true positives). As shown in Figure 3.8, a practical sound analysis should cover all the true positives and produce small number of false positives (i.e., the green region outside the dashed circle).

Despite the fact that pure static analysis is often expected to produce whole-program sound

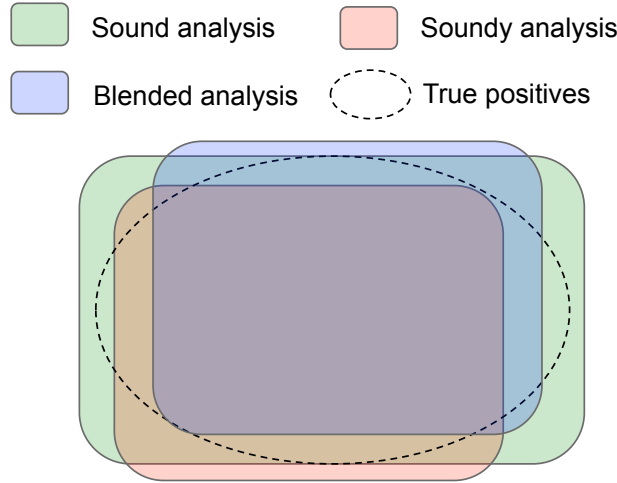


Figure 3.8: Soundness of blended analysis

results, there is no pure static analysis tool that is actually sound [42]. Specific language features (e.g., dynamic code generation) are difficult to model statically and handling these features statically often results in very imprecise solution, rendering the analysis impractical. Thus, pure static analysis makes unsound assumptions over these language constructs. Livshits *et al.* presented a new terminology, *soundy*, for such pure static analysis [42]. For JavaScript, pure static analysis makes unsound choices over multiple language constructs. To the best of our knowledge, none of the existing pure static analyses fully handles the feature of dynamic code generation in JavaScript. Some other language constructs such as `call` and `apply` functions have also been omitted from some pure static analyses. Therefore, all state-of-the-art pure static analyses for JavaScript are soundy analyses. We now reason about how the solutions of soundy pure static analysis and blended analysis of JavaScript compare in theory.

We first consider the call graphs constructed and used by both analyses. A soundy pure static analysis of JavaScript uses a call graph $\mathbf{CG}_{\text{soundy}}$, a representation of a set of functions F_{soundy} and their possible calling relations recognized from the statically accessible code. $\mathbf{CG}_{\text{soundy}}$, representing a set of possible executions, ignores specific language features. If the `eval` construct was omitted by the analysis, there may exist additional functions and/or calling relations not represented by $\mathbf{CG}_{\text{soundy}}$. JSBAF, on the other hand, captures profiling information needed to model dynamic features, but may not explore all executable paths in a JavaScript program. JSBAF analyzes multiple executions of a program. For each execution \mathbf{e} , the set of functions observed is $F_{\mathbf{e}}$. $F_{\mathbf{e}}$ may consist of two subsets: (i) $F_{\mathbf{e}(\text{soundy})}$, a set of functions visible from the soundy pure static analysis, $F_{\mathbf{e}(\text{soundy})} \subseteq F_{\text{soundy}}$, and (ii) $F_{\mathbf{e}(\text{dyn})}$, a set of functions profiled during execution of \mathbf{e} , whose existence is due to the language features not handled by the soundy analysis. The Static Infrastructure then constructed the call graph, $\mathbf{CG}_{\mathbf{e}}$, of the execution \mathbf{e} collected by the Execution Collector. Therefore, $\mathbf{CG}_{\mathbf{e}}$ is a conservative approximation of all the inter-procedural invocations among $F_{\mathbf{e}}$ that occur when

e is executed. Both $\text{CG}_{\text{soundy}}$ and CG_e may possibly introduce unexecutable inter- and intra-procedural paths in the JavaScript program. Note that CG_e is not necessarily a subgraph of $\text{CG}_{\text{soundy}}$ because of the nodes (i.e., $F_{e(\text{dyn})}$) and/or calling relations introduced by features not handled by the soundy analysis. Also, JSBAF may not explore all nodes and/or edges in $\text{CG}_{\text{soundy}}$.

The pure static analysis is sound with respect to the program except for the language constructs not handled. For blended analysis, the Static Infrastructure treats each execution as an entire program, building a conservative approximation of the calling structure and applying a sound static analysis to this representation. Therefore, blended analysis of a single execution is sound, producing no false negatives for the portion of the program it analyzes. Since JSBAF integrates the solutions on multiple executions to form the entire analysis solution and the solution of each execution is sound, therefore JSBAF is sound with respect to the observed executions.

In Figure 3.8, we describe the general relationship between a soundy analysis solution (i.e., the pink rectangle) and a blended analysis solution (i.e., the blue rectangle) for the same JavaScript program. Their intersection includes the part of the solution due to program constructs modeled by the soundy analysis and within those executions observed by JSBAF. There may be a set of results reported by soundy analysis that blended analysis does not calculate because (i) it does not explore every executable path in the program, and (ii) false positives introduced by the over-approximation of soundy analysis may be avoided by the more precise dynamic call graphs used by JSBAF. The blended analysis solution may also contain results missed by a soundy analysis because of the language constructs not modeled by soundy analysis, but observed and therefore modeled by blended analysis. Blended analysis may contain additional false positives because of the approximation in analysis of dynamic constructs. The goal of JSBAF is to retain most of the true positives found by a soundy pure static analysis while eliminating some false positives, and to discover more true positives by analyzing the dynamic features of JavaScript.²

3.2 An Instantiation: Blended Taint Analysis

Security is one of the critical issues associated with JavaScript web applications. Given the ubiquity of JavaScript, it is crucial to discover security vulnerabilities possibly introduced by use of its dynamic features. Many security problems can be formalized as information flow problems [12] which seek to preserve the integrity of data (i.e., not allow untrusted values to affect a sensitive value or operation) and confidentiality of data (i.e., keep sensitive values from being observed from outside the computation). Taint analysis detects flows of data that violate program integrity. To demonstrate the practicality of JSBAF, we have instantiated a blended taint analysis and performed empirical evaluation of its results.

²In Section 3.2, we use an instantiation of JSBAF, blended taint analysis, to demonstrate its practicality.

3.2.1 Blended Taint Analysis

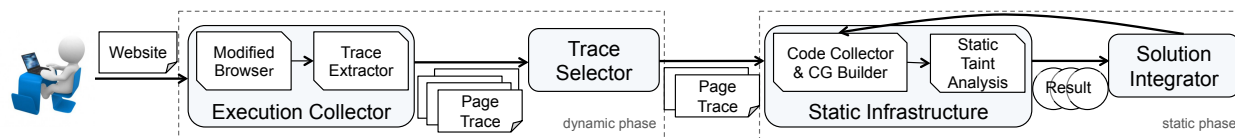


Figure 3.9: Blended taint analysis for JavaScript web applications

Figure 3.9 presents the workflow of our blended taint analysis. The work of the dynamic and static phases of blended taint analysis is patterned directly after the work of these phases in JSBAF. In the dynamic phase, a user interacts with a website manually exploring its functionalities, using a browser that instruments JavaScript operations. Traces of each webpage consisting of recorded function calls, constructors of created objects, and dynamically generated/loaded code that is not statically visible, are gathered by the Execution Collector. The Trace Selector selects a subset of the page traces that cover the behavior of the executed program well. In the static phase, the Code Collector identifies the JavaScript code that was executed, including both statically visible and invisible code. The Call Graph Builder creates a call graph from the recorded function calls and other collected function-specific information as node annotations. *Static Taint Analysis* is applied to the program represented by the call graph. The Solution Integrator combines solutions from different page traces into a single solution for that webpage. The final solution of a blended taint analysis is a set of source-sink pairs; these represent untrusted data (i.e., *sources*) which can reach sensitive operations (i.e., *sinks*).

Dynamic phase. Our Execution Collector relies on a specialized version of *TracingSafari*, an instrumented version of WebKit³ JavaScript engine developed for characterizing the dynamic behavior of JavaScript programs [59]. *TracingSafari* records operations including function calls, property adds and deletes, etc. It also collects events such as source file loads. Heavyweight instrumentation (e.g., property accesses and conditional predicates) of web applications often result in performance degradation of the browser. In the scenario of blended taint analysis, a user interacts with the websites to collect the page traces and avoiding significant performance overhead is desirable. We have designed a lightweight instrumentation by modifying *TracingSafari*.

Specifically, to assure the security of a website, the user explores webpages from the same domain. Execution of a website may involve code on several different webpages. The sequence of JavaScript instructions collected during an execution is decomposed into *page traces*; each trace is a consecutive sequence of Javascript instructions from the same webpage. A page

³<http://www.webkit.org>

trace consists of a dynamic call tree, recorded object creations, loaded program including the dynamically generated code. The only instructions recorded are function calls and object allocations.⁴ The Execution Collector also captures the actual number of arguments for each call to precisely model variadic functions. The Trace Extractor builds the set of page traces corresponding to each webpage collected from a set of executions.

There may be traces of the same page that are redundant on a large portion of their behavior. Therefore, performing static analysis on these page traces may result in similar solutions, while greatly increase the analysis cost. To avoid this situation, Trace Selector is designed to minimize the number of traces analyzed, while covering most of the observed program behavior. The trace selection algorithm takes all traces from one webpage collected by the Execution Collector and a threshold (i.e., a value between 0 and 1 that can be adjusted by the user of JSBAF) as input and works as follows. It starts with randomly selecting a trace from all the traces and then uses the criterion `dist` to iteratively select from the rest of the traces. Specifically, the `dist` score of each candidate trace is calculated against the selected traces. The candidate trace with the highest `dist` score is selected. This algorithm terminates when no more candidate traces reach the threshold or there are no longer any traces to select.

The core of the trace selection algorithm is to calculate the `dist` criteria of the candidate trace. This value is comprised of three factors: (i) function coverage (i.e., `distfunc`), (ii) object constructor coverage (i.e., `distobj`), and (iii) dynamically generated code coverage (i.e., `distdyn`). Covering more functions and observed object constructors explores more code and program paths, while covering the dynamically generated code expands the capability of blended analysis to analyze the program constructs that are normally omitted by a pure static analysis. Each factor is normalized to fall between 0 and 1. The value of `dist` is calculated via a linear combination of these factors to emphasize using traces that cover more functions, explore different object allocations and contain as much as possible of the dynamically generated code encountered, as follows:

$$\text{dist} = 0.5 \times \text{dist}_{\text{dyn}} + 0.4 \times \text{dist}_{\text{func}} + 0.1 \times \text{dist}_{\text{obj}}$$

These weights can be adjusted based on analysis requirements and budget to maintain the balance between performance and accuracy. We demonstrate the above choice of weights has produced reasonable analysis results in Section 3.2.2. Note that this heuristic is not specific to JavaScript but can be used for other dynamic languages with similar characteristics.

Static phase. The Static Infrastructure of our blended taint analysis for JavaScript was built on the *IBM T. J. Watson Libraries for Analysis (WALA)* static analysis framework⁵ that includes a JavaScript front-end. *WALA* parses JavaScript source code from a webpage

⁴In Sections 3.1.1 and 3.1.2, we discussed that other instructions (e.g., property accesses) may be collected.

⁵<http://wala.sourceforge.net>

producing an abstract syntax tree (AST) and translates the AST into the *WALA* intermediate form. This infrastructure statically models several language features of JavaScript such as lexical scoping and reflective property accesses [21].

Our Call Graph Builder generates the context-sensitive call graph of each page trace. As discussed in Section 3.1.2, we use the number of arguments (i.e., `argument.length`) as a context element in the call graph. Therefore, variadic functions have multiple nodes in our call graph with different contexts. The Call Graph Builder applies pruning to the code of all functions by annotating the unexecuted statements. In addition to the pruning technique presented by Dufour *et al.* [14], when branches of functions are determined by the value of `arguments.length`, we use that value to prune the statements on unexecuted branches to provide a more accurate approximation of variadic functions. The Code Collector deals with the dynamically generated code that is omitted by *WALA*, but collected in the page trace. Synthetic functions are generated for these dynamic features (e.g., `eval`) and the targets are the distinct dynamically generated code in the trace.

We have implemented a static taint algorithm to detect integrity violations in four steps:

- (i) A points-to analysis for JavaScript [72] is performed to obtain aliases of objects in the program. We modified the implementation in that when the analysis looks for the targets a function invocation, it seeks the the dynamic call graph built by our Call Graph Builder instead of constructing the call graph on-the-fly with the static algorithm in *WALA*.
- (ii) Sources and sinks are predefined and automatically identified in the program. A data source is called *tainted* when an untrusted third party has control of its value. JavaScript functions from untrusted third party code are considered to be sources; the variables created in these functions or whose values are returned by calling these functions are marked as tainted.

We consider two sets of objects to be *sensitive*. Properties of objects that hold important browser/user information are sensitive. In the implementation, we reused the same set of properties by Nentwich *et al.* [49]. Every variable in a statement that writes those properties is marked as a sink (e.g., the URL property of the `document` object). A persistent security vulnerability can happen if untrusted data is saved by the server. Therefore, parameters of functions which are sent to the server are sensitive (e.g., the parameter of `xmlhttprequest.send()`). These parameters are marked as sinks.

- (iii) A call graph reachability analysis is executed to filter out any node that is not on a direct call path from a function containing tainted source(s) to a function containing sink(s); the remaining nodes are candidates for taint propagation.

(iv) Taint propagation. An inter-procedural traversal of the call graph is performed from each source through candidate nodes to any reachable sink. At each encountered candidate function, an intra-procedural data dependence analysis is applied to track the tainted variables into candidate calls. The possible effects of calls to non-candidate functions are approximated: if one argument of the call is tainted, we assume all the arguments are tainted

as an optimization to avoid analysis of these functions. Call cycles are handled by fixed point iteration (i.e., when the variables of the recursive functions achieve fixed tainted states).

3.2.2 Evaluation

We conducted experiments comparing the practicality of blended taint analysis to two pure static taint analyses on popular JavaScript websites.

Pure static taint analyses. We implemented a pure static analysis on top of *WALA* that performs the four step algorithm presented in Section 3.2.1. Instead of using the dynamic call graph, the pure static taint analysis performs the standard on-the-fly call graph construction during points-to analysis [72].

The static infrastructure in *WALA* omits the semantics of `eval`. To increase the capability of pure static taint analysis, we added a naive model of `eval`. JavaScript variables that serve as the `eval` function parameters are considered as accessible in the `eval` calls. In pure static analysis, the `eval` functions are conservatively marked as additional sinks because the generated code is not visible without a complicated analysis of the `eval` functions.

In the experiment, we ran pure static taint analysis in two configurations. JavaScript libraries are frequently used in websites. The first configuration, Static Taint⁺, analyzes the code directly extracted from the webpages and any reachable library functions. The second configuration, Static Taint⁻ analyzes only the JavaScript code extracted from the webpages. We designed these two configurations in our experiments because some JavaScript libraries (e.g., jQuery) have posed significant challenges to pure static analysis in terms of scalability and precision. Static Taint⁻ ignores the JavaScript libraries and may still detect taint violations originating from the application code.

Hypotheses. Our experiments explore the following hypotheses: (i) blended taint analysis can scale to real-world JavaScript programs, and (ii) blended taint analysis is more accurate than pure static taint analysis, capable of discovering more security violations and eliminating some false alarms.

Benchmarks. The experiments were conducted with benchmarks consisting of 12 websites that are among the top 25 most popular sites on *www.alexa.com* at the time of evaluation (i.e., December 2012). A user of our Execution Collector, who had no knowledge of JSBAF and blended taint analysis, was instructed to explore different functionalities on webpages. Table 3.1 shows the statistics of the collected benchmarks. Each benchmark is formed from the user’s interaction with a website. A profiled interaction consists of individual traces, each containing a sequence of JavaScript instructions from a single webpage. The set of traces corresponding to the same webpage comprises a JavaScript program for blended taint analysis. The second column in Table 3.1 shows the number of webpages executed at each website and then analyzed. The third column represents the total number of page traces collected for each website.

Website	No. of pages	No. of traces
facebook	27	62
google	22	55
youtube	15	30
yahoo	30	69
wikipedia	27	65
amazon	9	13
twitter	32	53
blogspot	9	17
linkedin	32	54
msn	13	21
ebay	40	72
bing	7	14
totals	263	525

Table 3.1: Benchmarks

The experimental results were obtained on a 2.53 GHz Intel Core 2 Duo MacBook Pro with 4 GB memory running the Mac OS X 10.5 operating system.

Blended taint analysis results. Table 3.2 presents the time performance of the static phase of blended taint analysis, Static Taint⁺ and Static Taint⁻, each run under a limited time budget of 10 minutes. Columns 2 and 4 present the number of webpages that could not be analyzed within the time budget by Static Taint⁺ and Static Taint⁻, respectively. Static Taint⁺ was able to fully analyze two websites (i.e., *linkedin* and *bing*). For some sites, Static Taint⁺ did not scale on all/most webpages (e.g., *yahoo* and *amazon*). Static Taint⁻, which does not analyze library code, was capable of analyzing most application code from JavaScript webpages. To sum up, Static Taint⁺ timed out on 108 out of 263 webpages and Static Taint⁻ timed out on 12 out of 263 webpages. Blended taint analysis, on the other hand, was able to finish analyzing all selected page traces within the time budget.

Columns 3, 5 and 6 show the analysis time of each website averaged over those webpages that were not timed out for Static Taint⁺, Static Taint⁻ and blended taint analysis, respectively. The time cost of the static phase of blended taint analysis on a webpage is the total time of multiple static taint algorithms applied to each trace. In Table 3.2 the average analysis time of blended taint analysis exceeds that of Static Taint⁻ on all websites but *google*. This is mainly caused by the fact that (i) Static Taint⁻ ignores JavaScript libraries, and (ii) the same portion of JavaScript code may be analyzed duplicately in multiple traces.

The time cost of blended taint analysis is dominated by its static phase. Nevertheless, performance of the Execution Collector is still critical to the experience of the user of our blended taint analysis. We compared the performance of our modified *TracingSafari* and an original Safari running *JSBench* [57], a JavaScript benchmark generated from real websites,

Website	Static Taint ⁺		Static Taint ⁻		Static phase of blended taint
	No. of pages timed out	Average time (sec.)	No. of pages timed out	Average time (sec.)	Average time (sec.)
facebook	14	28.5	0	19.2	29.4
google	13	39.2	1	22.4	14.2
youtube	10	57.3	2	13.9	37.4
yahoo	24	33.0	3	12.1	48.3
wikipedia	2	18.1	2	18.1	23.0
amazon	9	-	0	7.7	32.9
twitter	5	42.8	1	14.0	62.3
blogspot	6	27.3	0	14.8	18.8
linkedin	0	28.8	0	21.7	39.4
msn	10	38.0	1	25.3	42.4
ebay	15	21.1	2	12.7	18.5
bing	0	16.5	0	16.5	27.4

Table 3.2: Taint analysis time

to observe the instrumentation overhead. Our instrumented Safari ran 42.7% slower than the original. The user should experience observable slowdown of the browser but still be able to run JavaScript websites for the testing scenario. Furthermore, *TracingSafari* is a dependent on a specific version of the browser because it modifies the JavaScript core engine. A browser-independent Execution Collector (e.g., *Jalangi* [63]) may be used to generalize our experiments to test websites on different browsers.

Overall, Static Taint⁺, analyzing code including JavaScript libraries (e.g., jQuery), could not complete analysis of 41% of the webpages we examined. Static Taint⁻, only analyzing application code, could not complete analysis on 4.6% of the webpages. Blended analysis, on the other hand, used dynamic information to focus the static analysis and ran to completion on all websites. Thus, given our timings for these analyses, we have support for our first hypothesis that blended taint analysis is scalable on real-world JavaScript websites.

Table 3.3 shows the results of the blended and the pure static taint analyses. Six of the 12 websites we experimented with contained reports from these analyses. We report the number of unique alarms for each website because duplicate alarms may originate from the same code in different webpages. Each alarm was checked manually to determine if it was a true positive (i.e., there actually exists at least one flow from a source to a sink) or not (i.e., false positive). For the sink-source pairs that flowed into an `eval` invocation reported for Static Taint⁺ or Static Taint⁻, we manually checked if there actually was a taint violation. In Table 3.3, columns 2, 4 and 6 present the number of true positives reported by Static Taint⁺, Static Taint⁻ and blended taint analysis, respectively. Columns 3, 5 and 7 show the number of false positives. Note that the results in Static Taint⁻ columns marked *

Website	Static Taint ⁺		Static Taint ⁻		Blended Taint	
	TP	FP	TP	FP	TP	FP
youtube	1	1	1	-	4	-
twitter	1	-	1*	-	3	-
linkedin	1	1	1*	1*	1	1
msn	-	-	-	-	2	-
ebay	2	-	-	-	3	-
bing	-	1	-	1*	-	-
totals	5	3	3	2	13	1

Table 3.3: Taint analysis results

mean the same alarms were reported by Static Taint⁺. Blended taint analysis reported 14 unique source-sink pairs from 5 websites; only one of them was a false positive. Static Taint⁺ reported 8 unique source-sink pairs from 5 websites; 3 of them were false positives. Static Taint⁻ reported 5 unique source-sink pairs from 4 websites; 2 of them were false positives.

Although Static Taint⁺ timed out on many webpages, it was able to discover 5 true positives, 3 of which were not discovered by Static Taint⁻. This suggests that it is crucial to model JavaScript libraries in the analysis in terms of detecting taint violations. Static Taint⁻ was able to locate only one different true positive from Static Taint⁺ on *youtube*, although it analyzed more webpages.

The blended taint analysis results in columns 6 and 7 in Table 3.3 support our second hypothesis that blended taint analysis is more accurate than pure static taint analysis:

- (i) Blended taint analysis discovered all the true positives that Static Taint⁺ or Static Taint⁻ reported.
- (ii) Blended taint analysis found 8 additional true positives that Static Taint⁺ did not report and 10 additional true positives that Static Taint⁻ did not report.
- (iii) For the 7 true positives detected by blended analysis, but not by either static analysis, 4 of them came from dynamic constructs the pure static analyses could not handle, while 3 of them were due to the scalability issue with Static Taint⁺.
- (iv) Blended analysis eliminated 2 false positives reported by Static Taint⁺ and 1 false positive reported by Static Taint⁻, leaving only one false positive reported in common with the two static analyses.

Threats to validity. There are several aspects of our experiments which might threaten the validity of our conclusions: (i) Because the traces of websites were manually collected by one user of the Execution Collector, the webpages explored may not be representative of all behaviors of web applications. The use of an automated tool to collect the traces may increase the ability to test for more general website usage. (ii) The accuracy of our implemented

framework is determined by limitations of the *WALA* interpretation of JavaScript. We found that there were parsing problems with some JavaScript websites, and some language constructs of JavaScript (e.g., the `with` construct) were ignored.

Chapter 4

State-sensitive Points-to Analysis

The unique object model of JavaScript including prototype-based inheritance and object property changes is one of the most important features that allow JavaScript applications to exhibit dynamic behavior. Software tools that analyze JavaScript programs need to take into account the dynamic behavior object model to be practical. In this chapter, we propose a novel flow- and context-sensitive points-to analysis that accurately handles dynamic changes in the behavior of JavaScript objects. We first present a code example that illustrates the imprecision of current points-to analysis. We have performed an empirical study on JavaScript objects that motivated and guided us to the design of the new analysis. We then present the state-sensitive points-to analysis for JavaScript and our experimental results.¹

4.1 Imprecision of Points-to Analysis

Three important dimensions of design choices for points-to analysis are flow sensitivity, context sensitivity and object representation. Some effective techniques for other programming languages may be insufficient due to the unique object model of JavaScript. A flow-insensitive analysis may produce imprecise results when obj-ref state changes, because it cannot perform strong updates. Context-sensitive analyses may produce imprecise results because they lack the power to distinguish between different obj-ref states for the same JavaScript object. In Figure 4.1, we present a JavaScript example to illustrate the sources of imprecision of a flow- and context-insensitive points-to analysis resulting from the dynamic behavior of JavaScript objects. We also demonstrate that object-sensitive analysis using the same object representation as Milanova *et al.* [47] is ineffective at distinguishing the function calls in the example.

Lines 2-6 show a polymorphic constructor function `X`, similar to the example in Figure 3.6.

