UNIVERSITY OF CALIFORNIA
Santa Barbara

Configurable and Sound Static Analysis of JavaScript: Techniques and Applications

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Vineeth Kashyap

Committee in Charge:

Professor Ben Hardekopf, Chair

Professor Chandra Krintz

Professor Tim Sherwood

March 2015

The Dissertation of
Vineeth Kashyap is approved:

Professor Chandra Krintz

Professor Tim Sherwood

Professor Ben Hardekopf, Committee Chairperson

December 2014

Configurable and Sound Static Analysis of JavaScript: Techniques and Applications

by

Vineeth Kashyap

I dedicate this thesis to my parents.

Acknowledgements

I am heavily indebted to a lot of people who positively influenced my PhD in a variety of ways. Words cannot fully express how thankful I am for having our paths crossed. I thank the following people, and for a lot more than just what I give them credit for here.

Ben Hardekopf; for being a dream PhD advisor—while it would take at least a few paragraphs to thank him, I will resort to being short and and say this PhD would not have happened without him.

Chandra Krintz; for instilling the much required enthusiasm in me through her own.

Tim Sherwood; for making me believe that it is okay to have crazy ideas in research.

Ben Wiedermann; for showing me how to have rigor in research and writing.

Kyle Dewey; for being available to bounce off my half-baked ideas and being of tremendous help in many of my research projects.

Current and previous members of the **PL Lab at UCSB** (Madhukar Kedlaya, Lawton Nichols, Dianne Wagner, Jared Roesch, Berkeley Churchill, Kevin Gibbons, Kevin Francis); for making our lab fun ($\tau.\rho.\omega.\lambda.\lambda$) and interesting.

John Sarracino and **Tommy Ashmore**; for a nice summer of collaborations.

Dave Herman; for bringing JavaScript into my research and mentoring me for two beautiful Mountain View summers.

Anand Kodaganur; for planting the crazy idea of pursuing a PhD in me.

Parents; for my existence, and always being there for me with their unconditional support.

I am sorry if I have missed any names—it is not deliberate. I would also like to thank the NSF whose generous funding made my research possible, and SIGPLAN and SIGSOFT for funding my conference travels.

Curriculum Vitæ

Vineeth Kashyap

Education

2009 – 2014	PhD in Computer Science, UCSB
2003 – 2007	BE in Computer Science, RVCE, India

Academic Experience

2010 – 2014	Research Assistant, University of California Santa Barbara
2009 – 2010	Teaching Assistant, University of California Santa Barbara

Industry Experience

2012 Summer	Research Intern, Mozilla Research
2011 Summer	Research Intern, Mozilla Research
2007 – 2009	Software Engineer, National Instruments R&D, India

Conference Publications

2015	**A Parallel Abstract Interpreter for JavaScript** Kyle Dewey, Vineeth Kashyap, and Ben Hardekopf *International Symposium on Code Generation and Optimization (CGO).*

2014 **JSAI: A Static Analysis Platform for JavaScript**

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

ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)

2014 **Sapper: A Language for Hardware-Level Security Policy Enforcement**

Xun Li, Vineeth Kashyap, Jason Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Tim Sherwood, Ben Hardekopf, and Frederic T. Chong

International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)

2014 **Security Signature Inference for JavaScript-based Browser Addons**

Vineeth Kashyap and Ben Hardekopf

International Symposium on Code Generation and Optimization (CGO)

2014 **Widening for Control-Flow**

Ben Hardekopf, Ben Wiedermann, Berkeley Churchill, and Vi-

neeth Kashyap

International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)

2013 **Type Refinement for Static Analysis of JavaScript**

Vineeth Kashyap, John Sarracino, John Wagner, Ben Wiedermann, and Ben Hardekopf

Dynamic Languages Symposium (DLS)

2011 **Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach**

Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf

IEEE Symposium on Security and Privacy (S&P)

2011 **Caisson: a Hardware Description Language for Secure Information Flow**

Xun Li, Mohit Tiwari, Jason Oberg, Vineeth Kashyap, Fred Chong, Tim Sherwood, and Ben Hardekopf

ACM Conference on Programming Language Design and Implementation (PLDI)

Abstract

Configurable and Sound Static Analysis of JavaScript: Techniques and Applications

Vineeth Kashyap

JavaScript is widespread. Web developers use JavaScript to enrich user experience via dynamic content ranging from scripts to enhance a web page's appearance, to full-blown web applications, to extending the functionality of web browsers in the form of browser addons. Desktop developers use JavaScript, e.g., for OS applications in Windows 8. JavaScript's growing prominence means that secure, correct, and fast JavaScript code is becoming ever more critical. Static analysis traditionally plays a large role in providing these characteristics: it can be used for security auditing, error-checking, debugging, optimization, and program refactoring, among other uses. Thus, a sound, precise static analysis platform for JavaScript can be of enormous advantage.

In this thesis, we present our work on creating a sound, precise, configurable and fast static analysis for JavaScript called JSAI that we have made openly available to the research community. JSAI is both a practical tool for JavaScript static analysis and also a research platform for experimenting with static analysis techniques. JSAI showcases a number of novel techniques to soundly compute a combination of type inference, pointer analysis, control-flow analysis, string analysis, and integer and boolean con-

stant propagation for JavaScript programs. It also provides a unique method for modularly configuring analysis precision that is based on fundamental new insights into the theory of static analysis. We describe precision-increasing techniques for the analysis using type refinement; and performance-increasing techniques for the analysis based on parallelization of JSAI. As an example use-case for JSAI, we discuss a novel security analysis for JavaScript-based browser addon vetting.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

JavaScript is pervasive. While it began as a client-side webpage scripting language, JavaScript has grown hugely in scope and popularity and is used to extend the functionality of web browsers via browser addons, to develop desktop applications (e.g., for Windows 8 [3]) and server-side applications (e.g., using Node.js [4]), and to develop mobile phone applications (e.g., for Firefox OS [5]). JavaScript's growing prominence means that secure, correct, maintainable, and fast JavaScript code is becoming ever more critical. Sound static analysis traditionally plays a large role in providing these characteristics: it can be used for security auditing, error-checking, debugging, optimization, program understanding, refactoring, and more.

However, JavaScript's inherently dynamic nature and many unintuitive quirks cause great difficulty for static analysis. For example, a simple line of code in JavaScript to access an index of a string as given in Figure 1.1 results in potentially several implicit type checks and implicit type conversions, that can lead to invocation of user-defined

code (lines `16` and `27` call methods that can be overridden by the user). Thus, unlike other traditional languages like C/C++ and Java, the static analysis cannot rely on existence of a control flow graph to begin with[1]—we need to perform an detailed analysis to compute the control flow graph.

In addition, static analysis of JavaScript is a very young research field, and the right analysis abstractions are not yet known. The abstractions in a static analysis (like abstract data domains and control-flow sensitivities like context-, heap-, and path- sensitivities) can have a significant impact on the performance and precision of the analysis. Such abstractions vary between programming languages—even between different application domains within the same language. All the existing static analyses for JavaScript (and for most other languages) bake-in such abstractions, making it hard to explore and experiment to find the right abstractions.

Our goal is to overcome these difficulties and provide a formally specified, well-tested, configurable and sound static analysis platform for JavaScript, immediately useful for a variety of client analyses.

[1] One can start the analysis with an unsound control flow graph and dynamically add edges during the analysis, but this is ad-hoc and is prone to errors.

```
 1: if myString is null or undefined then
 2:     type-error

 3: else
 4:     // convert myString to an object first?
 5:     if myString is a primitive then
 6:         obj = toObject(myString)
 7:     else
 8:         obj = myString
 9:     end if

10:     // convert i to a string

11:     // case 1: i is a primitive
12:     if i is a primitive then
13:         prop = toString(i)
14:     else
15:         if i.toString is callable then
16:             tmp = i.toString()
17:         else
18:             goto line 26
19:         end if
20:     end if

21:     // case 2: i is not a primitive, but i.toString() is
22:     if tmp is a primitive then
23:         prop = toString(tmp)

24:     // case 3: i.toString() is not a primitive; try i.valueOf()
25:     else
26:         if i.valueOf is callable then
27:             tmp2 = i.valueOf()
28:         else
29:             type-error
30:         end if

31:         if tmp2 is a primitive then
32:             prop = toString(tmp2)
33:         else
34:             type-error
35:         end if
36:     end if

37:     // retrieve the property from the object
38:     x = obj.prop
39: end if
```

Figure 1.1: The semantics of var x = myString[i];

1.1 Key Insights and Thesis Statement

There were several key ideas and insights that shaped this thesis; we discuss the most important of them here.

- Control-flow sensitivity (ex: flow-, context-, heap-, path-sensitivity, predicate abstraction, property simulation) can expressed modularly from the rest of the analysis specification. This is achieved by describing control-flow sensitivity as a widening operator parameterized by an equivalence relation that partitions states according to an abstraction of the program's history of computation.

- A sound analysis for JavaScript can be achieved by formally specifying the concrete and abstract semantics for JavaScript and connecting the two semantics using abstract interpretation. Specifying semantics in a executable manner using small-step operational abstract machine semantics allows for testing and modular specification of the analysis. The validity of the given concrete semantics for JavaScript can be claimed by thorough testing against a commercial JavaScript engine. Modular control-flow sensitivity allows for experimentation with a wide range of sensitivities for JavaScript analysis.

- Viewing analysis of JavaScript as a state transition system shows that it is highly amenable to parallelization. One can think of control-flow sensitivity used in the analysis as selection of states to merge (by over-approximating the merged

states)—introducing points of synchronization in a embarrassingly parallel exploration.

- The concept of type refinement—where the abstract type information propagated by the static analysis is refined within each branch of a conditional—can vastly aid the precision of JavaScript analysis, particularly when applied to the implicit conditional checks that do not show up in surface syntax.

- Specifying security policies for JavaScript-based browser addons using security signatures (detailed summary of interesting information flows and API usages) and automatically inferring them can help security auditing of third-party addons submitted for code-review in more automated manner. Our sound JavaScript analysis can be used to build such an automatic security signature inference engine.

These insights lead us to my thesis statement: **Configurable, sound, precise, and fast static analysis for JavaScript is feasible. We demonstrate this by building a formalized abstract interpreter for JavaScript called JSAI. JSAI forms the basis for a multitude of useful client analyses including security, error- checking, and program understanding.**

1.2 Contributions and Overview of the Thesis

The main contributions of this thesis (and the chapters that detail them) include:

- We provide a formal theoretical foundation for building configurable (with a wide-range of control-flow sensitivities) abstract interpreter [82] using widening for control flow (Chapter 2)[2].

- We build a configurable and sound abstract interpreter for JavaScript called JSAI [95]. JSAI is open-source, and includes a formally specified concrete and abstract semantics for JavaScript. The abstract semantics specifies a fundamental static analysis for JavaScript that can be used to build a variety of clients on top—the analysis is a combination of type inference, pointer analysis, control-flow analysis, string, number and boolean analysis with novel abstract string and object domains for JavaScript. Our formalisms and code have been positively evaluated by the FSE 2014 Artifact Evaluation Committee. We evaluate JSAI's performance and precision on a comprehensive benchmark suite comprising of multiple application domains and obtain novel insights by experimenting with a large number of context-sensitivities (Chapter 3).

- We improve the precision of JSAI using type refinement [97]—in particular by refining based on implicit checks on types by the JavaScript semantics—and show

[2]The first author of this work is Ben Hardekopf, I am a co-author.

that up to 86% precision improvement can be obtained on a static type-error detection client without affecting performance (Chapter 4).

- We parallelize JSAI [57] based on the insights highlighted in the previous section to obtain speedups between $2-4\times$ on average with a super-linear maximum of $36.9\times$ on 12 hardware threads when compared to the sequential version (Chapter 5)[3]. Our parallelized version of JSAI has been positively evaluated by the CGO 2015 Artifact Evaluation Committee.

- We show the usefulness of JSAI by building a client security auditing analysis for vetting JavaScript-based browser addons [96]. We describe a novel notion of security signature for browser addons and construct an analysis to infer these automatically, and empirically evaluate it on benchmark consisting of real-world addons from the official Mozilla addon repository (Chapter 6).

[3]The first author of this work is Kyle Dewey, I am a co-author.

Chapter 2

Constructing Configurable and Sound Abstract Interpreters via Widening for Control-Flow

2.1 Introduction

A program analysis designer must balance three opposing characteristics: soundness, precision, and tractability. An important dimension of this tradeoff is *control-flow sensitivity*: how precisely the analysis adheres to realizable program execution paths. Examples include various types of *path sensitivity* (e.g., property simulation [54] and predicate abstraction [39]), *flow sensitivity* (e.g., flow-insensitive [42] and flow-sensitive [101]), and *context sensitivity* (e.g., k-CFA [131] and object sensitivity [120]). By tracking realizable execution paths more precisely, the analysis may compute more precise results but also may become less tractable. Thus, choosing the right control-flow

sensitivity for a particular analysis is crucial for finding the sweet-spot that combines useful results with tractable performance.

We present a set of insights and formalisms that allow control-flow sensitivity to be treated as an independent concern, separately from the rest of the analysis design and implementation. This separation of concerns allows the analysis designer to empirically experiment with many different analysis sensitivities in a guaranteed sound manner, without modifying the analysis design or implementation. These sensitivities are not restricted to currently known strategies; the designer can easily develop and experiment with new sensitivities as well. Besides allowing manual exploration of potential new sensitivities, we also describe a mechanism to automatically create new sensitivities, based on the insight that the space of control-flow sensitivities forms a lattice. The meet and join operators of this lattice can be used to construct novel sensitivities from existing ones without requiring manual intervention.

Key Insights. Our key insight is that control-flow sensitivity is a form of widening, and that we can exploit this to separate control-flow sensitivity from the rest of the analysis. This chapter describes control-flow sensitivity as a widening operator parameterized by an equivalence relation that partitions states according to an abstraction of the program's history of computation. This widening-based view of control-flow sensitivity has both theoretical and practical implications: it generalizes and modularizes

existing insights into control-flow sensitivity, and provides the analysis designer with a method for implementing and evaluating many possible sensitivities in a modular way.

A common technique to formalize control-flow sensitivity is to abstract a program's concrete control flow as an abstract trace (i.e., some notion of the history of computation that led to a particular program point). There are many ways to design such an abstraction, including ad-hoc values that represent control-flow (e.g., the timestamps of van Horn and Might [138]), designed abstractions with a direct connection to the concrete semantics (e.g., the mementoes of Nielson and Nielson [123]), and calculated abstractions that result from the composition of Galois connections (e.g., the 0-CFA analysis derived by Midtgaard and Jensen [118]). Existing formalisms are also tied to the notion of abstraction by partitioning [51]: the control-flow abstraction partitions the set of states into equivalence relations, the abstract values of which are merged.

Our formalisms follow this general approach (tracing and partitioning). However, prior work starts from a subset of known control-flow approximations (e.g, context-sensitivity [102, 123, 132], 0-CFA [118], or various forms of k-limiting and store value-based approximations [112, 125]) and seeks to formalize and prove sound those specific control-flow approximations for a given analysis. In addition, most prior work calculates a series of Galois connections that leads to a specific (family of) control-flow sensitivity. In contrast, our work provides a more general view that specifies a superset of the control-flow sensitivities specified by prior work and exposes the possibility of

many new control-flow sensitivities, while simplifying the required formalisms and enabling a practical implementation based directly on our formalisms. As such, our work has similar goals to Might and Manolios' *a posteriori* approach to soundness, which separates many aspects of the precision of an analysis from its soundness [119]; however, our technique relies on a novel insight that connects widening and control-flow sensitivity.

Contributions. This chapter makes the following contributions:

A new formulation of control-flow sensitivity as a widening operator, which generalizes and modularizes existing formulations based on abstraction by partitioning. This formulation leads to a method for designing and implementing a program analysis so that control-flow sensitivity is a separate and independent component. The chapter describes several requirements on the form a semantics should take to enable separable control-flow sensitivity. Individually these observations are not novel; in fact, they may be well-known to the community. When collectively combined, however, they form an analysis design that permits sound, tunable control-flow approximation via widening. (Section 2.2)

2.2 Separating Control-Flow Sensitivity from an Analysis

In this section, we describe how to use widening to separate control-flow sensitivity from the rest of the analysis and make it an independent concern. We first establish our starting point: an abstract semantics that defines an analysis with no notion of sensitivity. We then describe a parameterized widening operator for the analysis and show how different instantiations of the parameter yield different control-flow sensitivities. Finally, we discuss some requirements on the form of semantics used by the analysis that make it amenable to describing control-flow sensitivity. The discussion in this section leaves the exact language and semantics being analyzed unspecified.

2.2.1 Starting Point

This subsection provides background and context on program analysis, giving us a starting point for our design. Nothing in this subsection is novel, the material is adapted from existing work [49]. For concreteness, we assume that the abstract semantics is described as a state transition system, e.g., a small-step abstract machine semantics; Section 2.2.4 will discuss more general requirements on the form of the semantics. The abstract semantics is formally described as a set of states $\hat{\varsigma} \in \Sigma^\sharp$ and a transition relation between states $\mathcal{F}^\sharp \subseteq \Sigma^\sharp \times \Sigma^\sharp$. The semantics uses a transition relation instead

of a function to account for nondeterminism in the analysis due to uncertain control-flow (e.g., when a conditional guard's truth value is indeterminate, and so the analysis must take both branches). The set of states forms a lattice $\mathcal{L}^\sharp = (\Sigma^\sharp, \sqsubseteq, \sqcap, \sqcup)$. We leave the definition of states and the transition relation unspecified, but we assume that any abstract domains used in the states are equipped with a widening operator.[1]

The program analysis is defined as the set of all reachable states starting from some set of initial states and iteratively applying the transition relation. This definition is formalized as a least fixpoint computation. Let $\mathring{\mathcal{F}}^\sharp(S) \stackrel{\textbf{def}}{=\!=} S \cup \mathcal{F}^\sharp(S)$, i.e., a relation that is lifted to remember every state visited by the transition relation $\mathcal{F}^\sharp$. The analysis of a program P is defined as $[\![P]\!]^\sharp \stackrel{\textbf{def}}{=\!=} \textbf{lfp}_{\Sigma^\sharp_{\mathcal{I}}} \mathring{\mathcal{F}}^\sharp$, i.e., the least fixpoint of $\mathring{\mathcal{F}}^\sharp$ starting from an initial set of states $\Sigma^\sharp_{\mathcal{I}}$ derived from P.