¹Part of the contents presented in this chapter was published in [78] and [80].

```

1 function P(){ this.p = new Y1(); }
2 function X(b){
3     this.__proto__ = new P();
4     if(b) { this.p = new Y2(); }
5     else this.q = new Y3();
6 }
7 var x = new X(true);
8 x.bar = function(v, z){ v.f = z; }
9 var z1 = new Z();
10 x.bar(x.p, z1);
11 ...
12 x.p = new A();
13 ...
14 var z2 = new Z();
15 x.bar(x.p, z2);

```

Figure 4.1: JavaScript example of dynamic object behavior

Objects created by X may or may not have the local property named p or q (lines 4 and 5) depending on the value of its argument. The statement in line 12 updates the value of local property p of an object pointed to by x if p exists; otherwise, the statement adds the local property named p to the object. Figures 4.2 and 4.3 show the points-to graphs that reflect the run-time behavior of this code. We use the line number to represent the object created (e.g., the object created by X at line 7 is O_7). We focus on two program points in the execution, lines 10 and 15. The nodes O_7 , O_4 , and O_3 and O_9 constitute the obj-ref state of O_7 at line 10 and the nodes O_7 , O_{12} , O_3 and O_{14} constitute the obj-ref state of O_7 at line 15. Note that O_1 is not visible from O_7 at lines 10 or 15 because of the existence of the local property named p . The obj-ref state of object O_7 is different at these two program points.

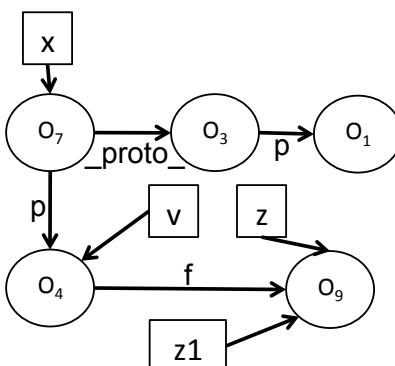


Figure 4.2: Run-time points-to graph at line 10

Constructor polymorphism (lines 2-6), object property change (line 12) and function invocations (lines 10 and 15) in this example make precise static points-to analysis hard to achieve

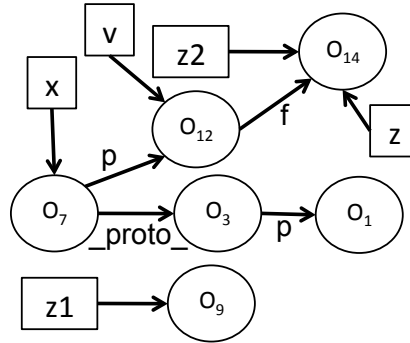


Figure 4.3: Run-time points-to graph at line 15

with current techniques. Figure 4.4 shows a static points-to graph for the example built by a flow- and context-insensitive points-to analysis. Dashed nodes and edges are imprecise points-to relations that cannot exist at runtime.

There are several sources of imprecision. Line 7 assigns variable x to an object created by the polymorphic constructor X . Not knowing the value of b , static analysis conservatively builds all the points-to relations possible from execution of X ; thus, results in the imprecise edge $\langle O_7, q \rangle, O_5$. When reading the property p of x (line 10), static analysis returns objects O_4 and O_1 because a conservative analysis cannot distinguish whether or not O_4 actually exists. However, O_1 is never accessible via $x.p$ because there always exists a local property named p of x in the example at runtime. Furthermore, because of the imprecise result of the read of $x.p$, invoking the `bar` function results in imprecise property reference from O_1 to O_9 . Flow-insensitive points-to analysis simply adds O_{12} to $O_7.p$ (line 12) because it cannot perform strong updates. Finally, because the context-insensitive analysis does not distinguish which objects v and z point to on different calls of `bar`, line 15 results in additional imprecision with respect to $O_4.f$ and $O_{12}.f$ (i.e., the points-to set of each local variable of `bar` is merged with the values at both call sites and then points-to algorithm calculates the solution based on the merged values).

We now discuss if existing techniques of flow and context sensitivity may avoid the imprecision from the flow- and context-insensitive analysis. First, a flow-sensitive analysis in general cannot strongly update indirect assignment statements because an abstract object may represent multiple instance variables. For example at line 12, if there is another variable x' that also refers to O_7 , strongly updating $O_7.p$ (i.e., adding O_{12} to $O_7.p$ and removing all other points-to relations of $O_7.p$) would result in the analysis not soundly approximating the program behavior because $x'.p$ should not be updated to point to O_{12} . All state-update statements that affect the object behavior are indirect assignment statements; therefore, flow sensitivity in general cannot improve the the analysis precision due to the dynamic behavior of JavaScript objects. Second, assuming there is a sound technique to strongly update the state-update statements, there still is need for an appropriate context-sensitive analysis to remove the imprecise edges $\langle O_4, f \rangle, O_{14}$ and $\langle O_{12}, f \rangle, O_9$, distinguishing calls to `bar`

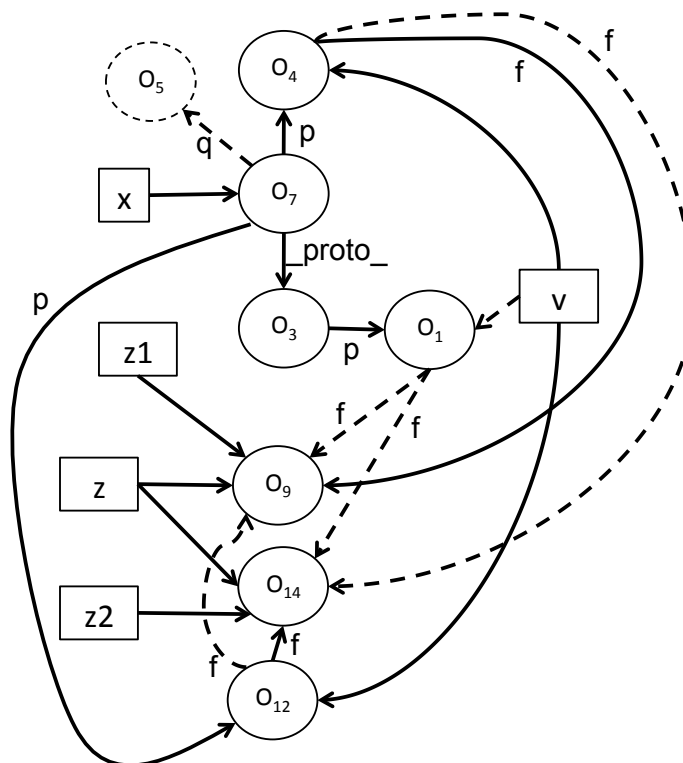


Figure 4.4: Flow- and context-insensitive points-to graph

(lines 10 and 15) by their calling contexts. Object-sensitive analysis, the popular choice of context sensitivity for object-oriented languages, is not able to differentiate these two call sites because they have the same receiver object O_7 , which has two different obj-ref state at these call sites. Our new points-to analysis is designed to handle these constructs more accurately and to address the challenges raised by obj-ref state updating and prototype-based inheritance.

4.2 Empirical Study of JavaScript Object Behavior

We have discussed in Section 4.1 that several design features of JavaScript objects render static points-to analysis imprecise. Nevertheless, because our goal is to design a dataflow analysis that is practical for analyzing real-world JavaScript applications, it is crucial to learn if and how these features of the JavaScript dynamic object model have been applied in JavaScript websites to guide our choices in the new points-to algorithm. Therefore, we have conducted an in-depth empirical study focusing on understanding the run-time behavior of JavaScript objects. We designed specific metrics for measuring JavaScript object behavior and summarized the behavioral patterns suggesting common practices.

4.2.1 Experimental Design

Benchmarks and tools. We chose to conduct the experiments on JavaScript websites to reflect the behavior of real JavaScript applications. In our study, we used the benchmarks collected by Richards *et al.* [59]. The benchmarks, collected by *TracingSafari*, consist of 114 dynamic traces (i.e., origin-traces²) extracted from 70 popular websites. Our study was implemented as an augmented version of an offline analysis tool, *TraceAnalyzer* [59].

Object categories. Different kinds of objects are allocated during the execution of JavaScript websites. We categorize these objects into the following kinds: (i) basic datatypes (i.e., the built-in objects including *Date*, *Array*, *String*, etc.), (ii) anonymous objects (i.e., the objects created via a pair of braces `{...}`), (iii) DOM objects (i.e., the HTML document objects), (iv) functions (i.e., the objects created by the `Function` constructor), (v) native objects (i.e., the objects created through execution of native code), and (vi) user objects (i.e., the objects created via the `new` constructor expression).

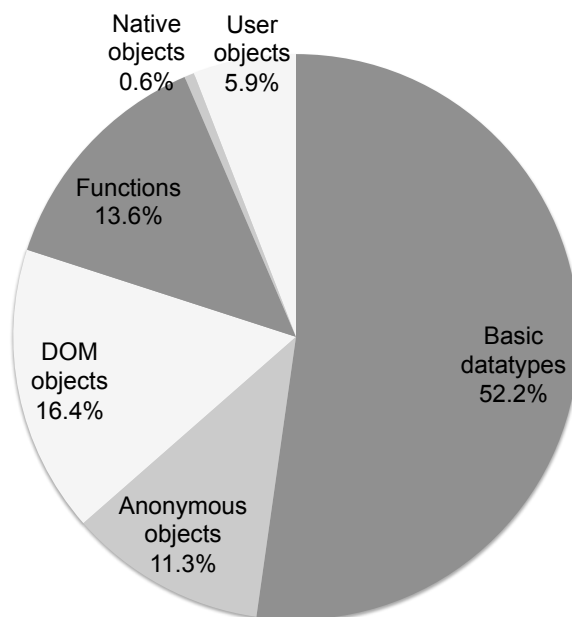


Figure 4.5: Percentage of object instances in each object category

Figure 4.5 shows the distribution of the object instances in these categories over all origin-traces. More than 50% of the instances are basic datatypes among which arrays are the most frequently created. The number of user objects are relatively small (5.9%). Figure 4.6 shows

²To distinguish from the notion of trace we discussed in Chapter 3, we use the term origin-trace to represent the traces we use for this empirical study, which were collected by Richards *et al.* using the original *TracingSafari*.

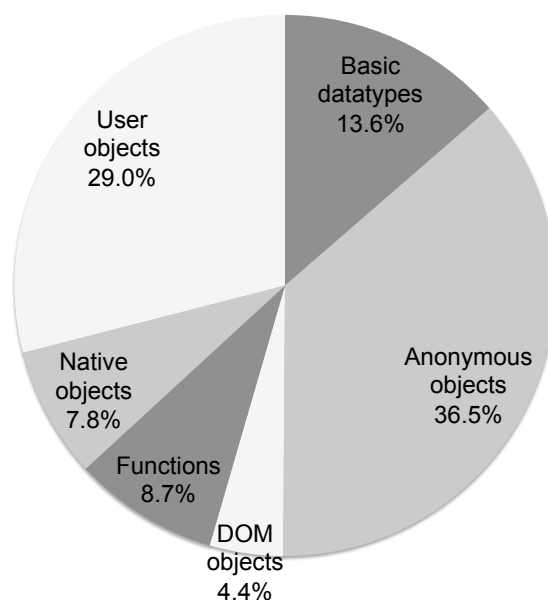


Figure 4.6: Percentage of operations associated with each object category

the distribution of the number of operations for each object category over all origin-traces. Most operations occurred on anonymous objects (36.5%) and user object (29.0%), while only 4.4% of the operations occurred on DOM objects. Comparing the results in Figure 4.5 and Figure 4.6, we observe that user objects are among the most active categories (i.e., on average 26 operations were associated with each user object). Because the precision of points-to analysis is affected by the choices on analyzing user objects and this object category exhibited active behavior, we focus on user objects in this study.

Experimental setup. An origin-trace is a compressed file containing source code and a sequence of statements that are recorded at runtime [59]. We focused on the following statements that are related to JavaScript object behavior: (i) property writes, (ii) property reads, (iii) property deletes and (iv) constructor returns. In the experiments, we analyzed the statements in sequence and assigned a unique operation kind for each statement. Table 4.1 shows the relation of statements to operation kinds. Note that the same statement may result in different operation kinds under different circumstances. In JavaScript, a property write statement can only change a local property; thus, a property write may result in one of the three operation kinds (i.e., add, override, and update) for better understanding of its effect on object properties and inheritance. The property lookup mechanism in JavaScript, on the other hand, may use the prototype chain to read an inherited property; hence, we assign one of the two operation kinds (i.e., read-inherit and read-local) to a property read statement. We use the constructed operation for a user object to distinguish its construction stage from the rest of its object lifetime.

Statements	Operation kind	Preconditions
property write	add	The property does not exist locally or on the prototype chain.
	override	The property does not exist locally but exists on the prototype chain.
	update	The property exists locally.
property read	read-inherit	The property does not exist locally.
	read-local	The property exists locally.
property delete	delete	
constructor return	constructed	

Table 4.1: The relationship between statements and operation kinds

We modified *TraceAnalyzer* to produce the operation kinds in Table 4.1. Our implementation produced both aggregated results and detailed information for individual objects. Figure 4.7 shows an example of the history information (i.e., sequence of operations) associated with an object. For each operation, we output the operation kind, property name, the ID and category of the property as well as other information (e.g., the property access chain from a read-inherit operation). The empirical study was conducted in a 2.66GHz Intel Core 2 Duo MacBook Pro with 4GB memory running the Mac OS X 10.6.8 operating system.

```

=====Object Information=====
609. Object ID: 15796
    Category: user object
    Operations: 24
    Prototype: 8054

-----History-----
1. OpKind: add      Property: fn          PropID: 9697(function)
2. OpKind: add      Property: overrideContext
   PropID: 15104(constructed by 9807(function))
3. OpKind: constructed
.....
21. OpKind: read-inherit  Property: contains  PropID: 8057(function)
    Chain: 15796-8054
22. OpKind: read-local   Property: fn        PropID: 9697(function)
23. OpKind: delete      Property: fn
24. OpKind: delete      Property: obj

```

Figure 4.7: Sample history information of an individual object from *yahoo*

4.2.2 Metrics

Operation kind distribution. The behavior of a user object is defined by its associated operations. An object is more dynamic when its properties change (e.g., override, add or delete) frequently. The percentage of read-inherit operations suggest the importance of precisely knowing the prototype mechanism. Figure 4.8 presents the distribution of read (i.e., read-local and read-inherit) vs. write and delete (i.e., add, update, override and delete) operations. For user objects, read operations comprised 81% of all operations, indicating a relatively small fraction of operations may possibly change object properties.

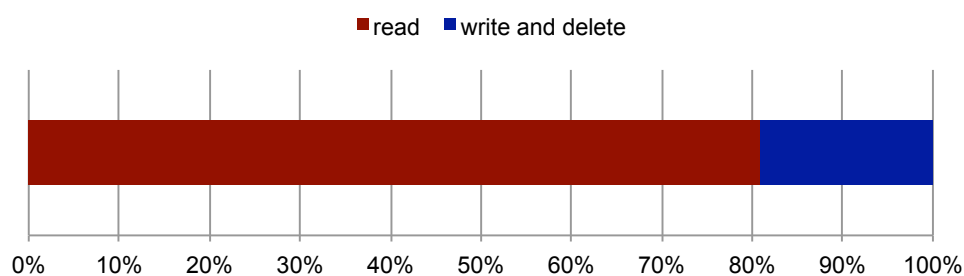


Figure 4.8: Read vs. write and delete operation distribution

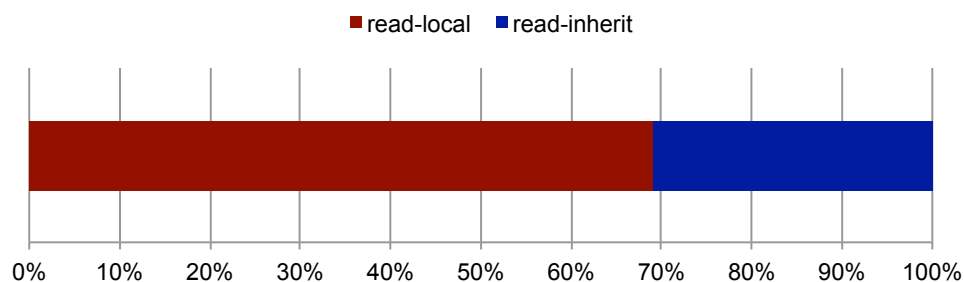


Figure 4.9: Read-local and read-inherit operation distribution

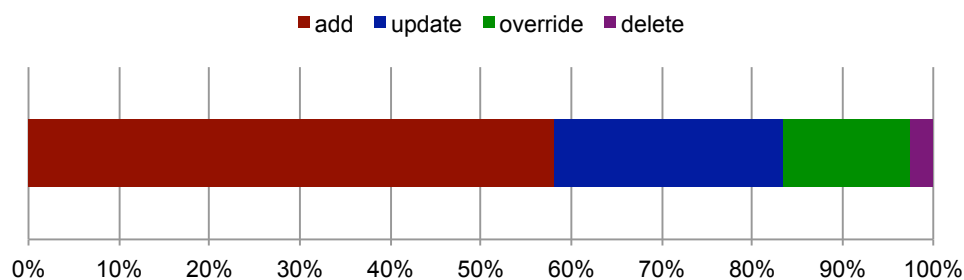


Figure 4.10: Write and delete operation distribution

Figures 4.9 and 4.10 present additional details on the information in Figure 4.8. Figure 4.9 shows that more than 30% read operations of user objects were read-inherit operations, suggesting user objects actively use their prototype chains to lookup properties. Figure 4.10 illustrates the distribution among write and delete operations. Delete operations were infrequently observed. About 3% of all write and delete operations of user objects were delete operations, which means properties of user objects are sometimes removed at some point during execution. Override and update operations occurred more often; specifically, 14% and 25% of write and delete operations of user objects were override and update operations, respectively. Add operations were most frequently observed, comprising 58% of write and delete operations of user objects.

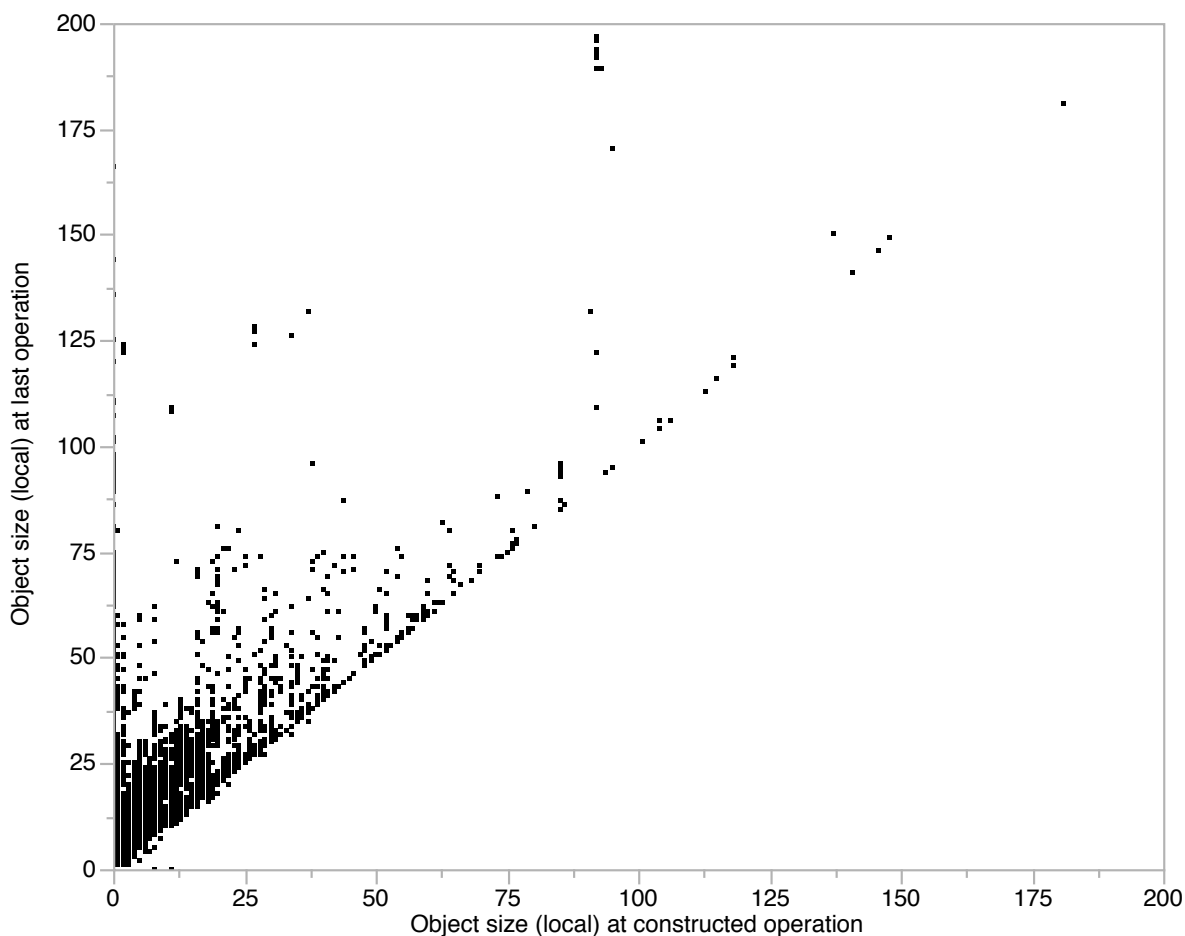


Figure 4.11: Local sizes of user objects at their constructed and last operations

Object size. We define the size of a JavaScript object as its number of accessible properties (including local and inherited properties) at a program point during execution. Since the property list of a JavaScript object is not fixed at runtime, object size may change.

We calculate JavaScript user object sizes at two crucial stages in object lifetime: at its constructed operation and its last observed operation (i.e., an approximation of the end of object lifetime). On average over all the user objects, the object size was 28 at the constructed operation. Figure 4.11 shows the local sizes of user object (i.e., counting only local properties) at their constructed and last operations. There were user objects whose local sizes were the same at both operations in their lifetime (i.e., the points on the $x = y$ line). However, we observed that the local sizes of many user objects grew significantly by the end of their lifetime compared to local sizes at their constructed operations. This result gives evidence that the local size of a JavaScript user object is usually not consistent at different stages of its lifetime and in most cases increases.

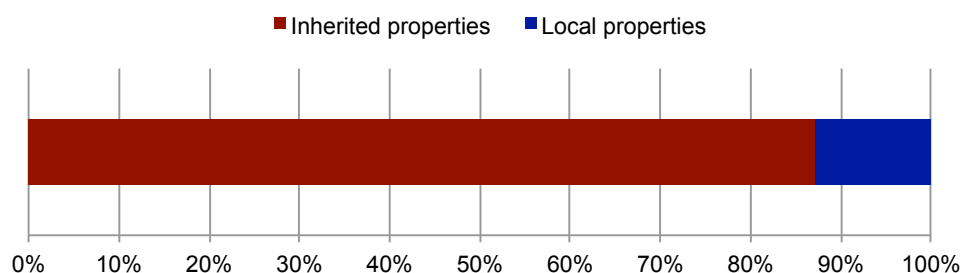


Figure 4.12: Inherited vs. local properties in user objects

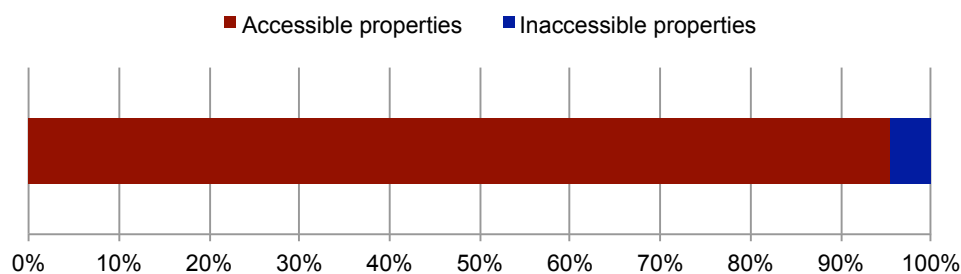


Figure 4.13: Accessible vs. inaccessible properties from prototype objects

Property inheritance. Inherited properties serve as the goal for code reuse, while overridden properties allow more specific behavior of objects. Figure 4.12 shows the percentage of inherited and local properties of all the accessible properties in user objects at their construction stage. For each user object at its constructed operation, we collected the local property list as well as the property lists of its prototype objects. All properties in the local property list were counted as local properties and a subset of the properties in the property list of its prototype objects were counted as inherited properties conforming to the JavaScript property lookup mechanism. We found that 13% of the properties were implemented for specific user objects, while most of the properties (87%) were inherited from prototype objects. Over all the properties in the prototypes of user objects, Figure 4.13 shows the percentage of

properties accessible from user objects. For the prototype objects of each user object at its constructed operation, properties that were overridden were counted as inaccessible properties; others were accessible properties. In Figure 4.13, 5% of the properties were overridden so that they were inaccessible and the rest (95%) were accessible.

4.2.3 Object Behavioral Patterns

In addition to the above overall metrics on JavaScript user objects, we present a study on the object behavioral patterns across the benchmarks and discuss some representative cases. An operation occurrence pattern illustrates a representative sequence of operations occurring on a specific object or property. A property change pattern presents frequently occurring changes from one object category to another on specific properties. We studied those object behavioral patterns that may affect the design choices of dataflow analysis from more than 1 million user objects and more than 9 million of their properties in the origin-traces.

Operation occurrence patterns. The recurring sequence of operations on objects or properties may suggest a common coding style and/or user interactions of JavaScript applications. Knowing the operation occurrence patterns, we can better design a dataflow analysis that accommodates the frequently observed patterns of object behavior. Table 4.2 shows the operation occurrence patterns we studied. We designed these patterns based on: (i) operation sequences frequently observed in the experiments and (ii) the usage of delegation in JavaScript. Patterns 1-5 reflect the sequences of operations on a specific property p and patterns 6-9 show the relationship between a constructed operation and other operations on a user object. We use two quantifiers to express the number of times an operation occurs (i.e., $+$ for operations occurring 1 or more times and $\{n\}$ for operations occurring n times exactly). We discuss each pattern and our empirical observations in detail below.

	Operation occurrence pattern	Notes
1	<code>(add p override p update p)+ → delete p</code>	regular and abnormal delete practices
2	<code>(add p override p update p){0} → delete p</code>	
3	<code>read-local p+ → delete p → read-inherit p+</code>	local&inherited property lookups for same p
4	<code>read-inherit p+ → override p → read-local p+</code>	
5	<code>read-local p+ → update p</code>	“temporary” property
6	<code>delete+ → constructed</code>	delete(s) before/after constructed operations
7	<code>constructed → delete+</code>	
8	<code>update+ → constructed</code>	interesting writes before/after constructed
9	<code>constructed → (add override)+</code>	

Table 4.2: Operation occurrence patterns of user objects

Because delete is not supported by most popular programming languages, its semantics have not been widely studied; therefore, there are few specific program analysis techniques that

handle deletes. In addition to property deletions that remove local properties, property write operations (i.e., add, override and update) may also add/change the values of local properties. We first study the relationship between write operations and delete operations.

- Pattern 1: $(\text{add } p \mid \text{override } p \mid \text{update } p)^+ \rightarrow \text{delete } p$. This pattern occurred on 106,137 properties of the user objects in 56 out of 114 origin-traces in the benchmarks. The average percentage of properties of user objects exhibiting Pattern 1 in the 56 origin-traces was 0.9% with standard deviation of 1.8% across the origin-traces. With this relatively high standard deviation, we observed the origin-traces that contributed most to this pattern were from *google*, especially *gmail* (i.e., 9% of properties of user objects in *gmail*, 62,055 in total). The scenarios of Pattern 1 can be interpreted as follows: (i) if the pattern occurs only a few times, the specific property is accessible only between the property write operation and the delete; (ii) if the pattern occurs on the same property many times, the property is most likely used as a temporary variable. The majority (99.8%) of properties that exhibit Pattern 1 only contain one iteration of the pattern. This suggests that most uses of delete are to end the lifetime of a specific property; after the local property is deleted, an inherited property (if it exists) will be accessible. Nevertheless, there are some properties exhibiting multiple occurrences of Pattern 1. For example, Pattern 1 occurred 149 times on one property of a user object in *mozilla*; this property `lastAction` is used to check if a function may be called and then it is deleted. In this scenario, the developer creates the property when needed and uses a delete statement to ensure that specific property only exists in a certain part of the program. This usage is considered as a legitimate use of delete.
- Pattern 2: $(\text{add } p \mid \text{override } p \mid \text{update } p)\{0\} \rightarrow \text{delete } p$. This pattern occurred on 70,143 properties of the user objects in 37 out of 114 origin-traces in the benchmarks. The average percentage of properties of user objects exhibiting Pattern 2 in the 37 origin-traces was 0.9% with standard deviation of 1.6%. The distribution of Pattern 2 is similar to Pattern 1 in that *google* dominates the uses of the delete operation with 7% of the properties of user objects (i.e., 45,440 in total) in *gmail* exhibiting Pattern 2. This pattern occurs more frequently than expected because Pattern 2 describes that a property `p` is deleted when it never had been added, overridden or updated. In JavaScript semantics, a delete operation on a non-existing local property does not alter the object. The developers of JavaScript websites are likely using the delete statement to ensure that a local property does not exist at some program point. Although the occurrence of Pattern 2 during execution will not produce a run-time error, it reflects the difficulty of controlling properties of a JavaScript object.

After a local property `p` is deleted from the object `o`, reading `o.p` uses the prototype chain of `o`. The following pattern shows that at different points of the execution, a delete operation may result in accessing local vs. inherited properties.

- Pattern 3: `read-local p+ → delete p → read-inherit p+`. This pattern occurred on 311 (0.03%) properties of the user objects in the benchmarks. Although this pattern does not occur as frequently as the others, it is the most straightforward pattern showing the influence of delete operation on the uses of an object property. The occurrences of Pattern 3 are limited to fewer than 10 websites (e.g., *npr.org*) and the deleted properties are all function properties. This implies specialization of the function properties; a read operation on the object results in use of a different property lookup mechanism at different program points (i.e., local property vs. prototype chain lookup).

The override operations also affect the local property list of an object such that reading a property of an object may result in read-inherit vs. read-local operations before and after the override operations, respectively. In addition, because an update operation changes the value of a local property, the read-local operations at different program points may return different results if an update operation occurs between them.

- Pattern 4: `read-inherit p+ → override p → read-local p+`. This pattern occurred on 281,160 properties of the user objects in 73 out of 114 origin-traces in the benchmarks. The average percentage of properties of user objects exhibiting Pattern 4 in the 73 origin-traces was 1.6% with standard deviation of 3.2%. The two websites that experienced significant number of Pattern 4 were *me.com* and *npr.org*. This pattern directly shows the impact of an override operation on the uses of the property and it occurs more frequently than Pattern 3. Pattern 4, similar to Pattern 3, indicates that understanding JavaScript property accesses can be difficult, requiring more accurate program analysis techniques to address this issue.
- Pattern 5: `read-local p+ → update p`. This pattern occurred on 616,825 properties of the user objects in 106 out of 114 origin-traces in the benchmarks. The average percentage of properties of user objects exhibiting Pattern 5 in the 106 origin-traces was 7% with standard deviation of 5.5%. As the most frequently observed pattern on a specific property in the benchmarks, Pattern 5 exists in almost all the websites. *go.com*, *facebook* and *yahoo* are the three websites with the highest percentage of properties of user objects experiencing this pattern (i.e., 41%, 23% and 21%, respectively). We observed there were 127,052 properties in the benchmarks that were read and updated more than once. If update happens frequently on a property, then this property may be regarded as a temporary variable. Frequently updating a property value is a common object-oriented practice for building data structures (e.g., list) and control structures (e.g., loop) in the program, while updating the object category of a property is unusual; thus, we have performed an in-depth study on frequently occurred patterns of property changes from one object category to another.

We divide the lifetime of a user object into two stages: before and after construction. Different object behavior is expected at these two stages: (i) before a user object is constructed,

properties should be frequently added/overridden and (ii) after a user object is constructed, its properties often may be used and updated. We first investigated the stage of a user object at which delete operations happen.

- **Pattern 6: `delete+` \rightarrow `constructed`.** This pattern illustrates that property deletion occurs in the construction stage of an object (i.e., the deleted property may not be accessed after the object is constructed). It occurred on 34,725 user objects in 23 out of 114 origin-traces in the benchmarks. The average percentage of user objects exhibiting Pattern 6 in the 23 origin-traces was 9% with standard deviation of 12%. Origin-traces from *google* (including *gmail* and *google docs*) all contained objects that experienced many occurrences of this pattern, from 18% to 51% of user objects in each origin-trace. Other websites that frequently exhibit Pattern 6 (more than 5% of user objects) are *virtualecrets.com* and *npr.org*. There is a strong correlation between Pattern 6 and Pattern 2. Most of the deleted properties within the construction stage do not exist locally at the time of deletion. For example, an object from *myspace* exhibits 7 deletions of the same property `q` in its constructor, although the property never exists locally. After inspecting the code, we found that in a constructor where several function properties of the object are defined, there is a delete statement (i.e., `delete this.q`) that will always execute without checking the existence of the property `q`.
- **Pattern 7: `constructed` \rightarrow `delete+`.** The occurrence of the delete operation after the construction stage is considered normal if the property is used before the deletion (e.g., Pattern 1). Pattern 7 occurred on 42,372 user objects in 56 out of 114 origin-traces in the benchmarks. The average percentage of user objects exhibiting Pattern 7 in the 23 origin-traces was 4.3% with standard deviation of 9%. Pattern 7 was observed in a larger set of websites than Pattern 6. In addition to *google*, many user objects from *facebook* also exhibited Pattern 7. We observed that deletions happen frequently in some objects; more than 500 objects are associated with at least 50 delete operations after the construction stage. Potentially, these objects exhibit very different behaviors in between these delete operations.

Property addition and overriding are common for an object within the construction stage. We conducted further study on the update operations in the construction stage and on the add/override operations that happen after an object is constructed.

- **Pattern 8: `update+` \rightarrow `constructed`.** This pattern occurred on 60,742 user objects in 90 out of 114 origin-traces in the benchmarks, much more widely observed than the delete operation related patterns (i.e., Patterns 6 and 7). The average percentage of user objects exhibiting Pattern 8 in the 90 origin-traces was 5.5% with standard deviation of 6.7%. Websites that most frequently experienced Pattern 8 were *yahoo*, *go.com* and *me.com*. Updating properties in the construction stage suggests that some JavaScript object constructor functions do not just create properties.

- Pattern 9: `constructed` \rightarrow `(add | override)+`. This pattern occurred on 265,224 (20.56%) user objects in 106 out of 114 origin-traces in the benchmarks. The average percentage of user objects exhibiting Pattern 9 in the 106 origin-traces was 28% with standard deviation of 23%. For some websites (i.e., *easychair.org*, *raphaeljs.com* and *me.com*), most user objects (more than 90%) experienced Pattern 9. Thus, the local property lists of many user objects are expanded by adding a new property or overriding an inherited property. This result conforms to our observations in Figure 4.11 and indicates that the behavior of a JavaScript object may not be represented by its properties at the point of its construction.

Property change patterns. An object property that changes from one object category to another may indicate significant change of behavior. Table 4.3 shows property change patterns for user objects. The `function` \rightarrow `user object`, `anonymous` \rightarrow `basic datatype` and `basic datatype` \rightarrow `user object` patterns all occurred frequently (more than 1000 times) in the benchmarks. These property changes suggest many properties are used for unrelated purposes at different program points, another coding practice that poses challenges for program analysis to reason about the object behavior. As for individual websites, *google*, *facebook* and *flapjax-lang.org* are among the websites we observed the most property change patterns. `function` \rightarrow `user` patterns occurred most frequently on *ebay*, while `anonymous` \rightarrow `basic` patterns were from *flapjax-lang.org*. For the less frequently observed patterns, *flapjax-lang.org*, *me.com* and *google.com* dominated the occurrences of `basic` \rightarrow `anonymous`, `basic` \rightarrow `function` and `user` \rightarrow `function` patterns, respectively.

Pattern	Occurrences	Pattern	Occurrences
<code>function</code> \rightarrow <code>user</code>	6384	<code>basic</code> \rightarrow <code>anonymous</code>	260
<code>anonymous</code> \rightarrow <code>basic</code>	3138	<code>basic</code> \rightarrow <code>function</code>	144
<code>basic</code> \rightarrow <code>user</code>	1320	<code>user</code> \rightarrow <code>function</code>	46

Table 4.3: Property change patterns of user objects

4.2.4 Summary

We have performed an empirical study on websites to understand the run-time behavior of JavaScript objects. We evaluated several dynamic metrics on user objects and investigated frequently occurring behavioral patterns. Based on our observations, the dynamic designs of JavaScript objects are widely used in real-world applications and thus, it is necessary for program analysis algorithms to include an accurate model to reason about JavaScript objects.

4.3 State-sensitive Points-to Analysis

Based on the motivating example and empirical study, we have designed a new points-to analysis for JavaScript. In this section, we will present our state-sensitive points-to analysis. We will explain the key ideas used in the analysis, including the intra-procedural program representation (i.e., the block-sensitive decomposition of control flow graphs), the solution space (i.e., the annotated points-to graph with access path edges and in-construction nodes), the transfer functions of state-update statements as well as the state-preserving statements, state sensitivity (i.e., a form of context sensitivity based on object sensitivity that captures changes in object behavior during execution) and block sensitivity (i.e., a partial flow sensitivity performed on the transformed CFG). Finally, we will discuss the implementation facts of our algorithm.

4.3.1 State-Preserving Block Graph

A flow-insensitive analysis ignores the control flow of a program while a flow-sensitive analysis typically uses an intra-procedural control flow graph. Our analysis aims to provide a better model of a JavaScript object whose reference state exhibits a flow-sensitive nature (e.g., allowing addition and deletion of object properties at any program point). Cognizant of the possible overhead introduced by a fully flow-sensitive analysis, we designed a partially flow-sensitive analysis that only performs strong updates when possible on state-update statements using a transformed CFG, called the *State-Preserving Block Graph (SPBG)*. Recall that the state-update statements, including the property write (i.e., add or update a property) and delete (i.e., remove a property), directly change the obj-ref state in JavaScript; all other statements (e.g., property read) are state-preserving statements.

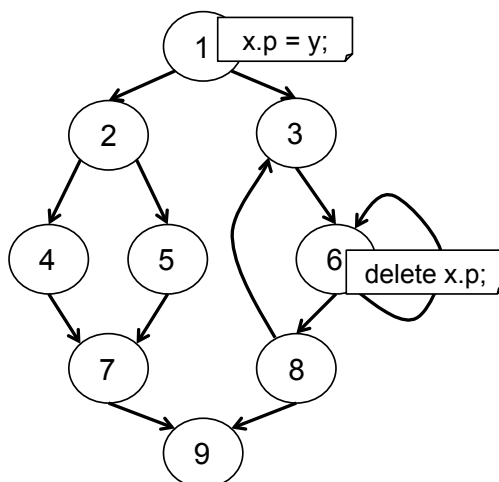


Figure 4.14: CFG

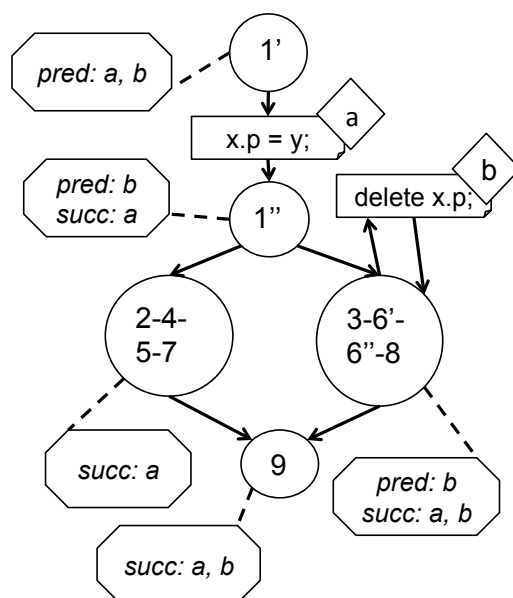


Figure 4.15: SPBG

Figure 4.15 shows an example SPBG compared to its original CFG (Figure 4.14). An SPBG is a transformed control flow graph whose basic blocks are aggregated into region nodes according to whether or not they contain a state-update statement. The SPBG also contains state-update statements as special singleton statement nodes (i.e., state-update nodes). An example of a region node (i.e., state-preserving node) is 2-4-5-7 in Figure 4.15 whereas node $x.p = y$ is an example of a state-update node. Note that by creating singleton nodes, the algorithm breaks apart former basic blocks (e.g., $1 \rightarrow \{1', x.p = y, 1''\}$).

We first split any basic blocks in the CFG that contain at least one state-update statement, obtaining a split-CFG. State-update statements that are property write and delete statements can be detected syntactically. In addition, because our analysis requires a call graph as input (Section 4.3.6), invocations that may result in an obj-ref state change are found by a call graph traversal. We then use a variant of the standard CFG construction algorithm [2] to build the split-CFG. The header nodes used include the standard headers (Section 2.1.2) plus (i) any state-update statement is a region header of a state-update node containing only that statement, and (ii) any state-preserving statement that immediately follows a state-update statement is a region header of a state-preserving node.

In an SPBG, state-preserving region nodes are formed based on grouping nodes in the split-CFG that share the same control flow relations with respect to state-update nodes. The possible control flow relations of node n_1 and n_2 in a split-CFG include: (1) n_1 is a successor of n_2 , (2) n_1 is a predecessor of n_2 , (3) n_1 is both a successor and a predecessor of n_2 (i.e., n_1 and n_2 exist in a loop) and (4) n_1 and n_2 have no control flow relation (e.g., n_1 and n_2 are present in different branches). We label each node in a split-CFG with its relations to each state-update node via depth-first searches. The set of labels form a signature for that

node. If nodes share the same signature it means that they have the same control flow relationship(s) to a (set of) state-update statement(s) so that they can be collapsed to a state-preserving node in the SPBG. Figure 4.15 shows the signatures of the state-preserving regions in the generated SPBG; **a** and **b** represent the state-update statements `x.p = y` and `delete x.p`, respectively. Basic blocks 2, 4, 5 and 7 are aggregated because they only appear as successors of `x.p = y` and have no control flow relation to `delete x.p`. The region node 2-4-5-7 is not further aggregated with basic block 9 because 9 is a successor of `delete x.p` but 2-4-5-7 is not.

4.3.2 Points-to Graph Representation

Our points-to graph representation includes novel components that facilitate the handling of strong updates for state-update statements. Recall that most flow-sensitive analysis algorithms cannot perform strong updates for indirect assignment statements (e.g., `x.p = y`) and few analyses consider property delete statements, which are uncommon in object-oriented languages. Two existing techniques help to enable strong updates for such statements in JavaScript: recency abstraction and access path maps.

Recency abstraction [6, 24] associates two memory-regions with each allocation site. The most-recently-allocated block, a concrete memory-region, allows strong updates and the not-most-recently-allocated block is a summary memory-region. We adapt the idea of recency abstraction to enable strong updates during analysis of constructor functions.

De *et al.* [11] performed strong updates at indirect assignments by computing the map from access paths (i.e., a variable followed a sequence of property accesses) to sets of abstract objects. This work demonstrated the validity of using access path maps to perform strong updates for indirect write statements in Java. We adapt this approach to points-to analysis for JavaScript by expanding the points-to graph representation instead of using separate maps.

Table 4.4 lists the nodes, edges and annotations in our points-to graph. In addition to the variable nodes `v` and abstract object nodes `o`, our points-to graph contains *in-construction object nodes* `@o`. Similar to the recency abstraction, an in-construction object always describes exactly one concrete object, which exists only during analysis of a constructor.

There are three kinds of edges. Variable reference and property reference edges exist in a traditional points-to graph. An *access path edge*, $\langle v, p \rangle, \phi o$, denotes that the property `p` of variable `v` refers to object `ϕo`. $\langle v, p \rangle$ represents an access path with length of 2 (i.e., a variable followed by one field access `v.p`).³

Our analysis calculates may pointer information, meaning that a points-to edge in the graph may or may not exist at runtime. To better approximate the obj-ref state of JavaScript

³The length of an access path is one more than the number of field accesses [11].




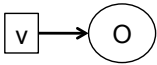
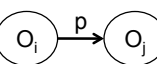
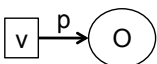
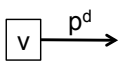
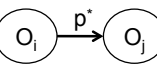
points-to graph G	node N	variable v	
		abstract object o	
		in-construction object $@o$	
	edge E	variable reference $(v, \phi o)$	
		property reference $(\langle \phi o_i, p \rangle, \phi o_j)$	
		access path $(\langle v, p \rangle, \phi o)$	
	annotation A	d annotation p^d	
		* annotation p^*	

Table 4.4: Expanded points-to graph with annotations

objects, we introduce annotations on property reference edges as well as access path edges. The annotations help to calculate must exist information for object property names. In our analysis, the d annotation on a property name p (i.e., p^d) denotes that the local property named p must not exist. This annotation only applies to access path edges in our points-to graph. The other annotation, $*$, applies to both property reference edges and access path edges. p^* denotes that the local property named p may not exist. Thus, property reference edges without annotation or access path edges without annotation represent must exist information for the property names in the expanded points-to graph. We use p^α to represent any kind of p^d , p^* or p edge. These annotated edges help us perform a more accurate property lookup.

$Pt(x)$ denotes the points-to set of x and $Pt(\langle \phi o, p \rangle)$ denotes the points-to set of the property p of ϕo . $Pt(\langle v, p \rangle)$ denotes the points-to set of access path $v.p$. We also define the operation $Alias(v)$ which returns the set of variables W such that v and $w \in W$ point to the same object. $apset(v)$ denotes the set of all access path edges of v (i.e., $apset(v) = \forall q : \{(\langle v, q^\alpha \rangle, \phi o)\}$).

In addition to the points-to graph, we use a mapping data structure to store intermediate information in the analysis. The map M is used to record the list of property names when an object is constructed. An abstract object (e.g., o) is the key in M whose value is the set of

local property names that exist when the constructor function of the abstract object returns (e.g., $\{p1, p2, p3\}$).

4.3.3 Transfer Functions

Now we describe the dataflow transfer functions for the statements shown in Tables 4.5 and 4.6.

Object creation $s_i : x = \text{newX}(a_1, a_2, \dots, a_n)$. In our analysis, an object creation statement (i.e., **new** statement) is modeled in three steps. $x = \text{new } X$ creates an in-construction object $@o_i$. Then the invocation of the constructor $\text{newX}(a_1, a_2, \dots, a_n)$ is modeled as a function call on $@o_i$. Upon the return of the constructor (i.e., ret_x), the analysis removes the in-construction object from the points-to graph and redirects all points-to relations from $@o_i$ to an abstract object (i.e., $\text{remove}(\mathbf{G}, @o_i)$). If the local property set of the in-construction object matches that of an existing abstract object with the same allocation site, the in-construction object is merged into the abstract object; otherwise, a new abstract object is created to replace the in-construction object.

In our analysis, there is at most one in-construction object for each creation site to allow strong updates. Thus, specific handling is required for recursive constructor calls. If an in-construction object is to be created when there already is an existing in-construction object for the same allocation site, our analysis resolves the existing in-construction object into a special abstract object whose set of properties upon construction is *unknown*. During the fixed point iteration, this special abstract object aggregates property reference edges due to the recursive constructor.⁴

The transfer function of the object creation statement ensures that abstract objects are based on their allocation site as well as their constructed local properties (i.e., an approximation of actual obj-ref state); in other words, the objects created at the same allocation site that contain the same set of local property names share the same abstract object in our analysis. This object representation is more precise than using one abstract object per creation site.

Direct write $x = y$. The effects of direct variable assignment on the points-to graph are relatively straightforward. $x = y$ creates points-to edges from x to all objects pointed to by y . Note that we perform weak updates on direct assignments. Although the analysis removes all the access paths edges of x from the points-to graph (i.e., $\mathbf{G} - \text{apset}(x)$), soundness is ensured because lookups through the abstract objects reflect less precise, yet over approximations (see Figure 4.18). Also, the access path edges of y cannot be copied to x because access path edges can only be added via strong updates.

Property write $x.p = y$. In general, strong updates cannot be performed on the prop-

⁴Our analysis uses a finite set of abstract objects and calling contexts, ensuring that a fixed point solution will be reached, in this case.