The analysis $[\![P]\!]^\sharp$ is intractable, because the set of reachable states is either infinite or, at the least, exponential in the number of nondeterministic transitions made during the fixpoint computation. The issue is control-flow—specifically, the nondeterministic choices that must be made by the analysis: which branch of a conditional should be taken, whether a loop should be entered or exited, which (indirect) function should be called, etc. The analysis designer at this point must either (1) bake into the abstract semantics a specific strategy for dealing with control-flow; or (2) ignore the issue in the formalized analysis design and use an ad-hoc strategy in the analysis implementation.

[1] If the domain is a noetherian lattice then the lattice join operator is a widening operator.

Our proposed widening operator is a means to formalize control-flow sensitivity in a manner that guarantees soundness, but does not require that a sensitivity to be baked into the semantics. On a practical level, it also allows the analysis designer to experiment with many different sensitivities without modifying the analysis implementation.

2.2.2 Widening Operator

Our goal is to *limit* the number of states contained in the fixpoint, while still retaining soundness. We do so by defining a widening operator for the fixpoint computation, which acts on entire sets of states rather than on individual abstract domains inside the states. This widening operator: (1) partitions the current set of reachable states into disjoint sets; (2) merges all of the states in each partition into a single state that over-approximates that partition; and (3) unions the resulting states together into a new set that contains only a single state per partition. The widening operator controls the performance and precision of the analysis by setting a bound on the number of states allowed: there can be at most one state per partition. Decreasing the number of partitions can speed up the fixpoint computation, thus helping performance, but can also merge more states together in each partition, thus hindering precision.

Formally, the widening operator for control-flow sensitivity is parameterized by a (unspecified) equivalence relation $\sim$ on abstract states. Given a widening operator $\triangledown$ on individual abstract domains, our new widening operator $\triangledown^\sharp$ is defined as:

$$\nabla^\sharp \in \mathcal{P}(\Sigma^\sharp) \times \mathcal{P}(\Sigma^\sharp) \to \mathcal{P}(\Sigma^\sharp)$$
$$A \nabla^\sharp B = \left\{ \nabla_{\hat{\varsigma} \in X} \hat{\varsigma} \;\middle|\; X \in (A \cup B)/\sim \right\}$$

where for a set S the notation $S/\sim$ means the set of partitions of S according to equivalence relation $\sim$, and the widening operator ∇ on individual abstract domains is used to merge the states in each resulting partition into a single state. Note that if the number of partitions induced by $\sim$ is finite, then the number of states in each partition is also finite because we apply the widening operator at each step of the fixpoint computation.

Theorem 2.2.1 (WIDENING)**.** *If the number of partitions induced by $\sim$ is finite, then $\nabla^\sharp$ is a widening operator.*

Proof. Follows from the definition of a widening operator [50]. □

We now lift the transition relation $\mathcal{F}^\sharp$ in a similar fashion as before, except instead of using set union we use our widening operator: $\overset{\nabla}{\mathcal{F}}{}^\sharp(S) \overset{\textbf{def}}{=\!=} S \;\nabla^\sharp\; \mathcal{F}^\sharp(S)$. Then the control-flow sensitive abstract semantics is defined as $[\![P]\!]^\sharp_\nabla \overset{\textbf{def}}{=\!=} \textbf{lfp}_{\Sigma^\sharp_\mathcal{I}} \overset{\nabla}{\mathcal{F}}{}^\sharp$.

Even though we have not specified the equivalence relation that parameterizes the widening operator, we can still prove the soundness of the analysis. Informally, because the widening operator merges the states within each partition using ∇, the reachable

states using $\overset{\triangledown}{\mathcal{F}}^\sharp$ over-approximate the reachable states using $\overset{\circ}{\mathcal{F}}^\sharp$. Thus, the control-flow sensitive abstract semantics is sound with respect to the original abstract semantics:

Theorem 2.2.2 (SOUNDNESS)**.**

$$\gamma([\![P]\!]^\sharp) \subseteq \gamma([\![P]\!]^\sharp_\triangledown)$$

Proof. We must show that (1) the least fixpoint denoted by $[\![P]\!]^\sharp_\triangledown$ exists; and (2) it over-approximates $[\![P]\!]^\sharp$.

1. The existence of the fixpoint follows from part 2 of the definition of a widening operator as given by Cousot and Cousot [50, def. 9.1.3.3].

2. That the widened fixpoint over-approximates the original fixpoint follows from part 1 of the definition of a widening operator as given by Cousot and Cousot [50, defs. 9.1.3.1–9.1.3.2].

□

2.2.3 Control-Flow Sensitivity

It remains to show how our widening operator determines the control-flow sensitivity of the analysis. The determining factor is how the states are partitioned, which is controlled by the specific equivalence relation on states $\sim$ that parameterizes the

widening operator. The question is, what constitutes a good choice for the equivalence relation? For Theorem 2.2.1 to hold, it must induce a finite number of partitions, but what other characteristics should it have? Our goal is tractability with a minimal loss of precision; this means we should try to partition the states so that there are a tractable number of partitions *and* the states within each partition are as similar to each other as possible (to minimize the information lost to merging).

A reasonable heuristic is to partition states based on how those states were computed, i.e., the execution history that led to each particular state. The hypothesis is that if two states were derived in a similar way then they are more likely to be similar. This heuristic of similarity is exactly the one used by existing control-flow sensitivities, such as flow-sensitive maximal fixpoint, k-CFA, object-sensitivity, property simulation, etc. These sensitivities each compute an abstraction of the execution history (e.g., current program point, last k call-sites, last k allocation sites, etc.) and use that abstraction to partition and merge the states during the analysis.

Therefore, the widening operator should partition the set of states according to their control-flow sensitivity approximation:

$$\hat{\varsigma}_1 \sim \hat{\varsigma}_2 \iff \pi_{\hat{\tau}}(\hat{\varsigma}_2) = \pi_{\hat{\tau}}(\hat{\varsigma}_2)$$

where each state contains an *abstract trace* $\hat{\tau}$ describing some abstraction of the execution history, and $\pi_{\hat{\tau}}(\hat{\varsigma})$ projects a state's abstract trace. This definition causes the widening operator to merge all states with the same trace, i.e., all states with the same approximate execution history. The widened analysis can be defined without specifying a particular abstract trace domain; different trace domains can be plugged in after the fact to yield different sensitivities.

Trace Abstractions. We have posited that control-flow sensitivity is based on an abstraction of the execution history of a program, called a trace. This implies that the trace abstraction is related to the *trace-based* concrete collecting semantics, which contains all reachable execution paths, i.e., sequences of states, rather than just all reachable states. An abstract trace is an abstraction of a set of paths in the concrete collecting semantics. For example, a flow-sensitive trace abstraction records the current program point, abstracting all paths that reach that program point. A context-sensitive trace abstraction additionally records the invocation context of the current function, abstracting all paths that end in that particular invocation context (e.g., as in Nielson and Nielson's mementoes [123]). Different forms of context-sensitivity define the abstract "context" differently: for example, traditional k-CFA defines it as the last k call-sites encountered in the concrete trace; stack-based k-CFA considers the top k currently active (i.e.,

not yet returned) calls on the stack; object sensitivity considers abstract allocation sites instead of call-sites; and so on.

We note that it is not necessary for the trace abstraction to soundly approximate the concrete semantics for the resulting analysis to be sound. The trace abstraction is a heuristic for partitioning the states; as long as the number of elements in the trace abstraction domain is finite (and hence the number of partitions enforced by the widening operator is finite), the analysis will terminate with a sound solution. In fact, it isn't strictly necessary for $\sim$ to be based on control-flow at all—exploring other heuristics for partitioning states would be in interesting avenue for future work.

2.2.4 Semantic Requirements

To benefit from widening-based control-flow sensitivity, an abstract semantics must satisfy certain requirements. To abstract control, the analysis must be able to introduce new program execution paths that over-approximate existing execution paths. To make this possible, we argue that there should be some explicit notion in the program semantics of the "rest of the computation"—i.e., a continuation. When the analysis abstracts control, it is abstracting these continuations. The explicit control-flow representation can take a number of possible forms. For example, it could be in the form of a syntactic continuation (e.g., if a program is in continuation-passing style then the "rest of the computation" is given as a closure in the store) or a semantic continuation (e.g.,

the continuation stack of an abstract machine). Since the abstract states form a lattice, any two distinct states must have a join, and (according to our requirement) this joined state must contain a continuation that over-approximates the input states' continuations. Thus, by joining states the analysis approximates control as well as data.