<p>Object creation: $\mathbf{s}_i : \mathbf{x} = \text{newX}(\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n)$</p> <p>Transfer function: (1) $\mathbf{x} = \text{new X} : (\mathbf{G} - \text{apset}(\mathbf{x})) \cup (\mathbf{x}, @o_i)$ (2) $\text{new X}((\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n)) : \mathbf{G} \cup \{\text{invoke}(\mathbf{G}, \mathbf{X}, @o_i, \mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n)\}$ (3) $\text{ret}_x : \text{remove}(\mathbf{G}, @o_i)$</p>
<p>Direct write: $\mathbf{x} = \mathbf{y}$</p> <p>Transfer function: $(\mathbf{G} - \text{apset}(\mathbf{x})) \cup \{(\mathbf{x}, \phi o_i) \mid \phi o_i \in \text{Pt}(\mathbf{y})\}$</p>
<p>Property write: $\mathbf{x.p} = \mathbf{y}$</p> <p>Transfer function: (1) if $\text{Pt}(\mathbf{x}) = 1$ and $\{\phi o_i \in @0 \mid \phi o_i \in \text{Pt}(\mathbf{x})\}$: $(\mathbf{G} - \{(\langle \langle @o_i, \mathbf{p}^\alpha \rangle, \phi o_j) \mid @o_i \in \text{Pt}(\mathbf{x}) \wedge \phi o_j \in \text{Pt}(\langle \langle @o_i, \mathbf{p}^\alpha \rangle) \}) \cup \{(\langle \langle @o_i, \mathbf{p} \rangle, \phi o_j) \mid @o_i \in \text{Pt}(\mathbf{x}) \wedge \phi o_j \in \text{Pt}(\mathbf{y})\})\}$ (2) otherwise : (2.1) $(\mathbf{G} - \{(\langle \langle \mathbf{x}, \mathbf{p}^\alpha \rangle, \phi o_i) \mid \phi o_i \in \text{Pt}(\langle \langle \mathbf{x}, \mathbf{p}^\alpha \rangle) \}) \cup \{(\langle \langle \mathbf{x}, \mathbf{p} \rangle, \phi o_j) \mid \phi o_j \in \text{Pt}(\mathbf{y})\})$ (2.2) $\mathbf{G} \cup \{(\langle \langle \phi o_i, \mathbf{p}^* \rangle, \phi o_j) \mid \phi o_i \in \text{Pt}(\mathbf{x}) \wedge \phi o_j \in \text{Pt}(\mathbf{y})\}$ (2.3) $\mathbf{G} \cup \{(\langle \langle \mathbf{z}, \mathbf{p}^* \rangle, \phi o_i) \mid \mathbf{z} \in \text{Alias}(\mathbf{x}) \wedge \text{Pt}(\mathbf{z}, \mathbf{p}^\alpha) \neq \emptyset \wedge \phi o_i \in \text{Pt}(\mathbf{y})\})$</p>

Table 4.5: Transfer functions of object creation, direct write and property write statements

<p>Property delete: delete $x.p$</p> <p>Transfer function: (1) if $\text{Pt}(x) = 1$ and $\{\phi_{o_i} \in @0 \mid \phi_{o_i} \in \text{Pt}(x)\}$:</p> $G - \{(\langle @o_i, p^\alpha \rangle, \phi_{o_j}) \mid @o_i \in \text{Pt}(x) \wedge \phi_{o_j} \in \text{Pt}(\langle @o_i, p^\alpha \rangle)\}$ <p>(2) otherwise :</p> <p>(2.1) $G - \{(\langle x, p^\alpha \rangle, \phi_{o_i}) \mid \phi_{o_i} \in \text{Pt}(\langle x, p^\alpha \rangle)\} \cup \{(\langle x, p^d \rangle, \text{null})\}$</p> <p>(2.2) $G \cup \{(\langle \phi_{o_i}, p^* \rangle, \phi_{o_j}) \mid \phi_{o_i} \in \text{Pt}(x) \wedge \phi_{o_j} \in \text{Pt}(\langle \phi_{o_i}, p \rangle)\}$ $- \{(\langle \phi_{o_i}, p \rangle, \phi_{o_j}) \mid \phi_{o_i} \in \text{Pt}(x) \wedge \phi_{o_j} \in \text{Pt}(\langle \phi_{o_i}, p \rangle)\}$</p> <p>(2.3) $G \cup \{(\langle z, p^* \rangle, \phi_{o_i}) \mid z \in \text{Alias}(x) \wedge \text{Pt}(z, p) \neq \emptyset \wedge \phi_{o_i} \in \text{Pt}(\langle z, p \rangle)\}$ $- \{(\langle z, p \rangle, \phi_{o_i}) \mid z \in \text{Alias}(x) \wedge \text{Pt}(z, p) \neq \emptyset \wedge \phi_{o_i} \in \text{Pt}(\langle z, p \rangle)\}$</p>
<p>Property read: $x = y.p$</p> <p>Transfer function: $(G - \text{apset}(x)) \cup \{(\langle x, \phi_{o_i} \rangle) \mid o_i \in \text{lookup}(y.p)\}$</p>
<p>Function invocation: $x = y.f(a_1, a_2, \dots, a_n)$</p> <p>Transfer function: $(G - \text{apset}(x)) \cup \{\text{invoke}(G, F, \phi_{o_i}, a_1, a_2, \dots, a_n) \mid \phi_{o_i} \in \text{Pt}(y) \wedge F \in \text{lookup}(y, f)\}$</p>

Table 4.6: Transfer functions of property delete, property read and function invocation statements

erty write statement because an abstract object may summarize multiple run-time objects; however, use of in-construction objects and access path edges enable strong updates in our analysis. In the points-to graph G , if x only refers to one object and the object is an in-construction object, we know that x refers to a specific concrete object. The analysis then performs strong updates on the property reference edges by removing the points-to edges in G denoting $@o_i.p$ (if they exist) and adding the new edges implied by $Pt(y)$. In other cases (i.e., the cardinality of $Pt(x)$ is more than 1 or x refers to an abstract object), we use access path edges to enable strong updates on property write statements. First, the access path of $x.p$ can be strongly updated by removing the access path edges in G denoting $x.p$ (if they exist) and adding the new edges (e.g., $(\langle x, p \rangle, o_j)$ where o_j is referred to by y). Second, the object(s) x points to are weakly updated (e.g., the edge $(\langle o_i, p^* \rangle, o_j)$ is inserted if x points to o_i and y points to o_j). The property reference edges are inserted with the $*$ annotation because the property write statement may not affect all variables pointing to the updated object. Last, the access path edges of the variables that have a may alias relation to x need to be weakly updated. For example, $(\langle z, p^* \rangle, o_i)$ is inserted to G if z may be an alias of x , and there exists at least an edge denoting $z.p$ (with or without annotation).

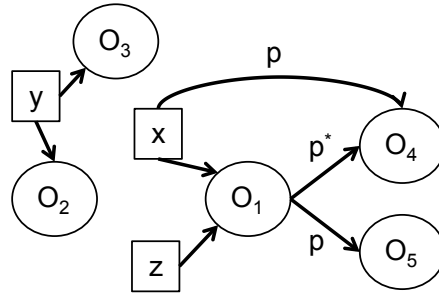


Figure 4.16: Property write example: input points-to graph

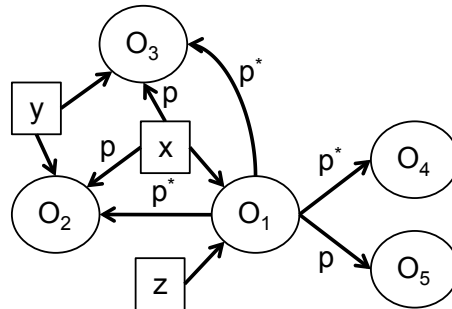


Figure 4.17: Property write example: updated points-to graph

Figures 4.16 and 4.17 show an example of the effects of a property write statement on the points-to graph. Figure 4.16 illustrates the input points-to graph for the property write statement $x.p = y$. In Figure 4.17, our analysis performs a strong update on the access path $x.p$ (i.e., delete $(\langle x, p \rangle, O_4)$ and add $(\langle x, p \rangle, O_2)$, $(\langle x, p \rangle, O_3)$) and inserts the edges $(\langle O_1, p^* \rangle, O_2)$, $(\langle O_1, p^* \rangle, O_3)$ (i.e., weak updates).

Property delete delete $x.p$. The transfer function of the delete statement is similar to the property write statement. Our analysis strongly updates the access path edges by removing the existing edges and adding a new edge (i.e., $\langle x, p^d \rangle, \text{null}$) that denotes x must not have a local access path $x.p$. When performing weak updates on the property reference edges of an object o_i that is referred to by x , all existing edges denoting $o_i.p$ should be annotated by $*$ because the property named p may not exist locally for o_i . The same rule applies when updating the access path edges of the aliases of x .

Property read $x = y.p$. JavaScript may seek the prototype-based inheritance when an object property is read, which is asymmetric to writing and deleting properties. Inaccurate model of property lookups may result in significant reduction of overall analysis practicality (e.g., when reading property p of an object, an analysis may report all properties named p in the prototype chain of the object to ensure analysis soundness); thus, we have designed a more precise property lookup mechanism enabled by our expanded points-to graph information.

```

Input: :  $v$  and  $p$ 
Output:  $P$  (accessible properties  $v.p$ )
1: if  $\text{Pt}(\langle v, p \rangle) \neq \emptyset$  then
2:    $P \cup \text{Pt}(\langle v, p \rangle) \cup \text{Pt}(\langle v, p^* \rangle)$ 
3:   return
4: else if  $\text{Pt}(\langle v, p^* \rangle) \neq \emptyset$  or  $\text{Pt}(\langle v, p^d \rangle) \neq \emptyset$  then
5:    $P \cup \text{Pt}(\langle v, p^* \rangle)$ 
6:   for each object  $\phi o$  in  $\text{lookup}(v, \text{\_proto\_})$  do
7:      $S.\text{push}(\phi o)$ 
8:   end for
9: else
10:  for each object  $\phi o$  in  $\text{Pt}(v)$  do
11:     $S.\text{push}(\phi o)$ 
12:  end for
13: end if
14: while  $S$  is not empty do
15:   $\phi o_i \leftarrow S.\text{pop}()$ 
16:   $P \cup \text{Pt}(\langle \phi o_i, p \rangle) \cup \text{Pt}(\langle \phi o_i, p^* \rangle)$ 
17:  if  $|\text{Pt}(\langle \phi o_i, p \rangle)| = 0$  and  $(\text{Pt}(\langle \phi o_i, \text{\_proto\_} \rangle) \neq \text{null}$  or
     $\text{Pt}(\langle \phi o_i, \text{\_proto\_}^* \rangle) \neq \text{null})$  then
18:    for each object  $\phi o_j$  in  $\text{Pt}(\langle \phi o_i, \text{\_proto\_} \rangle) \cup \text{Pt}(\langle \phi o_i, \text{\_proto\_}^* \rangle)$  do
19:       $S.\text{push}(\phi o_j)$ 
20:    end for
21:  end if
22: end while

```

Figure 4.18: Optimized object property lookup algorithm: $\text{lookup}(v, p)$

Figure 4.18 presents the details of our property lookup algorithm. This worklist algorithm iterates through all the accessible properties in the points-to graph when property p of variable v is read. It uses the access path in priority when looking up properties because these edges reflect the results of strong updates; property reference edges are used when access path edges are not available. Lines 1 to 12 initialize the algorithm upon three conditions. (1) If there exist access path edges for $v.p$ without annotation (i.e., property p must exist locally), the objects in the $\text{Pt}(\langle v, p \rangle)$ and $\text{Pt}(\langle v, p^* \rangle)$ are considered to be accessible properties (line 2) and the algorithm returns (line 3). (2) If there exist access path edges for $v.p$ with either annotation, the algorithm needs to lookup objects in the prototype chain. In this case, the objects in the $\text{Pt}(\langle v, p^* \rangle)$ (if $v.p^*$ exists) are considered to be accessible properties (line 5) and the algorithm pushes all the immediate prototype objects of v onto the worklist (lines 6 to 8). (3) Otherwise (i.e., no access path edge for $v.p$ exists), only the abstract objects are used for looking up so that all the objects in the $\text{Pt}(v)$ are pushed onto the worklist (lines 10 to 12). Lines 14 to 22 iterate the worklist. All the objects in $\text{Pt}(\langle \phi o, p \rangle)$ and $\text{Pt}(\langle \phi o, p^* \rangle)$ are considered to be accessible properties by our analysis (Line 16). Since an edge annotated with $*$ means that the property may not exist locally, the algorithm will continue looking up the prototype chain, until it reaches at least one points-to edge named p without annotation or the end of the prototype chain (Line 17 to 21). Thus, instead of finding all the properties named p in the prototype chain (i.e., `lookup_all(v, p)`), our algorithm can stop when it finds an existing property p (i.e., a property named p without annotation).⁵

This new property lookup algorithm `lookup(v, p)` mimics the run-time property lookup mechanism of JavaScript while still assuring the soundness of our analysis. For the example in Figure 4.17, `lookup(x, p)` results in O_2 and O_3 through the access path while `lookup(z, p)` results in O_2, O_3, O_4 and O_5 through the abstract object O_1 . In Table 4.6, the transfer function of the property read statements refers to this optimized object property lookup algorithm. Because we perform weak updates on the property read statements, similar to direct writes, the analysis removes all the access path edges of x from the points-to graph to ensure soundness.

Function invocation $x = y.f(a_1, a_2, \dots, a_n)$. The function invocation (e.g., $x = y.f(a_1, a_2, \dots, a_n)$) resolves for every receiver object pointed to by y . The invoked methods are determined by reading the property $y.f$ through our optimized lookup algorithm. Upon the return of function invocation, x is weakly updated by removing all its access path edges from G .

⁵The precision improvement of our property lookup algorithm is demonstrated via the experimental results on an analysis client in Section 4.4.

4.3.4 State Sensitivity

State sensitivity for JavaScript is a new form of context sensitivity derived from the notion of object sensitivity for languages such as Java. In object sensitivity, each function is analyzed separately for each object on which it may be invoked. For strongly typed languages like Java, often object sensitivity identifies objects by their creation sites. Calls of a function using two receiver objects (i.e., created at different sites) will result in two separate analyses of the function, even if the calls originated from the same call site. However, this is insufficient for JavaScript analysis, because object behavior may change dynamically at any program point during execution.

Obj-ref state, representing the notion of type for JavaScript objects, provides a more accurate approximation of JavaScript object behavior than object creation site. We present state sensitivity that uses obj-ref state as the context element. State-sensitive analysis analyzes each function separately for each obj-ref state on which it may be invoked. We define the following parameterized model for state sensitivity (i.e., k -state sensitivity).

- (i) 0-state sensitivity uses the receiver abstract object as a context element.
- (ii) 1-state sensitivity uses the receiver abstract object and its local properties as well as its prototype chain as a context element.
- (iii) k -state sensitivity uses all abstract objects in the context element of $(k-1)$ -state sensitivity and their local properties as well as their prototype chains as a context element.

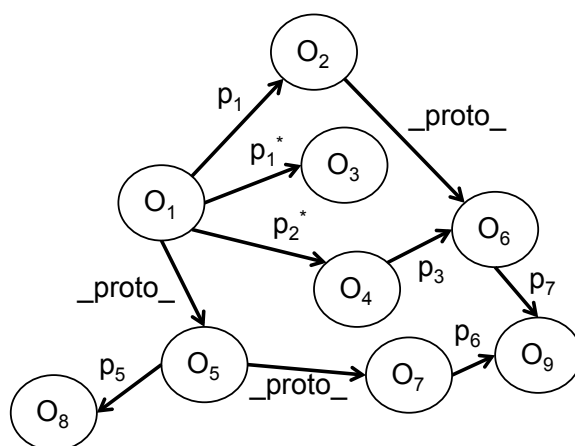


Figure 4.19: obj-ref state of O_1

By increasing the context depth of k , state-sensitive analysis, using a more expensive graph representation of obj-ref state as a context element, may result in more precise results.

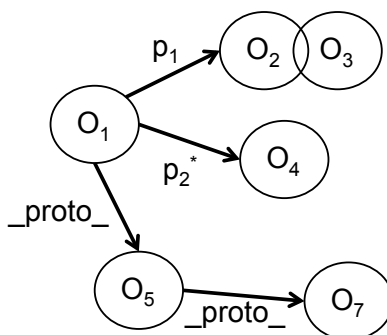


Figure 4.20: context element of 1-state sensitivity

0-state sensitivity is identical to 1-object sensitivity assuming they use the same object representation, while the context element of k -state sensitivity covers the longest access path of $O.p_1.p_2\dots p_k$ when each p_n ($n \in [1, k]$) is a local property. For example, Figure 4.19 shows the obj-ref state of object O_1 . If a function is invoked on O_1 , 1-object-sensitive and 0-state-sensitive analyses use the abstract object O_1 as the context element. The context element of 1-state-sensitive analysis includes O_1 , its local properties (i.e., O_2 , O_3 and O_4) and its chain of prototype objects (i.e., O_5 and O_7) shown in Figure 4.20, an approximation of the obj-ref state of O_1 . Note that the edges with the same local property name (annotated or not annotated) in the points-to graph are merged in the context element (e.g., $\langle O_1, p_1 \rangle, O_2$ and $\langle O_1, p_1^* \rangle, O_3$). In the example of Figure 4.19, 3-state sensitivity is capable of covering the complete graph representation of obj-ref state of O_1 , including the longest access path $O_1.p_2.p_3.p_7$. For more precise analysis results, state sensitivity would analyze each function separately for each complete obj-ref state on which it may be invoked. However, the graph representation of obj-ref state may contain many edges and nodes both locally and along prototype chains, which would be prohibitively expensive to use as a context element. Therefore, we use 1-state sensitivity in the current implementation of state-sensitive analysis. In our evaluation in Section 4.4, the term state-sensitive analysis refers to our implementation of 1-state-sensitive analysis.

4.3.5 Block Sensitivity

Our new points-to analysis algorithm is a fixed point calculation on the call graph, initialized with an empty points-to graph on entry to the JavaScript program, in which every SPBG is traversed in a flow-sensitive manner. Essentially, we have designed the points-to algorithm to emphasize precision for the obj-ref state information in the points-to graph and the SPBG to hide control flow not relevant to reference state updates.

More specifically, our analysis solves for the points-to graph on exit of each SPBG node. The transfer function for a node in the SPBG is one of two kinds: (i) for a state-update node perform strong update of the changed property, if possible (as in Tables 4.5 and 4.6), or

\cup	\emptyset	$\mathbf{v.p}^d$	$\mathbf{v.p}^*$	$\mathbf{v.p}$
\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
$\mathbf{v.p}^d$	\emptyset	$\mathbf{v.p}^d$	$\mathbf{v.p}^*$	$\mathbf{v.p}^*$
$\mathbf{v.p}^*$	\emptyset	$\mathbf{v.p}^*$	$\mathbf{v.p}^*$	$\mathbf{v.p}^*$
$\mathbf{v.p}$	\emptyset	$\mathbf{v.p}^*$	$\mathbf{v.p}^*$	$\mathbf{v.p}$

Table 4.7: Access path edges union rules

\cup	\emptyset	$\mathbf{o.p}^*$	$\mathbf{o.p}$
\emptyset	\emptyset	$\mathbf{o.p}^*$	$\mathbf{o.p}^*$
$\mathbf{o.p}^*$	$\mathbf{o.p}^*$	$\mathbf{o.p}^*$	$\mathbf{o.p}^*$
$\mathbf{o.p}$	$\mathbf{o.p}^*$	$\mathbf{o.p}^*$	$\mathbf{o.p}$

Table 4.8: Property reference edges union rules

(ii) for a state-preserving node perform a flow-insensitive analysis of the statements in that node, using an initial points-to graph (IN) and storing the fixed point reached in points-to graph OUT.

In a flow-sensitive points-to analysis, the points-to graph IN is formed as a union of the OUT points-to graphs of predecessor nodes. Due to the existence of annotations in our points-to graph, we apply specific union rules for the access path edges and property reference edges when two points-to graphs are unioned. For the access path edges (Table 4.7): (i) if access path $\mathbf{v.p}^\alpha$ does not exist in at least one predecessor, then $\mathbf{v.p}^\alpha$ does not exist after union; (ii) if $\mathbf{v.p}^d$ or $\mathbf{v.p}$ exists in both predecessors, then $\mathbf{v.p}^d$ or $\mathbf{v.p}$ respectively exists after union; (iii) otherwise, $\mathbf{v.p}^*$ exists after union. For the property reference edges (Table 4.8): (i) if $\mathbf{o.p}$ exists in both predecessors, then $\mathbf{o.p}$ exists after union; (ii) otherwise, if $\mathbf{o.p}$ or $\mathbf{o.p}^*$ exists in at least one predecessor, then $\mathbf{o.p}^*$ exists after union. These rules ensure analysis soundness when property lookup is performed.

4.3.6 State-sensitive Analysis as an Instantiation of JSBAF

We implemented our new points-to analysis with a client as the static component of JSBAF. This implementation choice enables more accurate object representation on polymorphic constructors. Specifically, the code pruning based on observed function calls and object

creations removes unexecuted statements in the constructors. Constructor polymorphism was handled by our improved object representation with specialized function bodies (i.e., objects created at the same allocation site with different sets of property names are represented as separate abstract objects). In addition, implementing state-sensitive analysis as an instantiation of JSBAF enabled us to perform evaluation on real-world JavaScript websites.

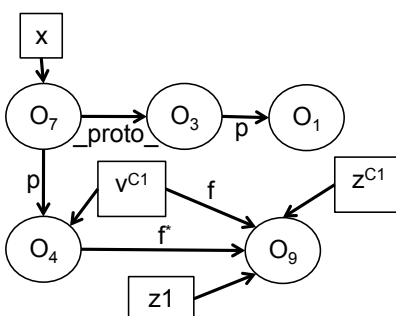


Figure 4.21: Blended state-sensitive points-to graph at line 10

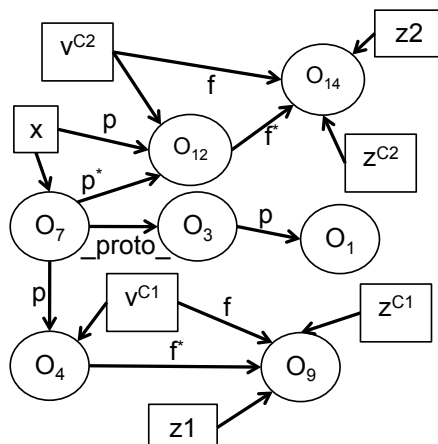


Figure 4.22: Blended state-sensitive points-to graph at line 15

Example. In comparison to the inaccurate points-to solution of a flow- and context-insensitive analysis for the JavaScript code in Figure 4.1, we now demonstrate the results of our state-sensitive points-to analysis in the context of blended analysis. Figures 4.21 and 4.22 show the points-to graphs obtained at lines 10 and 15, respectively. Because blended analysis executes the program and does not observe an object created by the constructor Y3, the code at line 5 is pruned so that our analysis does not generate the inaccurate node O_5 nor the edge $(\langle O_7, q \rangle, O_5)$. For the call statement at line 10, our 1-state-sensitive points-to analysis calculates the obj-ref state approximation of O_7 , namely $C1: \{O_7, p:O_4, _proto_:O_3\}$. Also, when looking up $x.p$ at line 10, our algorithm returns O_4 because there is no annotation on the property reference edge so that further lookup through the prototype chain is not necessary. Note that the points-to graph in Figure 4.22 is as precise as the run-time points-to graph (Figure 4.2).

At line 12, $x.p$ is strongly updated via the access path edge $(\langle x, p \rangle, O_{12})$. For the call statement at line 15, our points-to analysis calculates the obj-ref state approximation of O_7 , $C2: \{O_7, p: [O_4, O_{12}], _proto_: O_3\}$. Our points-to algorithm distinguishes this call site from line 10 because O_7 has a different obj-ref state here. The lookup of $x.p$ at line 15 follows the access path edge so that the node O_{12} is returned. Thus, in this example our analysis results in none of the inaccurate edges in the flow- and context-insensitive analysis (Figure 4.4) and reflects the run-time behavior of JavaScript objects (Figure 4.3).

4.4 Evaluation

In this section, we present experiments using JSBAF with our state-sensitive points-to analysis compared to an existing good points-to analysis [72], evaluating both with a REF client.

4.4.1 Experimental Design

REF Analysis. To evaluate the precision and performance of our points-to analysis, we implemented a JavaScript reference analysis (REF). The REF client calculates the set of objects returned by property lookup at a property read statement (i.e., $x = y.p$) or call statement (i.e., $x = y.p(\dots)$). For each of these statements s in a function being analyzed in calling context c , we compute $REF(s, c)$, the set of objects returned by a property lookup for each $o.p$ where o is pointed to by y . The cardinality of the REF set depends on the precision of the points-to graph and the property lookup operation; the smaller the set returned, the more useful for program understanding, for example.

In Figure 4.1, assume we add the function property

```
x.foo = function(){var a = this.p; return a;}
```

Effectively, `foo` returns the property lookup result for `this.p`. If `x.foo()` is called at line 11 before the state-update statement `x.p=new A()`, it will return O_4 . If `x.foo()` is called at line 13 after `x.p=new A()`, it will return O_{12} . For an analysis that is flow-insensitive or that cannot distinguish these call sites by calling context, the return value of each of these function calls will contain at least two objects (i.e., O_4 and O_{12}).