Some forms of semantics do not meet this requirement, including various forms proposed as being good foundations for abstract interpretation [115, 129, 130]. For example, big-step and small-step structural operational semantics implicitly embed the continuations in the semantic rules. Direct-style denotational semantics similarly embeds this information in the translation to the underlying meta-language. This means that there is no way to abstract and over-approximate control-flow; the analysis must use whatever control-flow the original semantics specifies (or, alternatively, use ad-hoc strategies baked into the analysis implementation to silently handle control-flow sensitivity). Some limited forms of control-flow sensitivity may still be expressed when the analysis takes care to join only those states that already have the same continuation (e.g., flow-sensitive maximal fixpoint), but many other forms (e.g., k-CFA or other forms of context-[in]sensitivity) remain difficult to express.

2.3 Related Work

In abstract interpretation, there is a relatively small but dedicated body of research on trace abstraction and on formalizing control-flow sensitivity as partitioning. What distinguishes our work from most prior efforts is a different focus: prior work focuses on the *integration* of control-flow abstractions into an existing analysis; our work focuses on the *separation* of control-flow abstractions from an existing analysis, so that it is easier for analysis designers to experiment with different control-flow sensitivities. In this section, we discuss the implications of these differences. Broadly, no prior work has couched control-flow sensitivity in terms of a widening operator based on abstractions of the program history, which permits a simpler, more general, and more tunable formulation of control-flow sensitivity.

***A Posteriori* Soundness.** Our work is most similar to Might and Manolios' *a posteriori* soundness for non-deterministic abstract interpretation [119], which also seeks to separate the aspects of an analysis that affect its precision from those that affect its soundness. Both techniques achieve this separation by introducing a level of indirection, although the mechanisms are different. Our technique uses an equivalence relation that partitions abstract states. Might and Manolios' uses an *abstract allocation policy* that can dynamically allocate the resources that determine how to partition abstract states. We accomplish soundness by leveraging the soundness of widening.

Might and Manolios accomplish soundness via their technique of an *a posteriori* proof: their abstract allocation policies induce a non-deterministic abstract semantics that can be shown to produce sound analysis results, even though the abstract semantics do not conform to the traditional simulation of the concrete semantics. Our work also re-formulates one of Might and Manolios' insights: that most control-flow (or heap-allocation) approximations are already sound because they add only extra information to the analysis. A particular strength of Might and Manolios' work is that it makes it easy to express sound, adaptable analyses. A particular strength of our work is that it makes it easy to declaratively describe many forms of analyses and to systematically combine them. It is not clear whether the two techniques are equally expressive, nor whether they are equally useful in practice. An interesting line of research would be to explore how well each technique is suited to the practical discovery, design, and implementation of precise analyses and how the two techniques might compete with or complement each other.

Trace Partitioning. Our work is similar in some respects to the trace partitioning work by Mauborgne and Rival [112, 125], which itself builds on the abstraction-by-partitioning of control-flow by Handjieva and Tzolovski [81]. Trace partitioning was developed in the context of the ASTRÉE static analyzer [52] for a restricted subset of the C language, primarily intended for embedded systems. Mauborgne and Rival

observe that usually abstract interpreters are (1) based on reachable states collecting semantics, making it difficult to express control-flow sensitivity; and (2) designed to silently merge information at control-flow join points[2]—what in dataflow analysis is called "flow-sensitive maximal fixpoint" [92]. They propose a method to postpone these silent merges when doing so can increase precision; effectively they add a controlled form of path-sensitivity. They formalize their technique as a series of Galois connections.

Mauborgne and Rival describe a denotational semantics-based analysis that can use three criteria to determine whether to merge information at a particular point: the last k branch decisions taken (i.e., whether an execution path took the *true* or *false* branch); the last k while-loop iterations (effectively unrolling the loop k times); and the value of some distinguished variable. These criteria are guided by syntactic hints inserted into a program prior to analysis; the analysis itself can choose to ignore these hints and merge information regardless, as a form of widening. This feature is a form of *dynamic partitioning*, where the choice of partition is made as the analysis executes. Our sum abstraction (Section **??**) is another form of dynamic partitioning.

The analysis described by Mauborgne and Rival requires that the program is non-recursive; it fully inlines all procedure calls to attain complete context-sensitivity. Because the semantics they formulate does not contain an explicit representation of con-

[2]By which they mean that the abstract semantics say nothing about merging information, but the implementation does so anyway.

tinuations, there is no way in their described system to achieve other forms of context-sensitivity (e.g., k-CFA, including 0-CFA, i.e., context-insensitive analysis) without heavily modifying their design, implementation, and formalisms (cf. our discussion in Section 2.2.4). Because our method seeks more generality, it can express all of the sensitivities described by Mauborgne and Rival.

Predicate Abstraction. Fischer et al. [67] propose a method to join dataflow analysis with predicate abstraction using *predicate lattices* to gain a form of tunable intra-procedural path-sensitivity. At a high level these predicate lattices perform a similar "partition and merge" strategy as our own method. However, our method is more general: we can specify many more forms of control-flow sensitivity due to our insights regarding explicit control state. One can consider their work as a specific instantiation of our method using predicates as the trace abstraction. On the other hand, Fisher et al. use predicate refinement to *automatically* determine the set of predicates to use, which is outside the current scope of our method. In order to do the same, our method would need to add a predicate refinement strategy.

Context Sensitivity. There are several papers that describe various abstract interpretation-based approaches to specific forms of context sensitivity, including Nielson and Nielson [123], Ashley and Dybvig [36], Van Horn and Might [138], and Midtgaard and Jensen [117, 118]. Nielson and Nielson describe a form of context-sensitivity based

on abstractions of the history of a program's calls and returns [123]. Although this formulation is separable, it is not as general as the one described in this chapter. For example, it cannot capture calls and returns in obfuscated binaries (which may contain no explicit calls and returns); to capture such behavior, a different formulation similar to property simulation is required [102]. Our parameterized, widening-based approach we describe is general enough to capture *either* of these formulations (and many more).

Ashley and Dybvig [36] give a reachable states collecting semantics formulation of k-CFA for a core Scheme-like language; they instrument both the concrete and abstract semantics with a *cache* that collects CFA information. The analysis as described in the paper is intractable (i.e., although it yields the same precision as k-CFA, the number of states remains exponential in the size of the program). Ashley and Dybvig implement a tractable, flow-insensitive version of the analysis independently from the formally-derived version, rather than deriving the tractable version directly from the formal semantics.

Van Horn and Might [138] also give a method for constructing analyses, using an abstract machine-based, reachable states collecting semantics of the lambda calculus. Their analysis includes a specification of k-CFA. An important contribution of their paper is a technique to abstract the infinite domains used for environments and semantic continuations using store allocation. As with Ashley and Dybvig, the analysis as described in their paper does not directly yield a tractable analysis. Van Horn and Might

describe a tractable version of their analysis (not formally derived from the language semantics) that uses a single, global store to improve efficiency, but disallows flow-sensitive analysis because it computes a single solution for the entire program.

Midtgaard and Jensen [117] derive a tractable, demand-driven 0-CFA analysis for a core Scheme-like language using abstract interpretation. Their technique specifically targets 0-CFA, rather than general k-CFA. They employ a series of abstractions via Galois connections, the composition of which leads to the final 0-CFA analysis. In a later paper, Midtgaard and Jensen derive another 0-CFA analysis to compute both call *and* return information [118]. We illustrate how to achieve a sound analysis with *arbitrary* control-flow sensitivity, without having to derive the soundness for each sensitivity.

2.4 Conclusions and Future Work

We have presented a method for program analysis design and implementation that allows the analysis designer to parameterize over control-flow abstractions. This separation of concerns springs from a novel theoretical insight that control-flow sensitivity is induced by a widening operator parameterized by trace abstractions. Our method makes it easier for an analysis designer to specify, implement, and experiment with many forms of control-flow sensitivity, which is critical for developing new, practical analyses. Our future work involves exploring these ideas further, for example, using

combinatorial optimization to explore the vast space of possible trace abstractions. Additionally, our method applies not only to control-flow but to *any* property of a program that can be abstracted and that might be useful to partition the analysis state-space.

Ultimately, the goal of our work is to raise the level of abstraction for analysis designers so that we spend less time specifying and implementing new ideas, and more time formulating and evaluating them.

Chapter 3

JSAI: The JavaScript Abstract Interpreter

3.1 Introduction

JavaScript is pervasive. While it began as a client-side webpage scripting language, JavaScript has grown hugely in scope and popularity and is used to extend the functionality of web browsers via browser addons, to develop desktop applications (e.g., for Windows 8 [3]) and server-side applications (e.g., using Node.js [4]), and to develop mobile phone applications (e.g., for Firefox OS [5]). JavaScript's growing prominence means that secure, correct, maintainable, and fast JavaScript code is becoming ever more critical. Static analysis traditionally plays a large role in providing these characteristics: it can be used for security auditing, error-checking, debugging, optimization, program understanding, refactoring, and more. However, JavaScript's inherently dynamic nature and many unintuitive quirks cause great difficulty for static analysis.

Our goal is to overcome these difficulties and provide a formally specified, well-tested static analysis platform for JavaScript, immediately useful for many client analyses such as those listed above. In fact, we have used JSAI in previous work to build a security auditing tool for browser addons [96] and to experiment with strategies to improve analysis precision [97]. We have also used JSAI to build a static program slicing [141] client and to build a novel abstract slicing [143] client. These are only a few examples of JSAI's usefulness.

Several important characteristics distinguish JSAI from existing JavaScript static analyses (which are discussed further in Section 6.7):

- JSAI is formally specified. We base our analysis on formally specified *concrete* and *abstract* JavaScript semantics. The two semantics are connected using abstract interpretation; we have soundness proof sketches for our most novel and interesting abstract analysis domain. JSAI handles JavaScript as specified by the ECMA 3 standard [62] (sans `eval` and family), and various language extensions such as Typed Arrays [6].

- JSAI's concrete semantics have been extensively tested against an existing commercial JavaScript engine, and the JSAI abstract semantics have been extensively tested against the concrete semantics for soundness.

- JSAI's analysis sensitivity (i.e., path-, context-, and heap-sensitivity) are user-configurable independently from the rest of the analysis. This means that JSAI allows arbitrary sensitivities as defined by the user rather than only allowing a small set of baked-in choices, and that the sensitivity can be set independently from the rest of the analysis or any client analyses.

JSAI's contributions include complete formalisms for concrete and abstract semantics for JavaScript along with implementations of concrete and abstract interpreters based on these semantics. While concrete semantics for JavaScript have been proposed before, ours is the first designed specifically for abstract interpretation. Our abstract semantics is the first formal abstract semantics for JavaScript in the literature. The abstract interpreter implementation is the first available static analyzer for JavaScript that provides easy configurability as a design goal. All these contributions are available freely for download as supplementary materials[1]. JSAI provides a solid foundation on which to build multiple client analyses for JavaScript. The specific contributions of this chapter are:

- The design of a JavaScript intermediate language and concrete semantics intended specifically for abstract interpretation (Section 3.3.1).

- The design of an abstract semantics that enables configurable, sound abstract interpretation for JavaScript (Section 3.3.2). This abstract semantics represents a

[1] `http://www.cs.ucsb.edu/~pllab`, under `Downloads`.

reduced product of type inference, pointer analysis, control-flow analysis, string analysis, and number and boolean constant propagation.

- Novel abstract string and object domains for JavaScript analysis (Section 3.3.3).
- A discussion of JSAI's configurable analysis sensitivity, including two novel context sensitivities for JavaScript (Section 3.4).
- An evaluation of JSAI's performance and precision on the most comprehensive suite of benchmarks for JavaScript static analysis that we are aware of, including browser addons, machine-generated programs via Emscripten [7], and open-source JavaScript programs (Section 3.5). We showcase JSAI's configurability by evaluating a large number of context-sensitivities, and point out novel insights from the results.

We preface these contributions with a discussion of related work (Section 6.7) and conclude with plans for future work (Section 5.6).

3.2 Related Work

In this section we discuss existing static analyses and hybrid static/dynamic analyses for JavaScript and discuss previous efforts to formally specify JavaScript semantics.

JavaScript Analyses. The current state-of-the-art static analyses for JavaScript usually take one of two approaches: **(1)** an unsound[2] dataflow analysis-based approach using baked-in abstractions and analysis sensitivities [41, 75, 85], or **(2)** a formally-specified type system requiring annotations to existing code, proven sound with respect to a specific JavaScript formal semantics but restricted to a small subset of the full JavaScript language [46, 77, 84, 135]. No existing JavaScript analyses are formally specified, implemented using an executable abstract semantics, tested against a formal concrete semantics, or target configurable sensitivity.

The closest related work to JSAI is the JavaScript static analyzer TAJS by Jensen et al [87, 89, 90]. While TAJS is intended to be a sound analysis of the entire JavaScript language (sans dynamic code injection), it does not possess any of the characteristics of JSAI described in Section 6.1. The TAJS analysis is not formally specified and the TAJS papers have insufficient information to reproduce the analysis; also the analysis implementation is not well documented, making it difficult to build client analyses or modify the main TAJS analysis. In the process of formally specifying JSAI, we uncovered several previously unknown soundness bugs in TAJS that were confirmed by the TAJS authors. This serves to highlight the importance and usefulness of formal specification.

[2]Most examples of this approach are intentionally unsound as a design decision, in order to handle the many difficulties raised by JavaScript analysis. Unsound analysis can be useful in some circumstances, but for many purposes (e.g., security auditing) soundness is a key requirement.

Various previous works [34, 70, 72, 85, 109, 134, 135] propose different subsets of the JavaScript language and provide analyses for that subset. These analyses range from type inference, to pointer analysis, to numeric range and kind analysis. None of these handle the full complexities of JavaScript. Several intentionally unsound analyses [8,41,65,110,139] have been proposed, while other works [75,85] take a best-effort approach to soundness, without any assurance that the analysis is actually sound. None of these efforts attempt to formally specify the analysis they implement.

Several type systems [46, 77, 84, 135] have been proposed to retrofit JavaScript (or subsets thereof) with static types. Guha et. al. [77] propose a novel combination of type systems and flow analysis. Chugh et. al. [46] propose a flow-sensitive refinement type system designed to allow typing of common JavaScript idioms. These type systems require programmer annotations and cannot be used as-is on real-world JavaScript programs.

Combinations of static analysis with dynamic checks [47, 72] have also been proposed. These systems statically analyze a subset of JavaScript under certain assumptions and use runtime checks to enforce these assumptions. Schäfer et al. [128] use a dynamic analysis to determine information that can be leveraged to scale static analysis for JavaScript. These ideas are complementary to and can supplement our purely static techniques.

JavaScript Formalisms. None of the previous work on static analysis of JavaScript have formally specified the analysis. However, there has been previous work on providing JavaScript with a formal concrete semantics. Maffeis et. al [111] give a structural smallstep operational semantics directly to the full JavaScript language (omitting a few constructs). Lee et. al [104] propose SAFE, a semantic framework that provides structural bigstep operational semantics to JavaScript, based directly on the ECMAScript specification. Due to their size and complexity, neither of these semantic formulations are suitable for direct translation into an abstract interpreter.

Guha et. al [76] propose a core calculus approach to provide semantics to JavaScript—they provide a desugarer from JavaScript to a core calculus called λ_{JS}, which has a smallstep structural operational semantics. Their intention was to provide a minimal core calculus that would ease proving soundness for type systems, thus placing all the complexity in the desugarer. However, their core calculus is too low-level to perform a precise and scalable static analysis (for example, some of the semantic structure that is critical for a precise analysis is lost, and their desugaring causes a large code bloat—more than $200\times$ on average). We also use the core calculus approach; however, our own intermediate language, called notJS, is designed to be in a sweet-spot that favors static analysis (for example, the code bloat due to our translation is between $6 - 8\times$ on average). In addition, we use an abstract machine-based semantics rather than a

structural semantics, which (as described later) is the prime enabler for configurable analysis sensitivity.

Configurable Sensitivity. Bravenboer and Smaragdakis introduce the DOOP framework [45] that performs flow-insensitive points-to analysis for Java programs using a declarative specification in Datalog. Several context-sensitive versions [98, 132] of the points-to analysis are expressible in this framework as modular variations of a common code base. Their framework would require significant changes to enable flow-sensitive analysis (especially for a language like JavaScript, which requires an extensive analysis to compute a sound SSA form) like ours, and they cannot express arbitrary analysis sensitivities (including path sensitivities) modularly the way that JSAI can.

3.3 JSAI Design

We break our discussion of the JSAI design into three main components: **(1)** the design of an intermediate representation (IR) for JavaScript programs, called notJS, along with its concrete semantics; **(2)** the design of an abstract semantics for notJS that yields the reduced product of a number of essential sub-analyses and also enables configurable analysis; and **(3)** the design of novel abstract domains for JavaScript analysis. We conclude with a discussion of various options for handling dynamic code injection.

The intent of this section is to discuss the design decisions that went into JSAI, rather than giving a comprehensive description of the various formalisms (e.g., the translation from JavaScript to notJS, the concrete semantics of notJS, and the abstract semantics of notJS). All of these formalisms, along with their implementations, are available in the supplementary materials.

3.3.1 Designing the notJS IR

JavaScript's many idiosyncrasies and quirky behaviors motivate the use of formal specifications for both the concrete JavaScript semantics and our abstract analysis semantics. Our approach is to define an intermediate language called notJS, along with a formally-specified translation from JavaScript to notJS. We then give notJS a formal concrete semantics upon which we base our abstract interpreter.[3]

Figure 3.3.1 shows the abstract syntax of notJS, which was carefully designed with the ultimate goal of making abstract interpretation simple, precise, and efficient. The IR contains literal expressions for numeric, boolean values and for **undef** and **null**. Object values are expressed with the **new** construct, and function values are expressed with the **newfun** construct. The IR directly supports exceptions via **throw** and **try-catch-fin**; it supports other non-local control flow (e.g., JavaScript's **return**, **break**, and **continue**) via the **jump** construct. The IR supports two forms of loops: **while** and **for**. The

[3]Guha et al [76] use a similar approach, but our IR design and formal semantics are quite different. See Section 6.7 for a discussion of the differences between our two approaches.

for construct corresponds to JavaScript's reflective `for..in` statement, which allows the programmer to iterate over the fields of an object. A method takes exactly two arguments: `self` and `args`, referring to the `this` object and `arguments` object; all variants of JavaScript method calls can be translated to this form. The **toobj**, **tobool**, **tostr**, **tonum** and **isprim** constructs are the explicit analogues of JavaScript's implicit conversions. JavaScript's builtin objects (e.g,. `Math`) and methods (e.g., `isNaN`) are properties of the global object that is constructed prior to a program's execution, thus they are not a part of the IR syntax.