Comparison with points-to analysis in [72]. We use the term `CORR` to refer to a blended version of correlation-tracking points-to analysis [72] and its REF client, the same points-to algorithm we used for blended taint analysis (Section 3.2.1). To demonstrate the additional precision of our analysis over `CORR`, we applied the correlation extraction transformation to our JavaScript benchmarks before performing our points-to analysis. We use the term `CORRBSSS` to refer to a blended version of this augmented new points-to analysis and its REF

Website	CORR			CORRBSSS		
	1	2-4	≥ 5	1	2-4	≥ 5
facebook	38%	52%	10%	50%	47%	3%
google	32%	51%	17%	53%	42%	5%
youtube	41%	47%	12%	54%	41%	5%
yahoo	48%	46%	6%	52%	45%	3%
wikipedia	29%	45%	26%	43%	39%	18%
amazon	45%	52%	3%	46%	51%	3%
twitter	32%	53%	15%	39%	49%	12%
blogspot	35%	34%	31%	53%	36%	11%
linkedin	34%	49%	17%	44%	50%	6%
msn	40%	36%	24%	48%	37%	15%
ebay	30%	40%	30%	46%	40%	14%
bing	41%	34%	25%	54%	37%	9%
Average	37%	45%	18%	48%	43%	9%

Table 4.9: REF analysis precision

client. For each algorithm, an object property lookup returns a REF set whose cardinality $|\text{REF}(s, c)|$ is calculated. For CORR, the `lookup_all` approximate algorithm described in Section 4.3.3 is used. For CORRBSSS, we use our optimized lookup algorithm `lookup` in Figure 4.18.

Benchmarks. We conducted the experiments with the same set of benchmarks described in Section 3.2.2 (i.e., traces collected from 12 popular websites) and the experimental results were obtained on a 2.53GHz Intel Core 2 Duo MacBook Pro with 4GB memory running the Mac OS X 10.5 operating system.

4.4.2 Experimental Results

Improved REF Precision. Table 4.9 shows the REF client results for the 012 websites. Columns 2-4 present the results for CORR and columns 5-7 present the results for CORRBSSS. For each website, columns 2 & 5, 3 & 6, and 4 & 7 in Table 4.9 correspond to the percentage of property lookup statements that return 1 object, 2-4 objects, and more than 4 objects, respectively. The result shown for each website is averaged over the corresponding percentage numbers for all of the webpages in that domain. For example, the 38% entry for *facebook* in column 2 is the average for CORR over the 27 webpages analyzed returning only 1 object.

Comparing columns 2-4 with 5-7 in Table 4.9 for each website, we see the relative precision improvement of CORRBSSS over CORR. For REF analysis, the best result is that the lookup returns only one object and the property lookup is more precise if the number of objects returned is smaller. On average over all the websites, CORR reported 37% of the property lookup

Website	CORR	CORRBSSS	Overhead
facebook	17.4	45.9	163%
google	13.0	30.4	134%
youtube	31.2	75.3	141%
yahoo	28.5	54.1	90%
wiki	16.0	40.1	151%
amazon	15.1	24.2	61%
twitter	38.1	94.5	148%
blog	15.9	42.4	137%
linkedin	27.8	62.0	167%
msn	34.4	57.9	68%
ebay	8.3	27.2	227%
bing	22.1	50.4	128%
Average	22.3	50.3	135%

Table 4.10: REF analysis cost (in seconds) on average per webpage

statements were resolved to a single object, while **CORRBSSS** improved this metric to 48%, a significant improvement. In addition, REF analysis results may become too approximate to be useful if too many objects are returned. Although 18% of the statements returned more than 4 objects for **CORR**, **CORRBSSS** reduced that number to 9%. These improved precision results indicate the potential for greater practical use of state-sensitive points-to information by client analyses.

We also investigated the average number of objects returned by a property lookup statement. For each website, we calculated the number of objects per statement on average over all its webpages. Over all the benchmarks, **CORR** produced on average 2.8 objects and **CORRBSSS** only reported on average 2.3 objects. Intuitively, this means that on average fewer objects at each property lookup statement must be examined to gain better understanding of the code.

REF Performance. An JavaScript analysis is practical if it scales to real-world websites. Because **CORRBSSS** is partially flow-sensitive and context-sensitive, it is important to demonstrate that this analysis is scalable. Table 4.10 shows the time performance of **CORR** versus **CORRBSSS**.⁶ Columns 2 and 3 present the average webpage analysis time for each website, averaging over all of its webpages. Both **CORR** and **CORRBSSS** completely analyzed all the benchmark programs. On average over all the websites, **CORR** completely analyzed a webpage in 22.3 seconds, while **CORRBSSS** did so in 50.3 seconds, incurring 135% average time overhead per webpage.

Discussion. We collected data characterizing benchmark program structure and complexity

⁶The time cost in Table 4.10 reflects the performance of the static phase of blended analysis. In the experiments, the dynamic phase of **CORR** and **CORRBSSS** is the same for both analyses.

Website	No. of functions	% of functions w/ update(s)	% of state-update stmt	No. of contexts
facebook	2123	9%	8%	4.0
google	1002	17%	6%	6.7
youtube	1329	7%	6%	3.9
yahoo	3810	4%	4%	2.4
wiki	270	10%	19%	4.8
amazon	729	6%	6%	1.9
twitter	618	15%	5%	3.4
blog	583	14%	14%	6.1
linkedin	920	8%	11%	3.6
msn	1537	8%	8%	2.8
ebay	581	18%	13%	7.5
bing	1131	7%	11%	4.9
Average	1219	10%	9%	4.3

Table 4.11: Benchmark and context statistics

to relate these characteristics to observed analysis precision and performance. The entries in Table 4.11 all represent averages per webpage that are averaged over an entire website. Column 2 shows the average number of functions in a JavaScript program. Column 3 shows the percentage of functions containing at least one state-update statement. Column 4 shows the percentage of statements that are state-update statements. Column 5 shows the number of contexts produced by CORRBSSS as a multiplier for column 2. On average over all the websites, 10% of the functions contained local state-update statement(s); these averages ranged from 4% for *yahoo* to 18% for *msn*. This suggests that the state-update statements are localized in a relatively small portion of the JavaScript program (e.g., in constructor functions). On average over all the websites, 9% of the statements were identified as state-update statements. The relatively small number of state-update statements means that our SPBG contained many fewer nodes than the corresponding CFGs; therefore the flow-sensitive analysis was more practical in cost on the SPBGs.

Now we compare the analysis precision observed in Table 4.9 with the number of contexts generated on average per function per page (column 5 in Table 4.11) to observe the effect of state sensitivity. *google*, *blog*, and *ebay* were the websites for which CORRBSSS improved precision the most, whereas *amazon*, *yahoos*, *twitter*, and *msn* were the websites for which CORRBSSS produced similar results to CORR. For the former websites, CORRBSSS generated the greatest number of contexts per function per webpage. For the latter websites, CORRBSSS generated the fewest. We observe strong correlation between the precision gain and the number of contexts generated by CORRBSSS, demonstrating that state sensitivity significantly increased analysis precision on these benchmarks, and suggesting that state sensitivity is an effective form of context sensitivity for JavaScript analysis.

As shown in Table 4.10, the CORRBSSS time overhead differed significantly for different web-

sites, from 61% (*amazon*) to 227% (*ebay.com*). We investigate several program characteristics to reason about such differences. First, the SPBGs created by CORRBSST determine the efficiency of the flow-sensitive analysis. On average over all the websites, an SPBG was comprised of about 6 nodes, explaining why CORRBSST scaled on real websites. Functions with large numbers of nodes in their SPBG usually contained multiple state-update statements and complex control flow. The largest number of nodes for an SPBG was 23 in *linkedin*. Second, the websites with the least performance overhead from CORRBSST were *amazon*, *msn* and *yahoo*. These websites contained a relatively small percentage of state-update statements (i.e., all below average) and CORRBSST generated the lowest number of contexts for them. The website that incurred the most overhead (i.e., *ebay*) contained 13% update statements, (i.e., the third highest percentage in our benchmarks), and the greatest number of contexts per function (i.e., 7.5) generated by CORRBSST. These results support the reasoning that more complex block structure and more context comparisons contribute to the higher overhead for CORRBSST.

Chapter 5

Adaptive Context-sensitive Analysis

In addition to object-oriented features such as prototype-based inheritance, JavaScript applications exhibit other programming paradigms such as functional (e.g., first-class functions) and procedural programming. The multi-paradigm feature of JavaScript poses challenges for designing effective program analysis techniques, because each technique (e.g., context sensitivity) often works well for a specific programming paradigm. In this chapter, we present a novel analysis that selectively applies a context-sensitive analysis depending on programming paradigms of a function. We first perform an empirical study on JavaScript benchmarks to compare the precision of different whole-program context-sensitive analyses and conclude that a specific context-sensitive analysis is often effective only on a portion of a JavaScript program. We then present our two-staged adaptive context-sensitive analysis for JavaScript using heuristics to select from various context-sensitive approaches for a function and discuss the experimental results.¹

5.1 Empirical Study of JavaScript Context Sensitivity

There is no theoretical comparison between the functional and call-strings approaches to context sensitivity that proves one better than the other. Therefore, we study the precision of different context-sensitive analyses in practice based on experimental observations. Such comparisons have been conducted for call-site and object sensitivity on Java [39, 68] as well as JavaScript [33] applications. Object sensitivity produced more precise results for Java benchmarks, while there was no clear winner across all benchmarks for JavaScript. The latter observation motivated us to perform an in-depth, fine-grained, function-level study which led to our design of a new context-sensitive analysis for JavaScript.

¹Part of the contents presented in this chapter was published in [79].

5.1.1 Experimental Design

Hypothesis. Our hypothesis is that a specific context-sensitive analysis may be more effective on a portion of a JavaScript program (i.e., some functions), while another kind of context sensitivity may produce better results for other portions of the same program. To test this hypothesis, we compared the precision of different context-sensitive analyses at the function level (i.e., we collect the results of different analyses for a specific function and compare their precision). In contrast, all previous work [39, 68, 33] reported overall precision results per benchmark program.

Analyses for comparisons. We compared across four different flow-insensitive analyses to study their precision. The baseline analysis is an analysis that applies the default context-sensitive analysis for JavaScript in *WALA* (i.e., only uses 1-call-site-sensitive analysis for the constructors to name abstract objects by their allocation sites and for nested functions to property access variables accessible through lexical scoping). In the implementation, the other three analyses (i.e., 1-call-site, 1-object and 1st-parameter) all apply the default analysis. In principle, these analyses are at least as precise as the baseline analysis for all functions.

Analysis clients and precision metrics. We compare precision results on two clients of points-to analysis. The first client, Pts-Size, is a points-to query returning the cardinality of the set of all values of all local variables in a function (i.e., the total number of abstract objects pointed to by local variables). The second client, REF that we also used for evaluating our state-sensitive analysis in Section 4.4, is a points-to query returning the cardinality of the set of all property values in all property reads (e.g., $x = y.p$) or call statements (e.g., $x = y.p(\dots)$) in a function (i.e., the total number of abstract objects returned by all the property lookups). For both clients, if analysis A_1 produces a smaller result than another analysis A_2 for a function `foo`, we say that A_1 is more precise than A_2 for `foo`.

Benchmarks. We conduct our comparisons on the same set of benchmarks used for the study performed by Kashyap *et al.* [33]. There are in total 28 JavaScript programs divided into four categories: *standard* (i.e., from *SunSpider*² and *V8*³), *addon* (i.e., Firefox browser plugins), *generated* (i.e., from the Emscripten LLVM test suite⁴) and *opensrc* (i.e., open source JavaScript frameworks). There are seven programs in each benchmark category. Details of the benchmark programs were provided in Kashyap *et al.* [33].

The results of our empirical study were obtained on a 2.4 GHz Intel Core i5 MacBook Pro with 8GB memory running the Mac OS X 10.10 operating system.

²<http://www.webkit.org/perf/sunspider/sunspider.html>

³<https://v8.googlecode.com/svn/data/benchmarks/v7/run.html>

⁴<http://kripken.github.io/emscripten-site/>

5.1.2 Results

We ran each analysis of a benchmark program under a time limit of 10 minutes. The baseline and 1-call-site-sensitive analyses finished analyzing all 28 programs under the time limit. 1-object-sensitive analysis timed out on 4 programs (i.e., *linq_aggregate*, *linq_enumerable* and *linq_functional* in the *opensrc* benchmarks and *fourinarow* in the *generated* benchmarks), while 1st-parameter-sensitive analysis timed out on 2 programs (i.e., *fasta* and *fourinarow* in the *generated* benchmarks).

Figures 5.1 and 5.2 show the relative precision results for Pts-Size and REF, respectively. In both figures, the horizontal axis represents the results from four benchmark categories (i.e., *standard*, *addon*, *generated* and *opensrc*) and the vertical axis represents the percentages of functions in each benchmark category on which an analysis produces the *best* results (i.e., more precise results than those from all other three analyses) or *equally precise* results. We consider the results of an analysis as equally precise as follows. (i) Baseline analysis is equally precise on a function if its results are as precise as each of the other three context-sensitive analyses. (ii) 1-call-site-sensitive, 1-object-sensitive or 1st-parameter-sensitive analysis is equally precise on a function if the results are more precise results than the baseline analysis, and if the analysis does not produce the best results but the results are at least as precise as the other two context-sensitive analyses. This definition indicates that multiple context-sensitive analyses (e.g., 1-call-site-sensitive and 1-object-sensitive analyses) may produce equally precise results on a function.

For example, the left four bars in Figure 5.1 present the precision results for the *addon* benchmarks for the Pts-Size client. The *baseline_equal* bar (i.e., the leftmost) shows that baseline analysis achieved as precise results as those from all three other analyses for 64% of the functions in the *addon* benchmarks, indicating context sensitivity does not make much difference for more than two thirds of the functions in these programs for the Pts-Size client. The *1-call-site_best* and *1st-parameter_best* bars (i.e., the parts of the second and fourth bars from left filled with patterns) show that 1-call-site-sensitive and 1st-parameter-sensitive analyses produced more precise results than all other analyses for 9% and 1.5% of the functions in the *addon* benchmarks, respectively. The *1-object_best* result missing from the third bar from left indicates that 1-object-sensitive analysis failed to produce the most precise results for any function in *addon* benchmarks. Nevertheless, the *1-object_equal* bar shows 1-object-sensitive analysis achieved equally precise results with 1-call-site-sensitive and/or 1st-parameter-sensitive analyses for 25% of the functions in the *addon* benchmarks. Comparing with the *1-call-site_equal* (26%) and *1st-parameter_equal* (1%) bars, we can predict that 1-call-site-sensitive and 1-object-sensitive analyses had similar precision on a quarter of the functions in the *addon* benchmarks.

The *baseline_equal* bars in Figure 5.1 show that analysis of a large percentage of functions in the benchmarks does not benefit from context sensitivity in terms of the Pts-Size results (i.e., from 32% for the *generated* benchmarks to 64% for the *addon* benchmarks). Also, 1-call-site-sensitive analysis had relatively consistent impact in the benchmarks, achieving

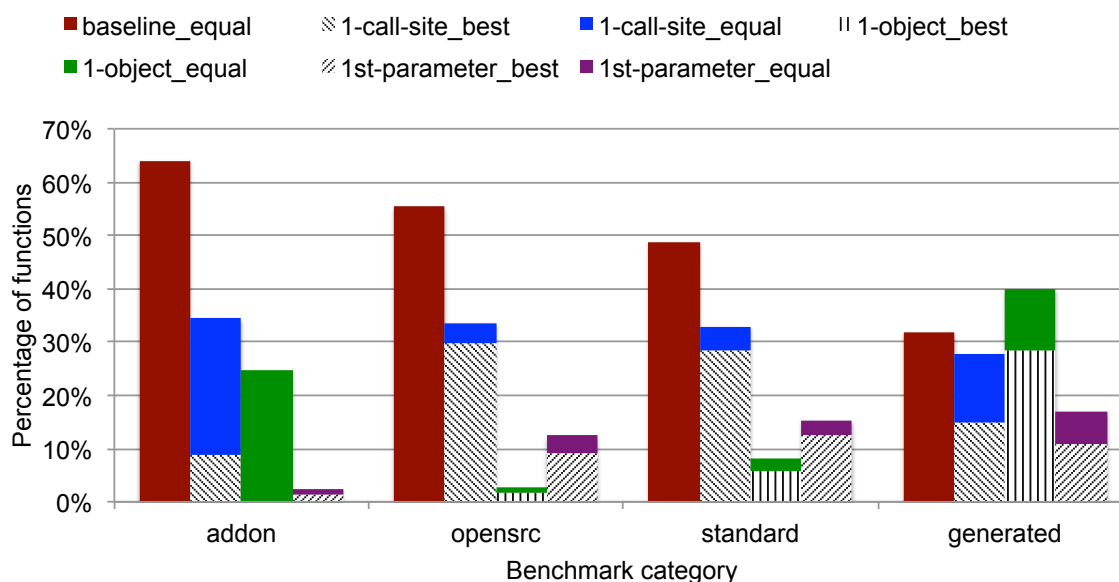


Figure 5.1: Precision results for the Pts-Size client

best or equally precise results for around 30% functions across all benchmark categories. In contrast, the precision of 1-object-sensitive analysis results seems dependent on the benchmark. Having little impact on the precision of the *opensrc* benchmarks, 1-object-sensitive analysis produced best results for 29% of the functions in the *generated* benchmarks with an additional 11% of the functions achieving equally best results. 1st-parameter-sensitive analysis, less studied in previous work, produced best results for about 10% functions in the *opensrc*, *standard* and *generated* benchmarks, a reasonable technique to improve precision for these programs. It is also interesting to learn from Figure 5.1 that different context-sensitive analyses may produce equally precise results on many functions in some benchmark categories. 1-call-site-sensitive and 1-object-sensitive analyses produced equally best results for 25% functions in the *addon* benchmarks. 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses produced equally best results for 3% functions in the *generated* benchmarks.

Recall that the REF client uses data from different parts of the points-to results than the Pts-Size client; in addition, the REF client may query the points-to results (i.e., all the property lookup statements in a function) less frequently than the Pts-Size client (i.e., all local variables in a function). Overall in Figure 5.2, context sensitivity improves precision less over baseline analysis for the REF than for the Pts-Size client. Baseline analysis produced as precise results as all other three analysis for more than 50% of the functions in all benchmark categories. About 96% of the functions in the *addon* benchmarks did not benefit from any context-sensitive analysis over the baseline analysis. 1-call-site-sensitive analysis achieved dramatically better results for the Pts-Size client than REF client in *addon*, *opensrc* and

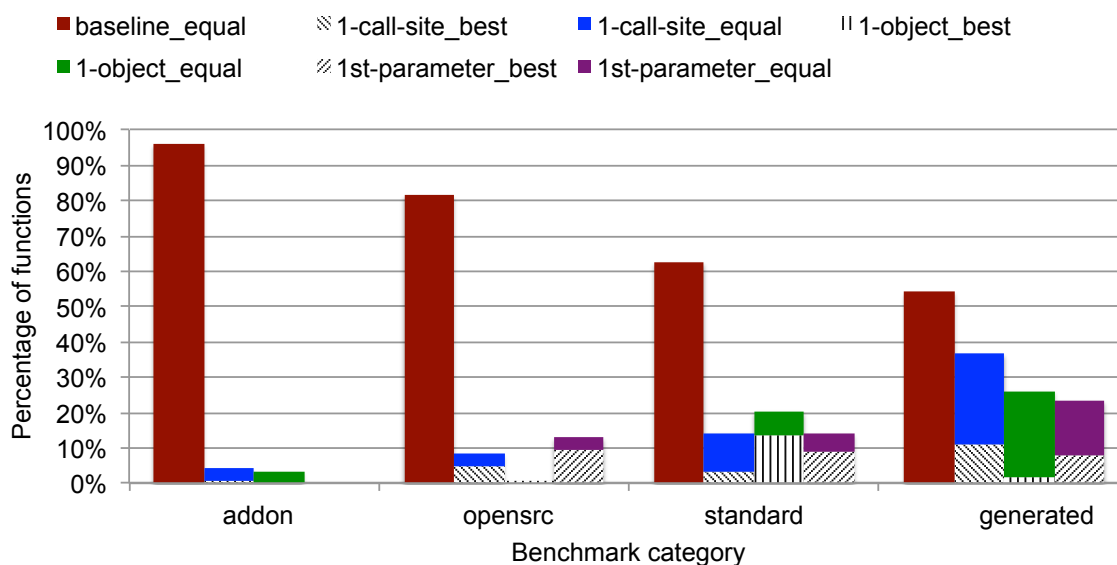


Figure 5.2: Precision results for the REF client

standard benchmarks. 1-object-sensitive analysis also achieved much better results for the Pts-Size client than the REF client in the *generated* benchmarks. On the other hand, 1st-parameter-sensitive analysis still remains effective in the *opensrc*, *standard* and *generated* benchmarks.

Summary. First, the effectiveness of specific context-sensitive analysis for JavaScript functions is sensitive to the programming paradigms. For example, 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses each produced best results on a large percentage of functions in the programs from the *generated* benchmarks. Second, the precision of context-sensitive analysis also depends on the analysis client.

Based on these observations, we believe JavaScript programs can benefit from an adaptive context-sensitive analysis that chooses an appropriate context-sensitive analysis for a specific function. We used these observations as guidance to design our new analysis.

5.2 Adaptive Context-sensitive Analysis

A context-sensitive analysis is designed to be useful for a specific programming paradigm. For example, object-sensitive analysis target the class-based model of object-oriented languages. The results from Section 5.1 indicate that JavaScript functions in one program may benefit from different context-sensitive analyses depending on the coding style of the functions. In this section, we present our adaptive context-sensitive points-to analysis. Starting with an overview of the adaptive analysis workflow, we will then discuss each major component of

this two-staged analysis including function characteristics and context selection heuristics.

5.2.1 Overview

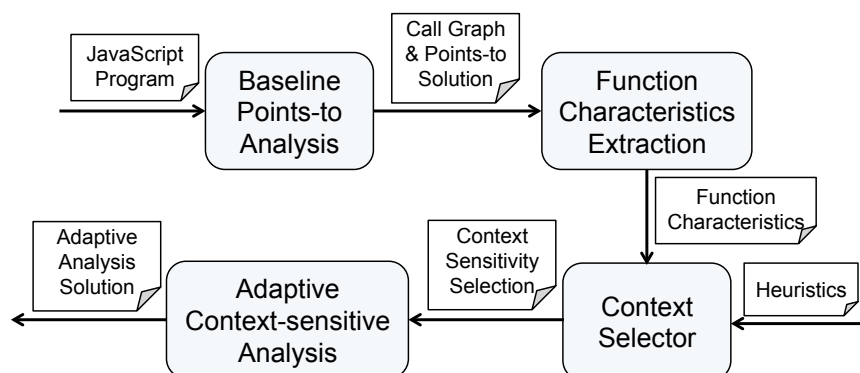


Figure 5.3: Workflow of two-staged adaptive context-sensitive analysis

To enable function-level context-sensitive analysis, we have designed an adaptive context-sensitive analysis that selectively applies specific context sensitivity per function. Figure 5.3 shows the workflow of our two-staged adaptive analysis. The first stage of our analysis performs a baseline points-to analysis for the target JavaScript program. Recall that the baseline points-to analysis is a mostly context-insensitive analysis so that despite of the imprecise results it may produce, it is often inexpensive in terms of performance for analyzing many JavaScript applications. Given the results of the baseline points-to analysis (i.e., call graph and points-to solution), we extract several characteristics of each function in the program. Each function characteristic is relevant to the precision of context-sensitive analysis for a specific client (e.g., the number of call sites that invoke `foo`). Having used machine learning to develop the heuristics to select a specific context-sensitive analysis for a function based on its function characteristics, the context selector chooses an appropriate context-sensitive analysis for each function in the program. Last, the second stage of our analysis performs an adaptive context-sensitive analysis based on the context sensitivity selection to produce precise results for the target program.

Adaptive context-sensitive analysis is the first analysis for JavaScript that selectively uses multiple context-sensitive analyses to analyze a program when context sensitivity may help improve precision. In our implementation, we have explored the context sensitivity selection between 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses. However, this workflow of adaptive analysis can generally be applied to select from other context-sensitive analyses as needed. The function characteristics also can be customized to accommodate the need of applying new context sensitivity and/or analysis client.