Note that our intermediate language is *not* based on a control-flow graph but rather on an abstract syntax tree (AST), further distinguishing it from existing JavaScript analyses. JavaScript's higher-order functions, implicit exceptions, and implicit type conversions (that can execute arbitrary user-defined code) make a program's control-flow extremely difficult to precisely characterize without extensive analysis of the very kind we are using the intermediate language to carry out. Other JavaScript analyses that do use a flow-graph approach start by approximating the control-flow and then fill in more control-flow information in an ad-hoc manner as the analysis progresses; this leads to both imprecision and unsoundness (for example, one of the soundness bugs we discovered in TAJS was directly due to this issue). JSAI uses the smallstep abstract machine semantics to determine control-flow during the analysis itself in a sound manner.

$$n \in Num \quad b \in Bool \quad str \in String \quad x \in Variable \quad \ell \in Label$$

$$\begin{aligned}
s \in Stmt ::= {} & \vec{s_i} \mid \textbf{if } e\ s_1\ s_2 \mid \textbf{while } e\ s \mid x := e \\
& \mid e_1.e_2 := e_3 \mid x := e_1(e_2, e_3) \mid x := \textbf{toobj } e \\
& \mid x := \textbf{del } e_1.e_2 \mid x := \textbf{newfun } m\ n \\
& \mid x := \textbf{new } e_1(e_2) \mid \textbf{for } x\, e\, s \mid \textbf{throw } e \\
& \mid \textbf{try-catch-fin } s_1\, x\, s_2\, s_3 \mid \ell\ s \mid \textbf{jump } \ell\ e \\
e \in Exp ::= {} & n \mid b \mid str \mid \textbf{undef} \mid \textbf{null} \\
& \mid x \mid m \mid e_1 \oplus e_2 \mid \odot e \\
d \in Decl ::= {} & \textbf{decl } \overrightarrow{x_i = e_i} \textbf{ in } s \\
m \in Meth ::= {} & (\texttt{self}, \texttt{args}) \Rightarrow d \mid (\texttt{self}, \texttt{args}) \Rightarrow s \\
\oplus \in BinOp ::= {} & + \mid - \mid \times \mid \div \mid \% \mid \ll \mid \gg \mid \ggg \mid < \\
& \mid \leq \mid \ \& \ \mid \ '|' \ \mid \veebar \mid \textbf{and} \mid \textbf{or} \mid +\!\!+ \mid \prec \mid \preceq \\
& \mid \approx \mid \equiv \mid \ . \ \mid \textbf{instanceof} \mid \textbf{in} \\
\odot \in UnOp ::= {} & - \mid \sim \mid \neg \mid \textbf{typeof} \mid \textbf{isprim} \mid \textbf{tobool} \\
& \mid \textbf{tostr} \mid \textbf{tonum}
\end{aligned}$$

Figure 3.1: The abstract syntax of notJS provides canonical constructs that simplify JavaScript's behavior. The vector notation represents (by abuse of notation) an ordered sequence of unspecified length n, where i ranges from 0 to $n - 1$.

An important design decision we made is to carefully separate the language into pure expressions ($e \in Exp$) that are guaranteed to terminate without throwing an exception, and impure statements ($s \in Stmt$) that do not have these guarantees. This decision directly impacts the formal semantics and implementation of notJS by reducing the size of the formal semantics[4] and the corresponding code to one-third of the previous size compared to a version without this separation, and vastly simplifying them. This is the first IR for JavaScript we are aware of that makes this design choice—it is a more radical choice than might first be apparent, because JavaScript's implicit conversions make it difficult to enforce this separation without careful thought. Other design decisions of note include making JavaScript's implicit conversions (which are complex and difficult to reason about, involving multiple steps and alternatives depending on the current state of the program) explicit in notJS (the constructs **toobj**, **isprim**, **tobool**, **tostr**, **tonum** are used for this); leaving certain JavaScript constructs unlowered to allow for a more precise abstract semantics (e.g., the `for..in` loop, which we leave mostly intact as **for** $x\, e\, s$); and simplifying method calls to make the implicit `this` parameter and `arguments` object explicit; `this` is often, but not always, the address of a method's receiver object, and its value can be non-intuitive, while `arguments` provides a form of reflection providing access to a method's arguments.

[4] Specifically, the number of semantic continuations and transition rules.

Given the notJS abstract syntax, we need to design a formal concrete semantics that (together with the translation to notJS) captures JavaScript behavior. We have two main criteria: **(1)** the semantics should be specified in a manner that can be directly converted into an implementation, allowing us to test its behavior against actual JavaScript implementations; **(2)** looking ahead to the abstract version of the semantics (which defines our analysis), the semantics should be specified in a manner that allows for configurable sensitivity. These requirements lead us to specify the notJS semantics as an abstract machine-based smallstep operational semantics. One can think of this semantics as an infinite state transition system, wherein we formally define a notion of *state* and a set of *transition rules* that connect states. The semantics is implemented by turning the state definition into a data structure (e.g., a Scala class) and the transition rules into functions that transform a given state into the next state. The concrete interpreter starts with an initial state (containing the start of the program and all of the builtin JavaScript methods and objects), and continually computes the next state until the program finishes.

We omit further details of the concrete semantics for space and because they have much in common with the abstract semantics described in the next section. The main difference between the two is that the abstract state employs sets in places where the concrete state employs singletons, and the abstract transition rules are nondeterministic

whereas the concrete rules are deterministic. Both of these differences are because the abstract semantics over-approximates the concrete semantics.

Testing the Semantics. We tested the translation to notJS, the notJS semantics, and implementations thereof by comparing the resulting program execution behavior with that of a commercial JavaScript engine, SpiderMonkey [9]. We first manually constructed a test suite of over 243 programs that were either hand-crafted to exercise various parts of the semantics, or taken from existing JavaScript programs used to test commercial JavaScript implementations. We then added over one million randomly generated JavaScript programs to the test suite. We ran all of the programs in the test suite on SpiderMonkey and on our concrete interpreter, and we verified that they produce identical output. Because the ECMA specification is informal we can never completely guarantee that the notJS semantics is equivalent to the spec, but we can do as well as other JavaScript implementations, which also use testing to establish conformance with the ECMA specification.

3.3.2 Designing the Abstract Semantics

The JavaScript static analysis is defined as an abstract semantics for notJS that over-approximates the notJS concrete semantics. The analysis is implemented by computing the set of all abstract states reachable from a given initial state by following the abstract transition rules. The analysis contains some special machinery that provides config-

urable sensitivity. We illustrate our approach via a worklist algorithm that ties these concepts together:

Algorithm 1 The JSAI worklist algorithm

```
1:  put the initial abstract state ς̂0 on the worklist
2:  initialize map partition : Trace → State♯ to empty
3:  repeat
4:      remove an abstract state ς̂ from the worklist
5:      for all abstract states ς̂′ in next_states(ς̂) do
6:          if partition does not contain trace(ς̂′) then
7:              partition(trace(ς̂′)) = ς̂′
8:              put ς̂′ on worklist
9:          else
10:             ς̂old = partition(trace(ς̂′))
11:             ς̂new = ς̂old ⊔ ς̂′
12:             if ς̂new ≠ ς̂old then
13:                 partition(trace(ς̂′)) = ς̂new
14:                 put ς̂new on worklist
15:             end if
16:         end if
17:     end for
18: until worklist is empty
```

The static analysis performed by this worklist algorithm is determined by the definitions of the abstract semantic states $\hat{\varsigma} \in State^\sharp$, the abstract transition rules[5] $\texttt{next\_states} \in State^\sharp \to \mathcal{P}(State^\sharp)$, and the knob that configures the analysis sensitivity $\texttt{trace}(\hat{\varsigma})$.

Abstract Semantic Domains. Figure 3.2 shows our definition of an abstract state for notJS. An abstract state $\hat{\varsigma}$ consists of a *term* that is either a notJS statement or an abstract value that is the result of evaluating a statement; an *environment* that maps variables to (sets of) addresses; a *store* mapping addresses to either abstract values, abstract ob-

[5] Omitted for space; available in supplementary materials.

jects, or sets of continuations (to enforce computability for abstract semantics that use semantic continuations, as per Van Horn and Might [138]); and finally a *continuation stack* that represents the remaining computations to perform—one can think of this component as analogous to a runtime stack that remembers computations that should completed once the current computation is finished.

Abstract values are either exception/jump values ($EValue^\sharp$, $JValue^\sharp$), used to handle non-local control-flow, or base values ($BValue^\sharp$), used to represent JavaScript values. Base values are a tuple of abstract numbers, booleans, strings, addresses, null, and undefined; each of these components is a lattice. Base values are defined as tuples because the analysis over-approximates the concrete semantics, and thus cannot constrain values to be only a single type at a time. These value tuples yield a type inference analysis: any component of this tuple that is a lattice $\perp$ represents a type that this value cannot contain. Base values do not include function closures, because functions in JavaScript are actually objects. Instead, we define a class of abstract objects that correspond to functions and that contain a set of closures that are used when that object is called as a function. We describe our novel abstract object domain in more detail in Section 3.3.3.

Each component of the tuple also represents an individual analysis: the abstract number domain determines a number analysis, the abstract string domain determines a string analysis, the abstract addresses domain determines a pointer analysis, etc. Com-

$$\hat{n} \in Num^\sharp \quad \widehat{str} \in String^\sharp \quad \hat{a} \in Address^\sharp \quad \hat{\odot} \in UnOp^\sharp \quad \hat{\oplus} \in BinOp^\sharp$$

$$\begin{aligned}
\hat{\varsigma} \in State^\sharp &= Term^\sharp \times Env^\sharp \times Store^\sharp \times Kont^\sharp \\
\hat{t} \in Term^\sharp &= Decl + Stmt + Value^\sharp \\
\hat{\rho} \in Env^\sharp &= Variable \to \mathcal{P}(Address^\sharp) \\
\hat{\sigma} \in Store^\sharp &= Address^\sharp \to (BValue^\sharp + Object^\sharp + \mathcal{P}(Kont^\sharp)) \\
\widehat{bv} \in BValue^\sharp &= Num^\sharp \times \mathcal{P}(Bool) \times String^\sharp \times \mathcal{P}(Address^\sharp) \times \\
&\quad \mathcal{P}(\{\textbf{null}\}) \times \mathcal{P}(\{\textbf{undef}\}) \\
\hat{o} \in Object^\sharp &= (String^\sharp \to BValue^\sharp) \times \mathcal{P}(String) \times \\
&\quad (String \to (BValue^\sharp + Class + \mathcal{P}(Closure^\sharp))) \\
c \in Class &= \{\textbf{function}, \textbf{array}, \textbf{string}, \textbf{boolean}, \textbf{number}, \textbf{date}, \\
&\quad \textbf{error}, \textbf{regexp}, \textbf{arguments}, \textbf{object}, \ldots\} \\
\widehat{clo} \in Closure^\sharp &= Env^\sharp \times Meth \\
\widehat{ev} \in EValue^\sharp &::= \textbf{exc}\ bv \\
\widehat{jv} \in JValue^\sharp &::= \textbf{jmp}\ \ell\ \widehat{bv} \\
\hat{v} \in Value^\sharp &= BValue^\sharp + EValue^\sharp + JValue^\sharp \\
\hat{\kappa} \in Kont^\sharp &::= \widehat{\textbf{haltK}} \mid \widehat{\textbf{seqK}}\ \vec{s}_i\ \hat{\kappa} \mid \widehat{\textbf{whileK}}\ e\ s\ \hat{\kappa} \mid \widehat{\textbf{lblK}}\ \ell\ \hat{\kappa} \\
&\quad \mid \widehat{\textbf{forK}}\ \overrightarrow{\widehat{str}}_i\ x\ s\ \hat{\kappa} \mid \widehat{\textbf{retK}}\ x\ \hat{\rho}\ \hat{\kappa}\ \textbf{ctor} \\
&\quad \mid \widehat{\textbf{retK}}\ x\ \hat{\rho}\ \hat{\kappa}\ \textbf{call} \mid \widehat{\textbf{tryK}}\ x\ s\ s\ \hat{\kappa} \mid \widehat{\textbf{catchK}}\ s\ \hat{\kappa} \\
&\quad \mid \widehat{\textbf{finK}}\ \overline{\hat{v}}\ \hat{\kappa} \mid \widehat{\textbf{addrK}}\ \hat{a}
\end{aligned}$$

Figure 3.2: Abstract semantic domains for notJS.

posing the individual analyses represented by the components of the value tuple is not a trivial task; a simple cartesian product of these domains (which corresponds to running each analysis independently, without using information from the other analyses) would be imprecise to the point of being useless. Instead, we specify a reduced product [50] of the individual analyses, which means that we define the semantics so that each individual domain can take advantage of the other domains' information to improve their results. The abstract number and string domains are intentionally unspecified in the semantics; they are configurable. We discuss our specific implementations of the abstract string domain in Section 3.3.3.

Together, all of these abstract domains define a set of simultaneous analyses: control-flow analysis (for each call-site, which methods may be called), pointer analysis (for each object reference, which objects may be accessed), type inference (for each value, can it be a number, a boolean, a string, **null**, **undef**, or a particular class of object), and extended versions of boolean, number, and string constant propagation (for each boolean, number and string value, is it a known constant value). These analyses combine to give detailed control- and data-flow information forming a fundamental analysis that can be used by many possible clients (e.g., error detection, program slicing, secure information flow, etc).

Abstract Transition Rules. Figure 3.3 describes a small subset of the abstract transition rules to give their flavor. To compute `next_states`$(\hat{\varsigma})$, the components of $\hat{\varsigma}$ are

matched against the premises of the rules to find which rule(s) are relevant; that rule then describes the next state (if multiple rules apply, then there will be multiple next states). The rules $1, 2$ and 3 deal with sequences of statements. Rule 1 says that if the state's term is a sequence, then pick the first statement in the sequence to be the next state's term; then take the rest of the sequence and put it in a **seqK** continuation for the next state, pushing it on top of the continuation stack. Rule 2 says that if the state's term is a base value (and hence we have completed the evaluation of a statement), take the next statement from the **seqK** continuation and make it the term for the next state. Rule 3 says that if there are no more statements in the sequence, pop the **seqK** continuation off of the continuation stack. The rules 4 and 5 deal with conditionals. Rule 4 says that if the guard expression evaluates to an abstract value that over-approximates **true**, make the **true** branch statement the term for the next state; rule 5 is similar except it takes the **false** branch. Note that these rules are nondeterministic, in that the same state can match both rules.

Configurable Sensitivity. To enable configurable sensitivity, we build on the insights of Hardekopf et al [82]. We extend the abstract state to include an additional component from a *Trace* abstract domain. The worklist algorithm uses the `trace` function to map each abstract state to its trace, and joins together all reachable abstract states that map to the same trace (see lines 10–11 of Algorithm 4). The definition of *Trace* is left to the analysis designer; different definitions yield different sensitivities. For example,

	Current State $\hat{\varsigma}$	Next State $\hat{\varsigma}'$	
1	$\langle s :: \vec{s}_i, \hat{\rho}, \hat{\sigma}, \hat{\kappa} \rangle$	$\langle s, \hat{\rho}, \hat{\sigma}, \widehat{\textbf{seqK}}\ \vec{s}_i\ \hat{\kappa} \rangle$	
2	$\langle \widehat{bv}, \hat{\rho}, \hat{\sigma}, \widehat{\textbf{seqK}}\ s :: \vec{s}_i\ \hat{\kappa} \rangle$	$\langle s, \hat{\rho}, \hat{\sigma}, \widehat{\textbf{seqK}}\ \vec{s}_i\ \hat{\kappa} \rangle$	
3	$\langle \widehat{bv}, \hat{\rho}, \hat{\sigma}, \widehat{\textbf{seqK}}\ \epsilon\ \hat{\kappa} \rangle$	$\langle \widehat{bv}, \hat{\rho}, \hat{\sigma}, \hat{\kappa} \rangle$	
4	$\langle \textbf{if}\ e\ s_1\ s_2, \hat{\rho}, \hat{\sigma}, \hat{\kappa} \rangle$	$\langle s_1, \hat{\rho}, \hat{\sigma}, \hat{\kappa} \rangle$	*if* **true** $\in \pi_{\hat{b}}([\![e]\!])$
5	$\langle \textbf{if}\ e\ s_1\ s_2, \hat{\rho}, \hat{\sigma}, \hat{\kappa} \rangle$	$\langle s_2, \hat{\rho}, \hat{\sigma}, \hat{\kappa} \rangle$	*if* **false** $\in \pi_{\hat{b}}([\![e]\!])$

Figure 3.3: A small subset of the abstract semantics rules for JSAI. Each smallstep rule describes a transition relation from one abstract state ς to the next state $\hat{\varsigma}'$. The phrase $\pi_{\hat{b}}([\![e]\!])$ means to evaluate expression e to an abstract base value, then project out its boolean component.

suppose Trace is defined as the set of program points, and an individual state's trace is the current program point. Then our worklist algorithm computes a flow-sensitive, context-insensitive analysis: all states at the same program point are joined together, yielding one state per program point. Suppose we redefine Trace to be sequences of program points, and an individual state's trace to be the last k call-sites. Then our worklist algorithm computes a flow-sensitive, k-CFA context-sensitive analysis. Arbitrary sensitivities (including path-sensitivity and property simulation) can be defined in this manner solely by redefining Trace, without affecting the worklist algorithm or the abstract transition rules. We explore a number of possibilities in Section 3.5.

3.3.3 Novel Abstract Domains

JSAI allows configurable abstract number and string domains, but we also provide default domains based on our experience with JavaScript analysis. We motivate and describe our default abstract string domain here. We also describe our novel abstract object domain, which is an integral part of the JSAI abstract semantics.