5.2.2 Function Characteristics

For a JavaScript function, we extracted characteristics from the baseline analysis results that are relevant to the precision of context-sensitive analyses for a specific client. The goal is to extract function characteristics that (i) intuitively are relevant to the precision of a specific analysis for a particular client, and (ii) do not require more costly analysis than a baseline points-to analysis. Table 5.1 shows that for a JavaScript function `foo`, we extract eight function characteristics (i.e., FC1, FC2, ..., FC8). Each function characteristic is related to the precision of a specific context-sensitive analysis (i.e., FC1-FC3, FC4-FC5, and FC6-FC8 are related to the precision of 1-call-site, 1-object and 1st-parameter, respectively).

	1-call-site	1-object	1st-parameter
context element approximations	FC1-CSNum	FC4-RCNum	FC6-1ParNum
	FC2-EquivCSNum		
client-related metrics	FC3-AllUse	FC5-ThisUse	FC7-1ParName
			FC8-1ParOther

Table 5.1: Function characteristics

For a specific context-sensitive analysis, we extracted two kinds of function characteristics: context element approximations and client-related metrics. A context element approximation predicts the number of distinct context elements generated for a function by a context-sensitive analysis, which determines its ability to distinguish between function calls. A client-related metric predicts the effectiveness of a context-sensitive analysis on a JavaScript function for a particular client. FC1-FC2, FC4 and FC6 in Table 5.1 are the context element approximations we designed for 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses, respectively. For example, FC4-RCNum presents an approximation of the number of receiver objects on which a function is called, computed from the baseline points-to results. FC3, FC5 and FC7-FC8 are the client-related metrics we designed for the Pts-Size client⁵ for 1-call-site, 1-object and 1st-parameter sensitivity, respectively. For example, FC5-ThisUse measures the usage frequency of the `this` object in the function body, because frequent use of the `this` object indicates that the precision of the Pts-Size client on the function depends on how accurately the `this` object is analyzed. Because 1-object-sensitive analysis potentially analyzes the `this` object more precisely, we use FC5-ThisUse to predict the effectiveness of 1-object-sensitive analysis on a function for the Pts-Size client. We now will define each function characteristic.

⁵We use the Pts-Size client to demonstrate the effectiveness of our approach because the empirical results in Section 5.1 suggest that the Pts-Size client is relatively more sensitive to the choice of context-sensitive analyses.

Function characteristics for 1-call-site-sensitive analysis. Recall that a 1-call-site-sensitive analysis uses the immediate call site of a function as the context element. We define FC1, the CSNum metric, as follows:

- FC1-CSNum: for function `foo`, the number of call sites that invoke `foo` in the baseline call graph G .

Although FC1 approximates the number of context elements that a 1-call-site-sensitive analysis would generate for `foo`, this metric may not be directly relevant to the precision of 1-call-site-sensitive analysis. Intuitively, if function `foo` is invoked from two call sites `CS1` and `CS2`, the analysis precision on `foo` is not likely to benefit from distinguishing between these two call sites if the parameters of `CS1` and `CS2` have the same values because these parameters are used in `foo` as local variables. More precisely, we define two call sites, `CS1:p0.foo(p1, p2, . . . , pn)` and `CS2:p'0.foo(p'1, p'2, . . . , p'n)`, to be *equivalent* if for each pair of receiver objects and parameters (i.e., p_i and p'_i) in `CS1` and `CS2`, the points-to sets of p_i and p'_i are the same. We then define FC2, the EquivCSNum metric using this definition of equivalent call sites, as follows:

- FC2-EquivCSNum: for function `foo`, the number of equivalence classes of call sites that invoke `foo` in the baseline call graph G .

Recall that the Pts-Size client calculates the cardinality of the set of abstract objects to which a local variable of `foo` may point. Intuitively, the precision of the Pts-Size client depends on the receiver object or parameters that are frequently used as local variables in the function body. For example, if a parameter `p` is never used in `foo`, even if 1-call-site-sensitive analysis distinguishes call sites that pass different values of `p`, the results of the Pts-Size client may not be different because `p` is never used locally. Theoretically, 1-call-site sensitivity may distinguish objects passed through any parameter as well as receiver objects via call sites. We define FC3, the AllUse metric, as follows:

- FC3-AllUse: for function `foo`, the total number of uses of the `this` object and all parameters.

Function characteristics for 1-object-sensitive analysis. 1-object-sensitive analysis distinguishes calls to a function if they correspond to different receiver objects. To approximate the number of context elements generated by 1-object-sensitive analysis for function `foo`, we define FC4, the RCNum metric, as follows:

- FC4-RCNum: for function `foo`, the total number of abstract receiver objects from all call sites that invoke `foo` in the baseline call graph G .

Naturally, 1-object-sensitive analysis would be effective on functions implemented with the object-oriented programming paradigm. The behavior of these functions is dependent on the objects on which they are called. Uses of the `this` object in a function is common in the object-oriented programming paradigm and 1-object-sensitive analysis should produce relatively precise results. We define FC5, the ThisUse metric, as follows:

- FC5-ThisUse: for function `foo`, the total number of uses of the `this` object.

Function characteristics for 1st-parameter-sensitive analysis. The *ith*-parameter sensitivity is designed to be effective when a specific parameter (e.g., the first parameter for 1st-parameter-sensitive analysis) has large impact on analysis precision. 1st-parameter-sensitive analysis uses the objects that the 1st parameter points to as context elements. We define FC6, the 1ParNum metric, as follows:

- FC6-1ParNum: for function `foo`, the total number of abstract objects to which the 1st parameter may point from all call sites that invoke `foo` in the baseline call graph G .

If a parameter `p` is frequently used in a function, it may be more important to apply context-sensitive analysis on `p` than on the receiver object. Also, if `p` is used as a property name in dynamic property accesses, using context sensitivity to distinguish the values of `p` significantly improves analysis precision [72]. We define FC7, the 1ParName metric, and FC8, the 1ParOther metric, as follows:

- FC7-1ParName: for function `foo`, the total number of uses of the 1st parameter as a property name in dynamic property accesses.
- FC8-1ParOther: for function `foo`, the total number of uses of the 1st parameter not as a property name.

Function characteristics extraction. For a function `foo` with n parameters, we extract three characteristics for 1-call-site sensitivity (i.e., CSNum, EquivCSNum and AllUse), two characteristics for 1-object sensitivity (i.e., RCNum and ThisUse). In our adaptive context-sensitive analysis, we actually apply *ith*-parameter-sensitive analysis for a function whose precision relies on how accurately the *ith* parameter is analyzed. Therefore, for each parameter of `foo`, we extract three function characteristics: *iParNum*, *iParName* and *iParOther*. In total, there are $6+3n$ function characteristics for `foo`.

5.2.3 Heuristics

The function characteristics defined in Section 5.2.2 are intuitive and easy to calculate from the baseline points-to graph and call graph. Nevertheless, it is still not clear how these

function characteristics are related to the precision of a context-sensitive analysis. In this section, we use empirical data to design the heuristics that define the relations between function characteristics and analysis precision.

Our goal is to select an appropriate analysis for a function given the set of its function characteristics. The heuristics are not obvious given that there are multiple context-sensitive analysis choices. To design useful heuristics, we first compared the precision of a pair of analyses on the function level and observed the impact of a subset of function characteristics on these two analyses. We then applied these heuristics to adaptively choose an appropriate context-sensitive analysis using the function characteristics (Section 5.2.4). More specifically, for the Pts-Size results from the benchmarks (Section 5.1), we compared the precision between all 2-combinations of baseline, 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses and derived the heuristics to select an analysis from each of the combinations.

For example, to choose between the baseline analysis and 1-call-site-sensitive analysis, we obtained the relevant subset of function characteristics (i.e., FC1-FC3) and the Pts-Size results of baseline and 1-call-site-sensitive analyses for each function `foo` in the benchmarks. If the Pts-Size result from 1-call-site-sensitive analysis is more precise than baseline analysis, 1-call-site sensitivity should be chosen when analyzing `foo`; otherwise, baseline analysis should be chosen. Given the list of function characteristics and corresponding analysis choices on the benchmark functions, we first used a machine learning algorithm⁶ to get the relationship (i.e., presented as a decision tree) between the function characteristics and analysis choice. We then manually adjusted the initial decision tree based on domain knowledge to decide on the heuristic. Specifically, we collapsed the branches of the decision tree produced by the machine learning algorithm, in order to ensure that the heuristic is intuitive and easy to interpret while the classifications still maintain good accuracy.

We report the accuracy of an analysis choice (e.g., 1-call-site-sensitive analysis) using the standard information retrieval metrics of *precision* and *recall*. Assuming $S1$ is the set of all functions in the benchmarks where 1-call-site-sensitive analysis produces more precise results than baseline analysis and $S2$ is the set of functions where 1-call-site-sensitive analysis is chosen by the heuristic. The precision of 1-call-site sensitivity classification is computed as

$$P_{1\text{-call-site}} = \frac{|S1 \cap S2|}{|S2|}$$

and the recall is computed as

$$R_{1\text{-call-site}} = \frac{|S1 \cap S2|}{|S1|}.$$

The *balanced F-score*, the harmonic mean of the precision and recall, is computed as

⁶We used the C4.5 classifier [55] implemented in Weka data mining software (<http://www.cs.waikato.ac.nz/ml/weka/>) to derive the initial decision tree.

$$F_{1\text{-call-site}} = 2 \times \frac{P_{1\text{-call-site}} \times R_{1\text{-call-site}}}{P_{1\text{-call-site}} + R_{1\text{-call-site}}},$$

where $F_{1\text{-call-site}}$ has its best value at 1 and worst score at 0 for choosing 1-call-site-sensitive analysis by this heuristic.

Figures 5.4 to 5.10 show the derived heuristics to make a choice between each pair of the analyses using function characteristic values. We discuss each pair of the analyses in turn:

<pre>FC2-EquivCNum = 1: baseline FC2-EquivCNum > 1: 1-call-site</pre>
--

Figure 5.4: Heuristic: baseline vs. 1-call-site

Baseline vs. 1-call-site. Three function characteristics (i.e., FC1-FC3) are relevant to the precision of 1-call-site-sensitive analysis for the Pts-Size client. For 1-call-site-sensitive analysis to produce more precise results than baseline analysis, the prerequisite is that there is more than one distinct 1-call-site-sensitive context element (i.e., FC1 > 1). Figure 5.4 shows the heuristic to choose between baseline and 1-call-site-sensitive analyses. 1-call-site-sensitive analysis is chosen over baseline analysis for a function `foo` if there is more than one equivalence class of call sites that invoke `foo` (i.e., FC2 > 1). This result indicates that the effectiveness of 1-call-site sensitivity depends on its ability to distinguish call sites with different receiver objects or different corresponding parameters. The balanced F-scores for baseline and 1-call-site-sensitive analyses in this heuristic are 0.46 and 0.8, respectively.

<pre>FC5-ThisUse = 0: baseline FC5-ThisUse > 0: 1-object</pre>

Figure 5.5: Heuristic: baseline vs. 1-object

Baseline vs. 1-object. FC4 and FC5 are relevant to the precision of 1-object-sensitive analysis for the Pts-Size client. For 1-object-sensitive analysis to produce more precise results than baseline analysis, the prerequisite is that there is more than one 1-object-sensitive context element (i.e., FC4 > 1). Figure 5.5 shows the heuristic to choose between baseline and 1-object-sensitive analyses. 1-object-sensitive analysis is chosen over baseline analysis for a function `foo` if the `this` object is used at least once in the function body of `foo` (i.e., FC5 > 0). This result suggests that 1-object-sensitive analysis is useful in terms of Pts-Size client for a function `foo` whose behavior relies on the values of the `this` object, even for a small number of 1-object-sensitive context elements for `foo`. The balanced F-scores for baseline and 1-object-sensitive analyses in this heuristic are 0.65 and 0.79, respectively.

Baseline vs. 1st-parameter. Three function characteristics (i.e., FC6-FC8) are relevant to the precision of 1st-parameter-sensitive analysis for the Pts-Size client. For 1st-parameter-sensitive analysis to produce more precise results than baseline analysis, the prerequisite is

```

FC7-1ParName = 0 AND FC8-1ParOther = 0: baseline
FC7-1ParName > 0 OR FC8-1ParOther > 0: 1st-parameter

```

Figure 5.6: Heuristic: baseline vs. 1st-parameter

that there is more than one 1st-parameter-sensitive context element (i.e., $FC6 > 1$). Figure 5.6 shows the heuristic to choose between baseline and 1st-parameter-sensitive analyses. 1st-parameter-sensitive analysis is chosen over baseline analysis for a function `foo` if the first parameter of `foo` is used (i.e., as the property name in dynamic property accesses or otherwise) at least once in the function body of `foo` (i.e., $FC7 > 0$ or $FC8 > 0$). Similar to 1-object sensitivity, 1st-parameter sensitivity is another functional approach that distinguishes calls based on the computation states of a parameter. It is expected for 1-object-sensitive or 1st-parameter-sensitive analysis to be effective on the function `foo` if the values of the `this` object or the first parameter affect the behavior of `foo`. The balanced F-scores for baseline and 1st-parameter-sensitive analyses in this heuristic are 0.49 and 0.83, respectively.

```

FC4-RCNum / FC2-EquivCNum <= 0.8: 1-call-site
FC4-RCNum / FC2-EquivCNum > 0.8
| FC5-ThisUse / FC3-AllUse <= 0.375: 1-call-site
| FC5-ThisUse / FC3-AllUse > 0.375: 1-object

```

Figure 5.7: Heuristic: 1-call-site vs. 1-object

1-call-site vs. 1-object. To select between 1-call-site-sensitive and 1-object-sensitive analyses, function characteristics related to both are considered (i.e., $FC1$ - $FC5$). In our adaptive analysis, two context-sensitive analyses are compared for a function when both of them would be chosen over baseline analysis (see Section 5.2.4). As a consequence, the prerequisite for the heuristic in this case is the number of equivalence classes of call sites is larger than 1 (i.e., $FC2 > 1$) and the `this` object is used at least once (i.e., $FC5 > 0$). Figure 5.7 shows the heuristic to choose between 1-call-site-sensitive and 1-object-sensitive analyses. The heuristic consists of the relationship between the metrics of both analyses. 1-call-site-sensitive analysis is selected if it generates a greater number of context elements than 1-object-sensitive analysis (i.e., $FC4 / FC2 \leq 0.8$) for a function. This result suggests that 1-call-site-sensitive and 1-object-sensitive analyses in this case are empirically comparable in terms of precision. The relationship between the numbers of context elements generated by each analysis on `foo` indicates which context-sensitive analysis may be more precise for that function. When the number of receiver objects that invoke `foo` is close to or larger than the number of equivalence classes of call sites (i.e., $FC4 / FC2 > 0.8$), if the `this` object is used quite frequently (i.e., $FC5 / FC3 > 0.375$) in `foo`, 1-object-sensitive analysis is more precise for `foo`; otherwise (i.e., $FC5 / FC3 \leq 0.375$), 1-call-site-sensitive analysis is selected. This result is intuitive in that 1-object-sensitive analysis produces more precise results than 1-call-site-sensitive analysis for the `Pts-Size` client when (i) 1-object-sensitive

analysis generates a number of context elements and (ii) the behavior of a function is heavily dependent on the values of the receiver object. The balanced F-scores for 1-call-site-sensitive and 1-object-sensitive analyses in this heuristic are 0.67 and 0.8, respectively.

```

FC7-1ParName = 0
| FC8-1ParOther / FC3-AllUse <= 0.19: 1-call-site
| FC8-1ParOther / FC3-AllUse > 0.19
| | FC6-1ParNum / FC2-EquivCSNum <= 3.8
| | | FC8-1ParOther / FC3-AllUse <= 0.35: 1-call-site
| | | FC8-1ParOther / FC3-AllUse > 0.35: 1st-parameter
| | FC6-1ParNum / FC2-EquivCSNum > 3.8: 1st-parameter
FC7-1ParName > 0: 1st-parameter

```

Figure 5.8: Heuristic: 1-call-site vs. 1st-parameter

1-call-site vs. 1st-parameter. Function characteristics FC1-FC3 and FC6-FC8 are considered to select between 1-call-site-sensitive and 1st-parameter-sensitive analyses. The prerequisite for this comparison is $FC2 > 1$ and the first parameter of the function is used at least once (i.e., $FC7 > 0$ or $FC8 > 0$). Figure 5.8 shows the heuristic to choose between 1-call-site-sensitive and 1st-parameter-sensitive analyses. 1st-parameter-sensitive analysis is always selected if the first parameter is ever used as a property name in dynamic property accesses because the dynamic property accesses in JavaScript make analysis results very imprecise [72] and 1st-parameter sensitivity is a technique that significantly improves the analysis precision in this situation. In other cases, 1-call-site sensitive analysis is preferred if uses of the first parameter are not important to the function behavior (i.e., $FC8 / FC3 \leq 0.19$). Also, similar to the heuristic that selects between 1-call-site-sensitive and 1-object-sensitive analyses (Figure 5.7), the heuristic between 1-call-site-sensitive and 1st-parameter-sensitive analyses is dependent on the relationship between the context elements generated by both analyses. If 1st-parameter-sensitive analysis potentially generates many more context elements than 1-call-site sensitive analysis (i.e., $FC6 / FC2 > 3.8$), we expect the 1st-parameter-sensitive analysis to be more precise. Otherwise (i.e., $FC6 / FC2 \leq 3.8$), depending on the importance of the first parameter to the function behavior, 1-call-site-sensitive analysis (when $0.19 < FC8 / FC3 \leq 0.35$) or 1st-parameter-sensitive analysis (when $FC8 / FC3 > 0.35$) is selected. The balanced F-scores for 1-call-site-sensitive and 1st-parameter-sensitive analyses in this heuristic are 0.73 and 0.66, respectively.

1-object vs. 1st-parameter. Finally, Figure 5.9 presents the heuristic that selects between 1-object-sensitive and 1st-parameter-sensitive analyses. Function characteristics FC4-FC8 are considered and the prerequisite is $FC5 > 0$ as well as $FC7 > 0$ or $FC8 > 0$. It is not surprising that 1-object-sensitive analysis is selected by the heuristic when the `this` object is more frequently used (i.e., $FC5 / FC8 > 1.34$) and 1st-parameter-sensitive analysis is selected when the condition is opposite (i.e., $FC5 / FC8 \leq 0.8$). When uses of the `this` object and the first parameter are similar (i.e., $0.8 < FC5 / FC8 < 1.34$), the number

```

FC7-1ParName = 0
| FC5-ThisUse / FC8-1ParOther <= 0.8: 1st-parameter
| FC5-ThisUse / FC8-1ParOther > 0.8
| | FC5-ThisUse / FC8-1ParOther <= 1.34
| | | FC4-RCNum / FC6-1ParNum < 0.5: 1st-parameter
| | | FC4-RCNum / FC6-1ParNum >= 0.5
| | | | FC4-RCNum / FC6-1ParNum <= 1: unknown
| | | | FC4-RCNum / FC6-1ParNum > 1: 1-object
| | FC5-ThisUse / FC8-1ParOther > 1.34: 1-object
FC7-1ParName > 0: 1st-parameter

```

Figure 5.9: Heuristic: 1-object vs. 1st-parameter

of context elements generated by these two analyses decides the selection: (i) if 1-object-sensitive analysis potentially generates more context elements than 1st-parameter-sensitive analysis (i.e., $FC5 / FC8 > 1$), we expect 1-object sensitive analysis to be more precise; (ii) if 1st-parameter-sensitive analysis generates more than twice the number of context elements than 1-object-sensitive analysis (i.e., $FC5 / FC8 < 0.5$), 1st-parameter-sensitive analysis is selected; (iii) otherwise (i.e., $0.5 \leq FC5 / FC8 \leq 1$), it is not clear from the data in the benchmarks which analysis produces more precise results because the function characteristics indicate that they have similar capability to analyze the function. In this case, we randomly select between 1-object-sensitive and 1st-parameter-sensitive analyses for the function whose characteristics fall in this region. The balanced F-scores for 1-object-sensitive and 1st-parameter-sensitive analyses in this heuristic are 0.79 and 0.86, respectively.

```

iParName < jParName: jth-parameter
iParName = jParName
| iParOther < jParOther: jth-parameter
| iParOther = jParOther
| | iParNum < jParNum: jth-parameter
| | iParNum >= jParNum: ith-parameter
| iParOther > jParOther: ith-parameter
iParName > jParName: ith-parameter

```

Figure 5.10: Heuristic: ith-parameter vs. jth-parameter

ith-parameter sensitivity heuristics. We now have discussed the function characteristics used in the heuristics to select from baseline, 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses. Various other context-sensitive analyses exist for improving analysis precision. For example, ith-parameter-sensitive analysis provides variations to distinguish function calls based on the computation states of parameters, decided by the parameter i .

To enable the selection of *ith*-parameter sensitivity, we apply the heuristics of 1st-parameter-sensitive analysis to select between *ith*-parameter-sensitive analysis and baseline (Figure 5.6), 1-call-site-sensitive (Figure 5.8) or 1-object-sensitive (Figure 5.9) analysis. To select between *ith*-parameter-sensitive and *jth*-parameter-sensitive analyses, we apply the heuristic shown in Figure 5.10. We designed this heuristic based on the observation that for parameter sensitivity, a functional approach, the uses of the parameter whose computation states are used to distinguish function calls usually are more closely related to the analysis precision. In Figure 5.10, because the uses of a parameter as a property name in the dynamic property accesses is the most important characteristic, if one parameter is used as a property name more often than the other, distinguishing function calls based on its values may produce more precise results. If the *ParName* characteristics are the same for parameters *i* and *j*, the uses of the parameters in other situations are compared to decide if *ith*-parameter-sensitive analysis is more/less precise than *jth*-parameter-sensitive analysis. Finally, if both client-related metrics (i.e., *ParName* and *ParOther*) cannot distinguish the parameters, the heuristic selects the parameter that points to more objects.

Summary. These heuristics are intuitive for making a choice between each pair of analyses. More importantly, the heuristics for the call-strings approach and the functional approaches (i.e., Figures 5.7 and 5.8) allow us to make a decision between two incomparable analyses. Finally, the heuristics in Figures 5.4 to 5.9 are accurate (i.e., good balanced F-scores) in terms of their effectiveness on the benchmark programs.

5.2.4 Selection Workflow

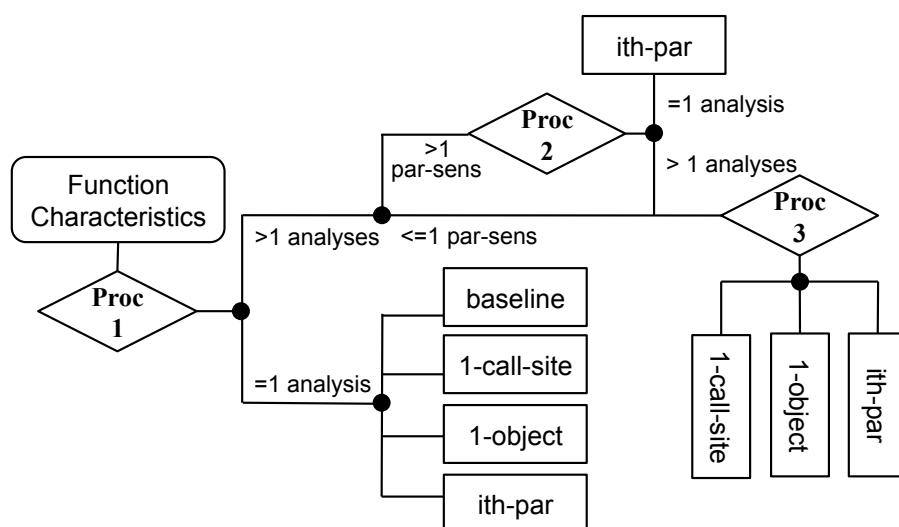


Figure 5.11: Workflow to select a context-sensitive analysis for a JavaScript function

The overall algorithm to select the context-sensitive analysis for a function is described in Figure 5.11. Given the function characteristics of function *foo*, Procedure 1 performs all

pairwise comparisons between baseline analysis and 1-call-site-sensitive, 1-object-sensitive and *ith*-parameter-sensitive analyses for all the parameters of `foo`. If Procedure 1 returns a single analysis, this analysis is selected for `foo`. Returning baseline analysis means none of the context-sensitive analyses makes much difference to improve precision for `foo`.

In case more than one choice is returned by Procedure 1, further comparisons are conducted to decide the specific context-sensitive analysis to use for `foo`. If the analysis precision of `foo` may benefit from applying parameter-sensitive analyses on multiple parameter choices returned by Procedure 1, Procedure 2 selects from among them to find the parameter *i* that may produce the most precise results when *ith*-parameter-sensitive analysis is applied. If the choices from Procedure 1 are now narrowed down to only the *ith*-parameter-sensitive analysis, this analysis is selected by our algorithm to analyze `foo`.

When necessary, Procedure 3 chooses from the remaining context-sensitive analyses that are returned by Procedures 1 and 2. If there are two remaining context-sensitive analyses to choose from, Procedure 3 applies the heuristic in Figure 5.7, 5.8 or 5.9 to decide on the context-sensitive analysis for analyzing `foo`. Otherwise (i.e., to choose from all three context-sensitive analyses), Procedure 3 compares each pair of 1-call-site-sensitive, 1-object-sensitive and *ith*-parameter-sensitive analyses and tries to find a best context-sensitive analysis for a majority of the pairs using heuristics in Figures 5.7, 5.8 and 5.9. For example, the adaptive analysis selects 1-call-site-sensitive analysis to analyze `foo` if it is chosen by both heuristics comparisons with 1-object-sensitive and *ith*-parameter-sensitive analyses. Finally, if Procedure 3 cannot decide on a specific accurate context-sensitive analysis (i.e., when each of the three heuristics returns a different analysis choice), the adaptive analysis randomly chooses an analysis for `foo`.

5.3 Evaluation

In this section, we first present the details of our experimental setup. We then evaluate our adaptive context-sensitive analysis using two sets of benchmarks. We compared the precision of adaptive analysis to other context-sensitive analyses applied to the entire program.

5.3.1 Experimental Setup

Our implementation of adaptive context-sensitive analysis was built on top *WALA*. The baseline points-to analysis, `ZERO_ONE_CFA` analysis in *WALA* that uses the default context sensitivity for JavaScript analysis, produced a call graph and a points-to solution from which we extracted the function characteristics. For the adaptive context-sensitive analysis, we implemented a new context selector⁷ that applies the context-sensitive analysis chosen by

⁷In *WALA*, the context element at a call site is decided by a context selector.

the heuristics for each function. Note that the default context-sensitive analysis is always used as well to ensure that the results of adaptive analysis are comparable to the baseline analysis.

The goals of the experiments included: (i) comparing the precision of adaptive context-sensitive analysis with each of the other context-sensitive analyses to learn if the adaptive analysis improves JavaScript analysis precision and (ii) studying the accuracy of selecting a specific context-sensitive analysis for each function to validate the quality of the heuristics presented in Section 5.2.3.

To achieve these goals, we evaluated our analysis on two sets of benchmarks: (i) the same benchmark programs on which we performed the empirical study in Section 5.1 (i.e., Benchmarks I including the 28 JavaScript programs collected by Kashyap *et al.* [33], divided into four categories) and (ii) four open-source JavaScript applications or libraries (i.e., Benchmarks II). The programs in Benchmarks II are (i) *Box2DWeb*, collected in the *Octane* benchmarks⁸, (ii) *minified.js* library⁹ version 1.0, (iii) *mootools* library¹⁰ version 1.5.1, and *benchmark.js* library¹¹ version 1.0.0. Because the heuristics were designed based on machine learning results using Benchmarks I, Benchmarks II serve to test if these heuristics can be applied by the adaptive context-sensitive analysis to arbitrary JavaScript programs to produce fairly accurate analysis results for the Pts-Size client.

5.3.2 Experimental Results

Results for Benchmarks I. Figure 5.12 shows the analysis precision results for Benchmarks I. We compared the results of our adaptive analysis with the context-sensitive analysis (i.e., 1-call-site-sensitive, 1-object-sensitive or 1st-parameter-sensitive analysis) that produced most accurate results for each program for these benchmarks. We define a context-sensitive analysis to be the *winner* analysis for a program if it was at least as precise as the other two context-sensitive analyses on the largest number of functions. For all 14 programs in the *addon* and *opensrc* benchmarks, 1-call-site-sensitive analysis was the winner among the 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses. For the *standard* benchmarks, 1st-parameter-sensitive analysis was winner on three programs and 1-call-site-sensitive analysis was winner on the other four programs. For all but one program in the *generated* benchmarks, 1-object-sensitive analysis was the winner analysis and 1-call-site-sensitive analysis was winner for *fourinarow*. In Figure 5.12, the winner analysis bar shows the percentage of the total number of functions in each benchmark category on which the winner analysis produced most accurate results. For example, the leftmost winner analysis bar represents that 1-call-site analysis (i.e., the winner analysis for all the

⁸<https://developers.google.com/octane/>

⁹<http://minifiedjs.com>

¹⁰<http://mootools.net>

¹¹<http://benchmarkjs.com>

programs in the *addon* benchmarks) produced at least as precise results as 1-object-sensitive and 1st-parameter-sensitive analyses for 98.8% of the functions in the *addon* benchmarks. The adaptive analysis bar shows the percentage of functions in each benchmark category for which our adaptive analysis produced at least as precise results as 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses.

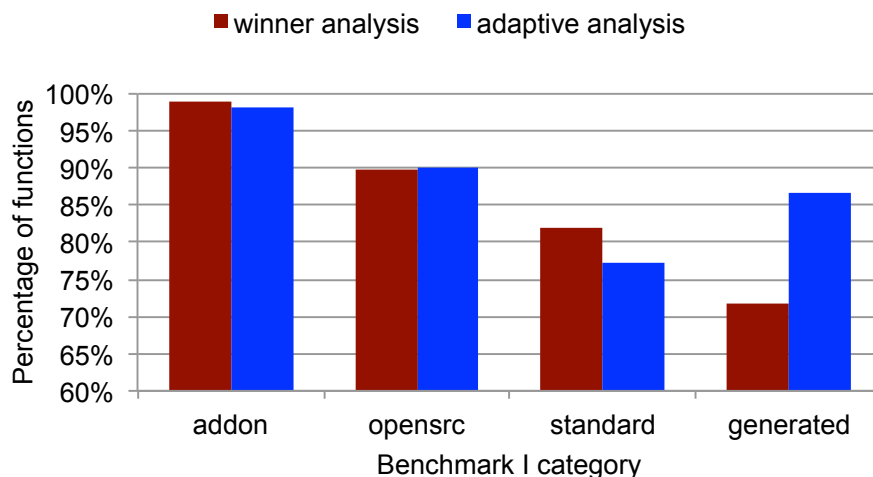


Figure 5.12: Analysis precision on Benchmarks I

The most important aspect of adaptive analysis is its ability to select an appropriate context-sensitive analysis for a specific function. Table 5.2 shows the accuracy of the analysis selection process for a function using the heuristics presented in Section 5.2.3 with the Pts-Size client. The first column in Table 5.2 lists the (set of) analyses that are best or equally precise (i.e., rows 4-7 in the first column) for a function (see Section 5.1). The second column shows the total number of functions in all programs from Benchmarks I on which the corresponding analyses were observed to produced the best or equally precise results. There were in total 1817 functions analyzed in Benchmarks I and the precision results of 977 functions were improved over baseline analysis by at least one context-sensitive analysis for the Pts-Size client. The last column presents the the number of functions on which the adaptive analysis matched the observed results (i.e., true positives for our heuristics). For those functions on which 1-call-site-sensitive and 1-object-sensitive analyses produced the best results, the selection heuristics resulted in good precision (i.e., 73.5% and 68%, respectively). However, the selection on 1st-parameter-sensitive analysis only achieved 48.8% precision. This is because our adaptive analysis chooses the appropriate *ith*-parameter-sensitive analysis to analyze a function using the parameter sensitivity. Here we are only checking the selection precision with respect to 1st-parameter-sensitive analysis; whereas *ith*-parameter-sensitive analysis ($i > 1$) was applied to analyze 51 functions in the programs of Benchmarks I.

1-call-site-sensitive and 1-object-sensitive analyses produced equally precise results in terms on Pts-Size client on 153 functions. The adaptive analysis correctly selected 1-call-site-

best / equally precise analysis	# of observed functions	# of selected functions (true positives)	true positive rate
1-call-site	351	258	73.5%
1-object	241	164	68.0%
1st-parameter	162	79	48.8%
1-call-site = 1-object	153	1-call-site: 39	94.1%
		1-object: 105	
1-call-site = 1st-parameter	39	1-call-site: 23	74.4%
		1st-parameter: 6	
1-object = 1st-parameter	6	1-object: 4	83.3%
		1st-parameter: 1	
1-call-site = 1-object = 1st-parameter	25	1-call-site: 2	100%
		1-object: 13	
		1st-parameter: 10	
total	977	704	72.1%

Table 5.2: Selection precision for Benchmarks I

sensitive or 1-object-sensitive analysis to analyze 144 of those 153 functions, and interestingly, the choice was leaning towards 1-object-sensitive analysis (i.e., 1-object-sensitive analysis for 105 functions comparing to 1-call-site-sensitive analysis for 39 functions). For the functions on which equally precise results were produced by 1-call-site-sensitive and 1st-parameter-sensitive analyses, adaptive analysis selects more functions to be analyzed by 1-call-site-sensitive analysis. The overall precision of selecting a context-sensitive analysis by our heuristics is very good (i.e., 72.1%); this is a measure of when adaptive analysis made the best choice possible. The above observations may help us to improve the heuristics in the future.

The time cost of our adaptive analysis is the sum of its two stages (i.e., the baseline points-to analysis to gather function characteristics and the subsequent adaptive context-sensitive analysis). We compare the performance of our adaptive analysis with the winner analysis for each program in Benchmarks I. On average over all the programs in Benchmarks I, our two-staged analysis introduced a 67% overhead. Nevertheless, the second stage (i.e., the adaptive context-sensitive analysis) is on average 19% faster than the winner analysis over the Benchmarks I programs. This result suggests that an appropriate choice of context sensitivity per function yields better performance and precision.

Results for Benchmarks II. Figure 5.13 shows initial analysis precision results using four programs from Benchmarks II. The sizes of these programs, in terms of the number

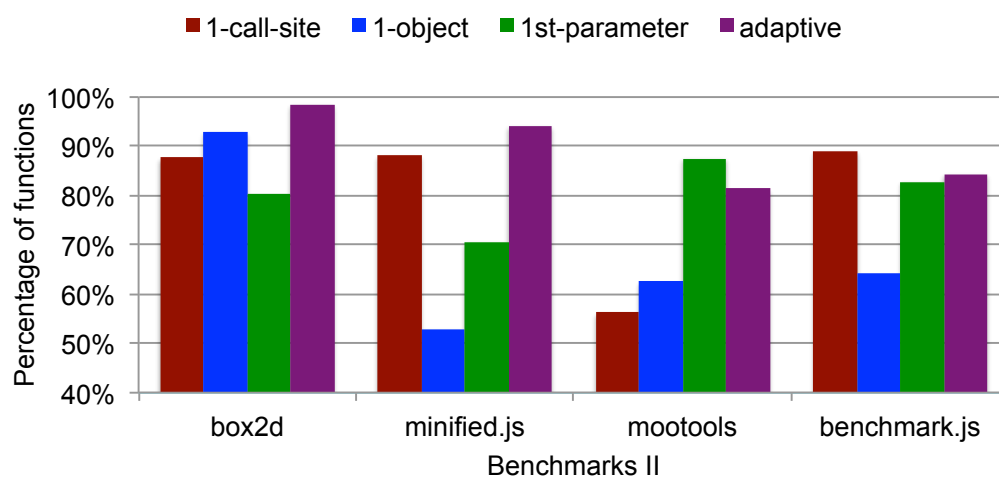


Figure 5.13: Analysis precision on Benchmarks II

of functions analyzed, are 126, 119, 80 and 64, respectively. The 1-call-site, 1-object and 1st-parameter bars represent the percentage of functions on which each context-sensitive analysis produced at least as precise results as the other two. The adaptive bar (rightmost) represents the percentage of functions on which the adaptive analysis produced at least as precise results as 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses. We picked these four programs in Benchmarks II because their analysis results for different context-sensitive analyses were varied. For example, 1-object-sensitive analysis was more precise than 1-call-site-sensitive and 1st-parameter sensitive analyses for *box2d*, while 1st-parameter-sensitive analysis was the most precise for *mootools*.

The results in Figure 5.13 show that our adaptive analysis achieved better results than any single context-sensitive analysis for *box-2d* and *minified.js*. For example, 1-object-sensitive analysis was at least as precise for 92.8% of the functions in *box-2d*; adaptive analysis improved these results to 98.4% of the functions. 1-call-site-sensitive analysis produced at least precise results for 88.2% of the functions in *minified.js*; adaptive analysis improved the results by 5.9% more functions. 1st-parameter-sensitive and 1-call-site-sensitive analyses were the most precise context-sensitive analyses for *mootools* and *benchmark.js*, respectively. While adaptive analysis produced results lower than these analyses, the results of adaptive analysis were close, only different for 6.2% and 4.7% of the functions in *mootools* and *benchmark.js*, respectively. Overall, our adaptive context-sensitive analysis was fairly accurate for analyzing these programs from Benchmarks II. This promising result indicates that the heuristics presented in Section 5.2.3 may be applicable in general to JavaScript programs.

5.3.3 Discussion

We have demonstrated the ability of our adaptive analysis that chooses a specific context-sensitive analysis for each function in order to significantly improve analysis precision. Nevertheless, this initial work has inspired us with more research ideas for further improvements of context-sensitive analyses for JavaScript. First, since we evaluated the adaptive analysis on a simple client of points-to analysis (i.e., Pts-Size), it would be interesting to know if adaptive analysis is effective to improve precision for other clients (e.g., security analysis). Second, context-sensitive analysis for JavaScript are not limited to 1-call-site, 1-object and ith-parameter. A deeper object-sensitive analysis (i.e., k-object) or another context-sensitive analysis (e.g., using the length of the parameter list at a call site as context element to distinguish JavaScript variadic functions as presented in Chapter 3) could be used by adaptive analysis. New heuristics need to be designed for selecting these analyses. Third, we would like to explore if analysis precision may benefit from applying multiple-sensitive analyses on a specific JavaScript function. The idea of hybrid context-sensitive analysis has been tried for analyzing Java programs [34]. Fourth, scalability is an important issue for JavaScript analyses, especially for analyzing JavaScript websites that use libraries heavily (e.g., jQuery). In this work, we do not address this problem, that is, when a baseline points-to analysis is not scalable for a large JavaScript application, typically a website.¹² In the future, we plan to focus on improving the performance of analysis of JavaScript websites using an adaptive approach.

Threats to validity. Although we used benchmarks collected by Kashyap *et al.* [33] as well as other JavaScript programs to evaluate adaptive context-sensitive analysis, these programs may not be representative of non-website JavaScript applications, which might threaten the validity of our conclusions as applicable to all JavaScript programs.

¹²We therefore have used different benchmarks for evaluating adaptive context-sensitive analysis in this chapter from those in Chapters 3 and 4.

Chapter 6

Related Work

In this chapter, we discuss the work related to our JavaScript dataflow analyses. We classify the related work into two categories: (i) related analyses of dynamic languages, and (ii) context-sensitive analysis.

6.1 Related Analyses of Dynamic Languages

Various analyses have been presented to study and/or handle the dynamic features of JavaScript. These empirical studies serve as the motivation for designing new JavaScript analyses, including ours. Most JavaScript analyses are static and some of them use a combination of dynamic and static analyses. We then discuss security analyses for JavaScript related to our blended taint analysis. Finally, we will present empirical studies and analyses on other dynamic programming languages.

6.1.1 Empirical Studies of JavaScript Applications

Richards *et al.* performed experiments on real-world JavaScript websites to study several aspects of dynamic behavior [59]. Popular websites were studied resulting in several conclusive observations: (i) properties are not just added at object initialization, (ii) variadic functions are not rare, (iii) the prototype hierarchy often changes within libraries, and (iv) property deletions are common in some websites. These observations suggested that common assumptions about the behavior of JavaScript applications were not valid and thus, motivated us to design new analyses that accommodate the dynamic behavior of JavaScript. We also have reused the dynamic traces and modified the analysis infrastructure in Richards *et al.* to perform the in-depth study of JavaScript object behavior (Chapter 4).

Ratanaworabhan *et al.* presented a study on comparing the behavior of JavaScript bench-

marks (e.g., *SundSpider* and *V8*) with real web applications [56]. The authors evaluated differences in behavior between the benchmarks and websites, including program size, complexity and behavior. Their results suggested that these benchmarks were not representative of real JavaScript usage. This study motivated us to evaluate *JSBAF* on JavaScript code extracted from websites.

Richards *et al.* investigated the uses of `eval` in real JavaScript applications [58]. This study categorized common patterns in the use of `eval`, revealing a majority of `eval` uses are not necessary, while complex pieces of code can also be generated. The results motivated us to design *JSBAF* with handling `eval` and other dynamic code generation mechanisms as one of its most important features.

Other studies have been focusing on specialized characteristics of JavaScript applications (e.g., security [81, 82] and errors [51, 50]). The results of these studies suggested the correlation between dynamic features and the errors in JavaScript applications. For example, Yue and Wang reported insecure JavaScript practices related to dynamic code generation [81]. These studies demonstrated that JavaScript program analysis needs to handle dynamic features to be useful in real clients.

6.1.2 JavaScript Analyses of Dynamic Features

We use *WALA* as the implementation platform of our static analysis algorithms for JavaScript. There have been multiple other works on JavaScript analysis built on top of *WALA*, sharing the same static analysis infrastructure as ours. Sridharan *et al.* presented a points-to analysis for JavaScript that focused on handling correlated dynamic property accesses [72]. Correlated property accesses (i.e., dynamic property reads and writes that use the same property name) were identified and then extracted into a function. Using the property name as the calling context, points-to analysis tracking correlation was shown to be more precise and efficient than a field-sensitive Andersen’s points-to analysis. We augmented the correlation tracking points-to analysis with state sensitivity. Our experiments demonstrated a significant improvement in the analysis precision.

Feldthaus *et al.* presented a scalable static analysis to construct unsound but accurate call graphs for JavaScript applications [15]. The field-based flow analysis only tracks function objects and ignores dynamic property reads and writes. Two call graph construction algorithms were designed: (i) pessimistic approach that only tracks calls of the form `(function(\bar{x}){...})(\bar{e})` where an anonymous function is directly applied to some arguments (i.e., one-shot calls), and (ii) optimistic approach that performs fixpoint iteration. The comparison between static and dynamic call graphs showed that the field-based analysis, while in principle unsound, produced an accurate call graph in practice. The authors have also demonstrated the generated call graphs may be useful for JavaScript IDE services. *JSBAF* uses dynamic analysis to generate call graphs for JavaScript applications and also collects the dynamically generated code as well as function variadicity information. Call

graphs generated by both approaches are unsound. Our state-sensitive points-to analysis may use either a dynamic call graph or a static call graph built by the field-sensitive flow analysis as the input.

Schäfer *et al.* proposed a dynamic analysis to identify determinate (i.e., always having the same value at a given program point) variables and expressions in JavaScript programs [62]. The approach soundly infers the determinacy facts that hold for any execution. It tracks the indeterminacy using techniques similar to dynamic information flow analysis. To ensure the soundness of the results, *counterfactual execution* is performed for unexecuted conditional branches. The authors performed case studies and suggested the results might be helpful to improve static points-to analysis. Determinacy information for JavaScript variables may further improve our points-to analysis algorithms.

Alimadadi *et al.* presented a hybrid change impact analysis for JavaScript [3]. Their analysis focuses on the interplay between the JavaScript code and the Document Object Model. The tool, *TOCHAL*, builds a hybrid system dependence graph, by combining static and dynamic call graphs. It uses *WALA* to construct static call graphs. *TOCHAL*'s models of handling DOM interactions and asynchronous JavaScript mechanisms may further improve the accuracy of our analyses.

TAJS is another analysis framework for JavaScript, including various static analysis algorithms. Jensen *et al.* presented a flow-sensitive static analysis that can precisely model prototype chains [30, 31]. In their analysis, the *absent* set indicated potentially missing properties. The property edges annotated with * play a similar role in our state-sensitive analysis. The authors also used recency abstraction to perform strong updates of property writes. Jensen's analysis is context-sensitive similar to 1-object sensitivity used in Java. Lazy propagation was introduced to improve the performance of flow- and context-sensitive analysis for JavaScript [31]. Jensen *et al.* extended *TAJS* providing abstract object models for HTML objects and event handlers in JavaScript web applications [29]. Dynamically generated code was recognized statically. Because some property names were unknown due to abstraction, this analysis sacrificed soundness by skipping the modeling for the particular property writes. Applications with libraries were excluded in the experiments.

Jensen *et al.* presented a static analysis to automatically transform common uses of `eval` into other language constructs [28]. The framework, *Unevalizer*, incorporates the `eval` transformation in the whole-program dataflow analysis. The analysis was designed to eliminate calls to `eval` with constant arguments and several cases with non-constant arguments. In their experiments, *Unevalizer* successfully eliminated many nontrivial `eval` calls in program slices and medium size web applications. *JSBAF* uses dynamic analysis to collect the code generated from `eval` calls. Both approaches expand the applicability of static analysis for JavaScript. *Unevalizer*, a sound technique, cannot always successfully eliminate the `eval` calls in complicated cases; *JSBAF*, on the other hand, collects all executed `eval` calls.

Following Schäfer *et al.*'s work on dynamic determinacy analysis [62], Esben *et al.* presented a static analysis that infers determinacy information and optimizes the analysis in *TAJS*

with multiple techniques [4]. Specifically, the analysis uses parameter sensitivity, loop specialization, context-sensitive heap abstraction and strengthened models of standard library functions. In their experiments, the combination of these techniques resulted in significant improvement of analysis precision and performance, scalable on analyzing small programs that use jQuery. Our adaptive analysis uses parameter sensitivity as one of the context-sensitive analysis choices.

Park and Ryu presented another scalable static analysis of JavaScript via loop sensitivity [53]. The authors identified the scalability problem with JavaScript analysis as the combination of imprecise analysis results in loops and the dynamic nature of object property names. Their analysis improves precision in loops by distinguishing each iteration of a loop as much as needed during analysis with different loop contexts for each iteration depending on the analysis results of loop conditional expressions, similar to loop unrolling. Park’s analysis was implemented as an extension of *SAFE* analysis framework [36]. Their experimental results showed that the loop-sensitive analysis was scalable in analyzing some real JavaScript webpages. The introduction of loop sensitivity in this work may expand the choices of context sensitivity in our adaptive context-sensitive analysis in the future.

Kashyap *et al.* presented *JSAI*, an abstract interpreter for JavaScript [33]. *JSAI* was designed to be configurable for analysis sensitivity (i.e., path, context, and heap sensitivity). The authors evaluated the analysis precision and performance on JavaScript benchmarks with different configurations and made the observation that there was no clear winner across all benchmarks, in terms of JavaScript context sensitivity. This result motivated us to perform an in-depth study on JavaScript context sensitivity and design an adaptive context-sensitive analysis. We also reused the benchmarks by Kashyap *et al.* [33] for the study and evaluation of our adaptive analysis.

Madsen *et al.* presented a static analysis of JavaScript focusing on frameworks and libraries [43]. The authors designed a *use analysis* combined with a points-to analysis to recover information about the structure of objects and to infer the missing inter-procedural flow introduced by the unavailable native code. The analysis assumes (i) the object properties are not dynamically added or removed after the object has been fully initialized, (ii) the presence of a property does not rely on program control flow, and (iii) property names should not be computed dynamically, etc. Our analyses hold assumptions of JavaScript applications based on observed dynamic behavior [59]. Our work is complementary to this technique, in that more precise points-to results would make it more practical.

Several type-based approaches were proposed for JavaScript that support dynamic features such as prototype-based inheritance (e.g., [9, 37]). It is difficult to compare our analyses with them as to practicality, because no empirical evidence on large JavaScript programs was presented.

Jalangi is a dynamic analysis framework for JavaScript [63]. It has been used for various dynamic analyses (e.g., [18, 54]). We may use *Jalangi* as the dynamic analysis component of *JSBAF* in the future.

6.1.3 JavaScript Security Analyses

Tripp *et al.* presented *JSA*, a hybrid security analysis for JavaScript web applications [74]. This hybrid approach uses a web crawler to obtain concrete DOM values which enable a string analysis as a refinement of the taint analysis. Their algorithm takes a static call graph as input. *JSBAF* uses a dynamic call graph that is unsound but more accurate than a static call graph. Blended analysis also collects other dynamic information (e.g., variadic functions) to specialize the static phase. It is future work to investigate the accuracy of the static call graph used by *JSA*.

Guarnieri *et al.* presented *ACTARUS*, a static taint analysis for JavaScript built on top of *WALA* [21]. Language constructs, including object creations, reflective property accesses, and property lookups were modeled, but reflective calls like `eval` were not modeled. Our state-sensitive analysis improved the models of prototype-based inheritance and property lookups via flow and context sensitivity.

Guarnieri and Livshits presented another static points-to analysis to detect security and reliability issues and experimented with JavaScript widgets [19]. JavaScript_{SAFE} is a subset of JavaScript that static analysis can soundly approximate, even with reflective calls such as `call` and `apply`. A conservative model of prototyping was used with a flow- and context-insensitive analysis. Other dynamic constructs such as `eval` were not handled. This JavaScript static analysis cannot model all of the language's dynamic features, whereas *JSBAF* handles the more common dynamic features used by real websites.

Chugh *et al.* presented an information flow analysis for JavaScript [10]. The staged approach analyzes the statically visible code first and then incrementally analyzes the dynamically generated code. A similar approach was proposed by Guarnieri and Livshits [20]. *JSBAF* differs from these approaches in two ways: (i) blended analysis collects dynamically generated code during profiling rather than doing this incrementally, and (ii) blended analysis also facilitates potentially more precise modeling of other dynamic features whose semantics depend on run-time information.

Other security analyses were presented to detect cross-site scripting [49], cross-origin capability leaks [7], and code injections [61] of JavaScript applications. We present *JSBAF* as a general-purpose analysis framework that can be instantiated with different security analysis clients.

6.1.4 Studies and Analyses of Other Dynamic Languages

Other programming languages such as PHP and Ruby share similar dynamic features with JavaScript (e.g., run-time code generation). Findings and analyses of these dynamic programming languages may be applicable to JavaScript. We discuss some work that is mostly related to ours.

Furr *et al.* presented a profile-guided static typing via program transformation for Ruby programs [17]. The run-time instrumentation gathered profiles of dynamic feature usage and these features were then replaced with statically analyzable alternatives. The idea of specializing difficult to analyze program constructs via dynamic information was also explored by our blended analysis and Tripp *et al.* [74] for JavaScript analysis. The usage reported by Furr *et al.* [17] shows that dynamic features are pervasive throughout the Ruby benchmark suite, especially the `eval` construct; similar findings on `eval` usage were also reported for JavaScript websites [58].

Honker and Harland conducted experiments on Python programs studying their dynamic behavior [27]. The authors observed frequent uses of reflective features and investigated dynamic object modifications. Our state-sensitive analysis is designed for objects whose behavior changes at runtime. If objects of other programming languages exhibit similar behavior as JavaScript objects [80], state sensitivity may also be applicable to analyze these programs.

PHP is a popular dynamic programming language for designing server-side web applications. Various techniques have been developed to study and analyze PHP applications. Hills *et al.* presented an empirical study on PHP feature usage [25]. The authors proposed several dynamic metrics similar to those in Richards *et al.* [59] and provided guidance for developing program analysis tools for PHP.

PHANTM, a hybrid static and dynamic analyzer for PHP, was presented by Kneuss *et al.* [35]. This hybrid approach records the configuration data of the program and performs static analysis on concretized program state; thus many configuration variables become constant. *JSBAF* is a tightly coupled hybrid approach in that static analysis focuses on the dynamic calling structure, handling several dynamic features of JavaScript.

Hauler and Kofroň presented a static analysis framework of PHP applications [23]. It consists of two phases: (i) the first phase resolves the dynamic constructs in the program via heap, value and declaration analyses, and (ii) the second phase proceeds in a way similar to a one for a language without dynamic features. The authors implemented a taint analysis of PHP applications based on their analysis framework and compared it to other PHP taint analyses (i.e., *PIXY* [32] and *PHANTM* [35]). Their results showed the proposed PHP analysis was more accurate than existing tools. This analysis handles several PHP features (e.g., dynamic accesses to associative arrays); similar features were handled by our work and other JavaScript analyses (e.g., [72]).

6.2 Context-sensitive Analysis

We have covered several whole-program context-sensitive analysis techniques (e.g., call-site and parameter sensitivity) in Chapter 2. In this section, we first discuss the context-sensitive analysis mostly related to our state-sensitive analysis (i.e., object sensitivity) and then

present the work related to our adaptive context-sensitive analysis.

6.2.1 Object-sensitive Analysis

State-sensitive analysis is inspired by object sensitivity. Milanova *et al.* first introduced object sensitivity and implemented an object-sensitive points-to analysis for Java using a receiver object represented by its creation site as a context element [47]. The experiments performed by Lhoták and Hendren showed object sensitivity is the better choice as a calling context when analyzing Java programs [38]. Changes to object properties in JavaScript render object creation sites insufficient to represent object behavior, whereas state sensitivity captures object behavior changes better.

Smaragdakis *et al.* formalized object sensitivity, summarizing its variations [68]. They introduced type sensitivity where object type was used as the context element. For dynamically-typed languages like Javascript, type is a run-time notion, encapsulated in the idea of *obj-ref state* used as a context element.

6.2.2 Selective Context-sensitive Analysis

To the best of our knowledge, adaptive context-sensitive analysis is the first analysis for JavaScript that selectively uses multiple context-sensitive analyses to analyze a program when context sensitivity may help improve precision. Our work is related to approaches that apply context-sensitive analysis selectively for other programming languages.

Context-sensitive analysis has been deeply investigated for other object-oriented languages such as Java. However, these object-oriented languages do not seem as amenable to our approach of using different context-sensitive analyses on different functions. Castries and Smaragdakis presented hybrid context-sensitive points-to analysis for Java [34]. Several combinations of call-site and object-sensitive analyses were explored and evaluated for precision. Their results showed that selectively adding call-site-sensitive analysis to specific places in the program (e.g., static calls) significantly improved the precision of object-sensitive points-to analysis for Java. Our adaptive analysis automatically chooses an appropriate context-sensitive analysis for each function in JavaScript program.

Several works were proposed to tune the context sensitivity of an analysis based on pre-analysis results. Smaragdakis *et al.* presented introspective analysis that aims to improve the performance of a context-sensitive analysis for Java [69]. Introspective analysis selectively refines allocation sites or call sites based on the heuristics consisting of metrics computed from context-insensitive points-to results. The heuristics are tunable via constant parameters. In our adaptive analysis, we designed the heuristics based on the results learnt from JavaScript benchmarks, and function characteristics extracted from baseline analysis and syntactic analysis. The heuristics in our analysis focus on “which” context-sensitive analysis

may improve precision instead of “if” context sensitivity would be of benefit.

Sridharan and Bodík presented a refinement-based points-to analysis for Java [70] that refines sensitivity for heap accesses and method calls. It also is demand-driven in that it skips irrelevant code in the analysis. Our adaptive context-sensitive analysis aims to improve precision for the whole program.

Guyer and Lin presented a client-driven analysis for C that automatically adjusts its precision in response to the needs of client analyses [22]. This client-driven analysis monitors polluting assignments (i.e., the program points that result inaccuracy in the analysis) and tunes context as well as flow sensitivity to improve precision. Liang and Naik presented another client-driven algorithm for Java that prunes away analysis results irrelevant to refinement for more precision [40]. For these techniques, a pre-analysis is used to determine the program points for refinement. Baseline points-to results of adaptive analysis is used to derive the function characteristics for our heuristics. Furthermore, our adaptive analysis involves more than one context-sensitive analysis.

Oh et al. presented a selective context-sensitive analysis for C guided by an impact pre-analysis [52]. The impact pre-analysis applies full context sensitivity (i.e., ∞ -CFA) but with simplified abstract domain and transfer functions to infer the impacts of context sensitivity in the main analysis. The heuristics in our adaptive analysis focus on the characteristics of a function to indicate whether analysis precision for a function would benefit from a specific context-sensitive analysis. Our pre-analysis, the baseline analysis, is comparable with the adaptive analysis in terms of abstract domain and transfer functions.

Chapter 7

Conclusions and Future Work

JavaScript is the *lingua franca* of client-side of web applications and it is also becoming one of the most popular programming languages. Nevertheless, it lacks effective software tools that can automatically analyze JavaScript applications. The dynamic language features of JavaScript pose great challenges for designing practical dataflow analysis techniques. Existing static analyses often are ineffective analyzing JavaScript applications that exhibit dynamic features such as run-time code generation and prototype-based inheritance. We aim to design new program analysis techniques that handle the dynamic features of JavaScript and thus produce practical analysis results. To achieve this goal, we have designed a general-purpose blended analysis framework for JavaScript (i.e., JSBAF) as well as two novel context-sensitive analysis algorithms.

7.1 JavaScript Blended Analysis Framework

JSBAF is a general-purpose analysis framework that tightly couples dynamic and static analyses to address analysis challenges raised by several JavaScript features. Specifically, the dynamic phase of JSBAF collects the run-time information of a JavaScript application including function calls and dynamically generated code and the static phase of JSBAF takes advantage of this dynamic information by focusing the static dataflow analysis on the dynamic calling structures. Therefore, based on the dynamic information, specific optimizations such as pruning and context sensitivity can be applied to handle JavaScript dynamic features such as function variability and constructor polymorphism. The components of JSBAF are flexibly replaceable depending on the analysis resource and clients. We also have discussed the soundness of the analysis results produced by JSBAF.

We instantiated JSBAF to perform blended taint analysis for JavaScript and experimented with our implementation on popular JavaScript websites. Comparisons to two pure static taint analyses showed that blended taint analysis had significantly better performance and

accuracy than the static techniques, attesting the practicality of JSBAF.

7.2 State-sensitive Points-to Analysis

We have performed an in-depth study of the behavior of JavaScript objects in web applications, finding that the properties associated with a JavaScript object often changes at different program points during execution. This characteristic of JavaScript objects makes it difficult to design an effective analysis for JavaScript. Based on the observations, we introduced state-sensitive points-to analysis that models object behavior changes accurately by using a hierarchical program representation emphasizing state-update statements, by defining state sensitivity, a better context sensitivity mechanism for a dynamic language, and by enhancing the points-to graph representation for improved object property lookups. We implemented state-sensitive points-to analysis as the static phase of JSBAF. Experimental results on the REF client showed our analysis significantly improved the precision of a good JavaScript points-to analysis [72].

7.3 Adaptive Context-sensitive Analysis

The effectiveness of context-sensitive analysis on a JavaScript program depends on its coding style because JavaScript features object-oriented, functional as well as procedural programming paradigms. The fact that there was no winner context-sensitive analysis for the JavaScript benchmarks we examined motivated us to design an adaptive analysis . Our adaptive points-to analysis applies a specialized context-sensitive analysis per function, using heuristics based on function characteristics derived from an inexpensive points-to analysis. Our experimental results show that adaptive analysis is more precise than any single context-sensitive analysis for several programs in the benchmarks, especially for those multi-paradigm programs whose analysis precision can benefit from multiple context-sensitive techniques. Our adaptive analysis is the first for JavaScript that selectively chooses multiple context-sensitive techniques during the analysis of a program.

7.4 Future Work

There are several possible directions of future work to further explore adaptive analysis. The results presented in Chapter 5 suggest that the adaptive analysis is a promising technique towards increasing the precision of JavaScript analysis. As discussed in Section 5.3.3, we have been inspired with more ideas for further improvements of context-sensitive analyses for JavaScript. We are interested in exploring the effects of applying multiple context-sensitive analyses per function and also investigating the impact of applying deeper context

sensitivity. Multi-sensitive analyses often are prohibitively expensive to be practical when performing whole-program analysis; thus, the benefits of multi-sensitive analyses in terms of precision have not been explored. Our adaptive analysis allows uses of multi-sensitive analysis on a small portion of functions whose precision may significantly benefit from multiple context sensitivity; thus, the adaptive analysis that applies multiple context sensitivity per function may result in more precise results without significant performance overhead. Similarly, adaptive analysis may achieve the balance between precision and performance when applying deeper context sensitivity. In addition, a deeper context-sensitive analysis in the context of adaptive analysis may expose new concept of context sensitivity. Specifically, the parametrized models of context sensitivity (i.e., k -object sensitivity) defined and experimented in the literature are for whole-program single-context-sensitive analysis. To explore deeper context sensitivity in the context of adaptive analysis, different levels of context sensitivity may use different context-sensitive techniques. For example, a 2-context-sensitive analysis for function `foo` may use call-site sensitivity in the first level and object sensitivity in the second level of context sensitivity. This notion of mixed context sensitivity in a k -context-sensitive analysis was not explored before and may become a new direction for research in the area of context-sensitive analysis.

Another possible direction of future work is to perform a more thorough investigation of state-sensitive analysis. We presented k -state sensitivity, a parameterized model of context sensitivity. We have evaluated the effectiveness of 1-state-sensitive analysis on the REF client. It remains unknown if deeper state sensitivity may further improve analysis precision and if the extra context sensitivity will result in a significant performance overhead. It also is interesting to know if state sensitivity works well for other analysis clients.

Another possible direction of future work is to explore an alternate design of JSBAF and/or a new paradigm to combine dynamic and static analyses. The current design of JSBAF focuses static analysis on dynamic call structures and each trace is analyzed separately; thus, multiple static analyses may be performed for the same program and the analysis solution is an over-approximation of the behavior of observed executions. An alternate design of JSBAF (e.g., Figure 3.2) may exhibit different precision and performance results; this is worthwhile to evaluate. More significantly, for some analysis clients, there is a need for soundness over most program behavior; therefore, a new combination of dynamic and static analyses may be designed to accommodate the requirements of such a client. For example, a static analysis might be performed to obtain the over-approximation of program behavior, while dynamic information might be used on-the-fly to replace the part of static analysis solution that is too imprecise.

Despite of the fact that new static analysis algorithms are being developed towards practical analysis of JavaScript, the state-of-the-art static analysis of JavaScript still suffers from scalability issues on real JavaScript applications, especially JavaScript libraries. As a more general direction of future work, being aware of the difficulty of understanding why an analysis technique is ineffective on a specific program, we aim to design an automated approach to diagnose the root causes of analysis impracticality (i.e., imprecision and unscalability).

For example, the uses of complicated JavaScript libraries such as jQuery pose challenges for precise and scalable analysis of JavaScript web applications. It is crucial to identify the causes of imprecision of static analysis when analyzing these libraries. It often takes a huge amount manual effort to locate these root causes and it is difficult to be complete; therefore, an automated tool may help to exhaustively locate the causes of imprecision and push forward the state-of-the-art of analyses. Also a user of such a tool can decide on the appropriate analysis for certain requirements.

Bibliography

- [1] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming, ECOOP '95*, pages 2–26, London, UK, UK, 1995. Springer-Verlag.
- [2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
- [3] S. Alimadadi, A. Mesbah, and K. Pattabiraman. Hybrid dom-sensitive change impact analysis for javascript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 321–345, 2015.
- [4] E. Andreasen and A. Møller. Determinacy in static analysis for jQuery. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '14*, pages 17–31, New York, NY, USA, 2014. ACM.
- [5] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, pages 571–580, New York, NY, USA, 2011. ACM.
- [6] G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. In *Proceedings of the 13th International Conference on Static Analysis, SAS'06*, pages 221–239, Berlin, Heidelberg, 2006. Springer-Verlag.
- [7] A. Barth, J. Weinberger, and D. Song. Cross-origin JavaScript capability leaks: Detection, exploitation, and defense. In *Proceedings of the 18th Conference on USENIX Security Symposium, SSYM'09*, pages 187–198, Berkeley, CA, USA, 2009. USENIX Association.
- [8] B. Blanchet. Escape analysis for object-oriented languages: Application to java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming*,

- Systems, Languages, and Applications*, OOPSLA '99, pages 20–34, New York, NY, USA, 1999. ACM.
- [9] R. Chugh, D. Herman, and R. Jhala. Dependent types for JavaScript. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 587–606, New York, NY, USA, 2012. ACM.
- [10] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 50–62, New York, NY, USA, 2009. ACM.
- [11] A. De and D. D'Souza. Scalable flow-sensitive pointer analysis for java with strong updates. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 665–687, Berlin, Heidelberg, 2012. Springer-Verlag.
- [12] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.
- [13] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 118–128, New York, NY, USA, 2007. ACM.
- [14] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 59–70, New York, NY, USA, 2008. ACM.
- [15] 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 2013 International Conference on Software Engineering*, ICSE '13, pages 752–761, Piscataway, NJ, USA, 2013. IEEE Press.
- [16] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc., 2006.
- [17] M. Furr, J.-h. D. An, and J. S. Foster. Profile-guided static typing for dynamic scripting languages. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 283–300, New York, NY, USA, 2009. ACM.
- [18] L. Gong, M. Pradel, M. Sridharan, and K. Sen. Dlint: Dynamically checking bad coding practices in JavaScript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 94–105, New York, NY, USA, 2015. ACM.

- [19] S. Guarnieri and B. Livshits. GATEKEEPER: Mostly static enforcement of security and reliability policies for JavaScript code. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 151–168, Berkeley, CA, USA, 2009. USENIX Association.
- [20] S. Guarnieri and B. Livshits. GULFSTREAM: Staged static analysis for streaming JavaScript applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development*, WebApps'10, pages 6–6, Berkeley, CA, USA, 2010. USENIX Association.
- [21] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the World Wide Web from vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 177–187, New York, NY, USA, 2011. ACM.
- [22] S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 214–236, Berlin, Heidelberg, 2003. Springer-Verlag.
- [23] D. Hauzar and J. Kofroň. Framework for static analysis of PHP applications. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 689–711, 2015.
- [24] P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 200–224, Berlin, Heidelberg, 2010. Springer-Verlag.
- [25] M. Hills, P. Klint, and J. Vinju. An empirical study of PHP feature usage: A static analysis perspective. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 325–335, New York, NY, USA, 2013. ACM.
- [26] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.
- [27] A. Holkner and J. Harland. Evaluating the dynamic behaviour of Python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, ACSC '09, pages 19–28, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.
- [28] S. H. Jensen, P. A. Jonsson, and A. Møller. Remediating the eval that men do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 34–44, New York, NY, USA, 2012. ACM.

- [29] 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 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11*, pages 59–69, New York, NY, USA, 2011. ACM.
- [30] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis, SAS '09*, pages 238–255, Berlin, Heidelberg, 2009. Springer-Verlag.
- [31] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural analysis with lazy propagation. In *Proceedings of the 17th International Conference on Static Analysis, SAS'10*, pages 320–339, Berlin, Heidelberg, 2010. Springer-Verlag.
- [32] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy, SP '06*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.
- [33] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wieder-
mann, and B. Hardekopf. Jsai: A static analysis platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014*, pages 121–132, 2014.
- [34] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, pages 423–434, New York, NY, USA, 2013. ACM.
- [35] E. Kneuss, P. Suter, and V. Kuncak. Runtime instrumentation for precise flow-sensitive type analysis. In *Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 300–314. Springer Berlin Heidelberg, 2010.
- [36] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *International Workshop on Foundations of Object Oriented Languages, FOOL' 12*, 2012.
- [37] B. S. Lerner, J. G. Politz, A. Guha, and S. Krishnamurthi. Tejas: Retrofitting type systems for JavaScript. In *Proceedings of the 9th Symposium on Dynamic Languages, DLS '13*, pages 1–16, New York, NY, USA, 2013. ACM.
- [38] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: Is it worth it? In *Proceedings of the 15th International Conference on Compiler Construction, CC'06*, pages 47–64, Berlin, Heidelberg, 2006. Springer-Verlag.
- [39] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):3:1–3:53, Oct. 2008.

- [40] P. Liang and M. Naik. Scaling abstraction refinement via pruning. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 590–601, New York, NY, USA, 2011. ACM.
- [41] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '86, pages 214–223, New York, NY, USA, 1986. ACM.
- [42] 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 soundness: A manifesto. *Commun. ACM*, 58(2):44–46, Jan. 2015.
- [43] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 499–509, New York, NY, USA, 2013. ACM.
- [44] J. Malenfant, M. Jacques, and F. N. Demers. A tutorial on behavioral reflection and its implementation. In *Proceedings of the Reflection*, volume 96, pages 1–20, 1996.
- [45] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: A unified model. *Acta Inf.*, 28(2):121–163, Dec. 1990.
- [46] F. Meawad, G. Richards, F. Morandat, and J. Vitek. Eval begone!: Semi-automated removal of eval from JavaScript programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 607–620, New York, NY, USA, 2012. ACM.
- [47] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, Jan. 2005.
- [48] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. JSEFT: automated JavaScript unit test generation. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–10, 2015.
- [49] F. Nentwich, N. Jovanovic, E. Kirida, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, 2007.
- [50] F. S. Ocariza Jr., K. Bajaj, K. Pattabiraman, and A. Mesbah. An empirical study of client-side JavaScript bugs. *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 0:55–64, 2013.
- [51] F. S. Ocariza Jr., K. Pattabiraman, and B. Zorn. JavaScript errors in the wild: An empirical study. In *Proceedings of the 2011 IEEE 22Nd International Symposium on Software Reliability Engineering*, ISSRE '11, pages 100–109, Washington, DC, USA, 2011. IEEE Computer Society.

- [52] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi. Selective context-sensitivity guided by impact pre-analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 475–484, New York, NY, USA, 2014. ACM.
- [53] C. Park and S. Ryu. Scalable and precise static analysis of JavaScript applications via loop-sensitivity. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 735–756, 2015.
- [54] M. Pradel and K. Sen. The good, the bad, and the ugly: An empirical study of implicit type conversions in JavaScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 519–541, 2015.
- [55] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.
- [56] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real web applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development, WebApps'10*, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.
- [57] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of JavaScript benchmarks. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '11*, pages 677–694, New York, NY, USA, 2011. ACM.
- [58] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In *Proceedings of the 25th European Conference on Object-oriented Programming, ECOOP'11*, pages 52–78, Berlin, Heidelberg, 2011. Springer-Verlag.
- [59] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10*, pages 1–12, New York, NY, USA, 2010. ACM.
- [60] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proceedings of the 12th international conference on Compiler construction, CC'03*, pages 126–137, Berlin, Heidelberg, 2003. Springer-Verlag.
- [61] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP '10*, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society.

- [62] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Dynamic determinacy analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 165–174, New York, NY, USA, 2013. ACM.
- [63] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013*, pages 488–498, New York, NY, USA, 2013. ACM.
- [64] R. Sethi. *Programming Languages, Concepts & Constructs, 2nd edition*. Addison Wesley, 1996.
- [65] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, pages 189–234, 1981.
- [66] O. G. Shivers. *Control-flow Analysis of Higher-order Languages of Taming Lambda*. PhD thesis, Carnegie Mellon University, 1991.
- [67] Y. Smaragdakis and G. Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015.
- [68] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 17–30, New York, NY, USA, 2011. ACM.
- [69] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 485–495, New York, NY, USA, 2014. ACM.
- [70] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 387–400, New York, NY, USA, 2006. ACM.
- [71] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav. Alias analysis for object-oriented programs. In D. Clarke, J. Noble, and T. Wrigstad, editors, *Aliasing in Object-Oriented Programming*, pages 196–232. Springer-Verlag, Berlin, Heidelberg, 2013.
- [72] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of JavaScript. In *Proceedings of the 26th European conference on Object-Oriented Programming*, ECOOP'12, pages 435–458, Berlin, Heidelberg, 2012. Springer-Verlag.

- [73] Stackoverflow. Stackoverflow 2015 developer survey. <http://stackoverflow.com/research/developer-survey-2015#tech>.
- [74] O. Tripp, P. Ferrara, and M. Pistoia. Hybrid security analysis of web JavaScript code via dynamic partial evaluation. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014*, pages 49–59, New York, NY, USA, 2014. ACM.
- [75] W3Techs. W3techs web technology surveys. http://w3techs.com/technologies/overview/client_side_language/all.
- [76] P. Wegner. Dimensions of object-based language design. In *Conference proceedings on Object-oriented programming systems, languages and applications, OOPSLA '87*, pages 168–182, New York, NY, USA, 1987. ACM.
- [77] S. Wei and B. G. Ryder. Practical blended taint analysis for JavaScript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013*, pages 336–346, New York, NY, USA, 2013. ACM.
- [78] S. Wei and B. G. Ryder. State-sensitive points-to analysis for the dynamic behavior of JavaScript objects. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, pages 1–26, 2014.
- [79] S. Wei and B. G. Ryder. Adaptive context-sensitive analysis for JavaScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 712–734, 2015.
- [80] S. Wei, F. Xhakaj, and B. G. Ryder. Empirical study of the dynamic behavior of JavaScript objects. *Software: Practice and Experience*, 2015.
- [81] C. Yue and H. Wang. Characterizing insecure JavaScript practices on the web. In *Proceedings of the 18th International Conference on World Wide Web, WWW '09*, pages 961–970, New York, NY, USA, 2009. ACM.
- [82] C. Yue and H. Wang. A measurement study of insecure JavaScript practices on the web. *ACM Trans. Web*, 7(2):7:1–7:39, May 2013.