Abstract Strings. Our initial abstract string domain $\mathit{String}^\sharp$ was an extended string constant domain. The elements were either constant strings, or strings that are definitely numbers, or strings that are definitely not numbers, or $\top$ (a completely unknown string). This string domain is similar to the one used by TAJS [89], and it is motivated by the precision gained while analyzing arrays: arrays are just objects where array indices are represented with numeric string properties such as `"0"`, `"1"`, etc, but they also have non-numeric properties like `"length"`. However, this initial string domain was inadequate.

In particular, we discovered a need to express that a string is *not* contained within a given hard-coded set of strings. Consider the property lookup `x := obj[y]`, where `y` is a variable that resolves to an unknown string. Because the string is unknown, the analysis is forced to assign to `x` not only the lattice join of all values contained in `obj`, but also the lattice join of all the values contained in all prototypes of `obj`, due to the rules of prototype-based inheritance. Almost all object prototype chains termi-

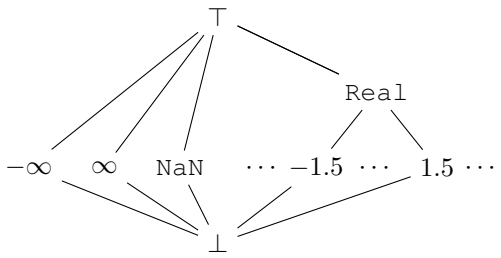

Figure 3.4: Our default number abstract domain, further explained in Section 3.3.3.

nate in one of the builtin objects contained in the global object (`Object.prototype`, `Array.prototype`, etc); these builtin objects contain the builtin values and methods. Thus, all of these builtin values and methods are returned for any object property access based on an unknown string, polluting the analysis. One possible way to mitigate this problem is to use an expensive domain that can express arbitrary complements (i.e., express that a string is *not* contained in some arbitrary set of strings). Instead, we extend the string domain to separate out *special* strings (`valueOf`, `toString` etc, fixed ahead of time) from the rest; these special strings are drawn from property names of builtin values and methods. We can thus express that a string has an unknown value that is *not* one of the special values. This is a practical solution that improves precision at minimal cost.

The new abstract string domain depicted in Figure 3.5 (that separates unknown strings into numeric, non-numeric and special strings) was simple to implement due to JSAI's configurable architecture; it did not require changes to any other parts of the implementation despite the pervasive use of strings in all aspects of JavaScript semantics.

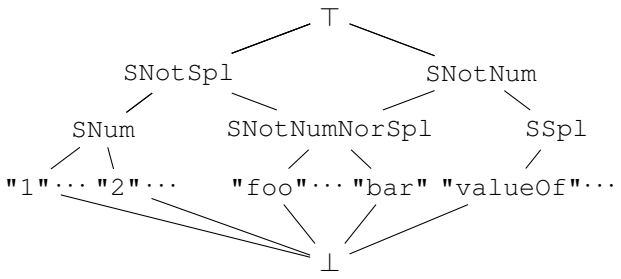

Figure 3.5: Our default string abstract domain, further explained in Section 3.3.3.

Abstract Objects. We highlight the abstract domain $Object^\sharp$ given in Figure 3.2 as a novel contribution. Previous JavaScript analyses model abstract objects as a tuple containing **(1)** a map from property names to values; and **(2)** a list of definitely present properties (necessary because property names are just strings, and objects can be modified using unknown strings as property names). However, according to the ECMA standard objects can be of different *classes*, such as functions, arrays, dates, regexps, etc. While these are all objects and share many similarities, there are semantic differences between objects of different classes. For example, the `length` property of array objects has semantic significance: assigning a value to `length` can implicitly add or delete properties to the array object, and certain values cannot be assigned to `length` without raising a runtime exception. Non-array objects can also have a `length` field, but assigning to that field will have no other effect. The object's class dictates the semantics of property enumerate, update, and delete operations on an object. Thus, the analysis must track what classes an abstract object may belong to in order to accurately

model these semantic differences. If abstract objects can belong to arbitrary sets of classes, this tracking and modeling becomes complex, error-prone, and inefficient.

Our innovation is to add a map as the third component of abstract objects that contains class-specific values. This component also records which class an abstract object belongs to. Finally, the semantics is designed so that any given abstract object must belong to exactly one class. This is enforced by assigning abstract addresses to objects based not just on their static allocation site and context, but also on the constructor used to create the object (which determines its class). The resulting abstract semantics is much simpler, more efficient, and precise.

3.4 Showcasing Configurability

Analysis *sensitivity* (path-, context-, and heap-sensitivity) hsa a significant impact on the usefulness and practicality of the analysis. The sensitivity represents a tradeoff between precision and performance: the more sensitive the analysis is the more precise it can be, but also the more costly it can be. The "sweet-spot" in this tradeoff varies from analysis to analysis and from program to program. JSAI allows the user to easily specify different sensitivities in a modular way, separately from the rest of the analysis.

A particularly important dimension of sensitivity is *context-sensitivity*: how the (potentially infinite) possible method call instances are partitioned and merged into a

finite number of abstract instances. The current state of the art for JavaScript static analysis has explored only a few possible context-sensitivity strategies, all of which are baked into the analysis and difficult to change, with no real basis for choosing these over other possible strategies.

We take advantage of JSAI's configurability to define and evaluate a much larger selection of context-sensitivities than has ever been evaluated before in a single paper. Because of JSAI's design, specifying each sensitivity takes only 5–20 lines of code; previous analysis implementations would have to hard-code each sensitivity from scratch. The JSAI analysis designer specifies a sensitivity by instantiating a particular instance of Trace; all abstract states with the same trace will be merged together. For context-sensitivity, we define Trace to include some notion of the calling context, so that states in the same context are merged while states in different contexts are kept separate.

We implement six main context-sensitivity strategies, each parameterized in various ways, yielding a total of 56 different forms of context-sensitivity. All of our sensitivities are flow-sensitive (JavaScript's dynamic nature means that flow-insensitive analyses tend to have terrible precision). We empirically evaluate all of these strategies in Section 3.5; here we define the six main strategies. Four of the six strategies are known in the literature, while two are novel to this work. The novel strategies are based on two hypotheses about context definitions that might provide a good balance between precision and performance. Our empirical evaluation demonstrates that these hypothe-

ses are false, i.e., they do not provide any substantial benefit. We include them here not as examples of good sensitivities to use, but rather to demonstrate that JSAI makes it easy to formulate and test hypotheses about analysis strategies—each novel strategy took only 15–20 minutes to implement. The strategies we defined are as follows, where the first four are known and the last two are novel:

Context-insensitive. All calls to a given method are merged. We define the context component of Trace to be a unit value, so that all contexts are the same.

Stack-CFA. Contexts are distinguished by the list of call-sites on the call-stack. This strategy is k-limited to ensure there are only a finite number of possible contexts. We define the Trace component to contain the top k call-sites.

Acyclic-CFA. Contexts are distinguished the same as Stack-CFA, but instead of k-limiting we collapse recursive call cycles. We define Trace to contain all call-sites on the call-stack, except that cycles are collapsed.

Object-sensitive. Contexts are distinguished by a list of addresses corresponding to the chain of receiver objects (corresponding to full-object-sensitivity in Smaragdakis et al. [132]). We define Trace to contain this information (k-limited to ensure finite contexts).

Signature-CFA. Type information is important for dynamically typed languages, so intuitively it seems that type information would make good contexts. We hypothesize

that defining $Trace$ to record the types of a call's arguments would be a good context-sensitivity, so that all k-limited call paths with the same types of arguments would be merged.

Mixed-CFA. Object-sensitivity uses the address of the receiver object. However, in JavaScript the receiver object is often the global object created at the beginning of the program execution. Intuitively, it seems this would mean that object sensitivity might merge many calls that should be kept separate. We hypothesized that it might be beneficial to define $Trace$ as a modified object-sensitive strategy—when object-sensitivity would use the address of the global object, this strategy uses the current call-site instead.

3.5 Evaluation

In this section we evaluate JSAI's precision and performance for a range of context-sensitivities as described in Section 3.4, for a total of 56 distinct sensitivities. We run each sensitivity on 28 benchmarks collected from four different application domains and analyze the results, yielding surprising observations about context-sensitivity and JavaScript. We also briefly evaluate JSAI as compared to TAJS [89], the most comparable existing JavaScript analysis.

3.5.1 Implementation and Methodology

We implement JSAI using Scala version 2.10. We provide a model for the DOM, event handling loop (handled as non-deterministic execution of event-handling functions), and other native APIs used in our benchmarks. The baseline analysis sensitivity we evaluate is **fs** (flow-sensitive, context-insensitive); all of the other evaluated sensitivities are more precise than **fs**. The other sensitivities are: $k.h$-**stack**, h-**acyclic**, $k.h$-**obj**, $k.h$-**sig**, and $k.h$-**mixed**, where k is the context depth for k-limiting and h is the heap-sensitivity (i.e., the context depth used to distinguish abstract addresses). The parameters k and h vary from 1 to 5 and $h \leq k$.

We use a comprehensive benchmark suite to evaluate the sensitivities. Most prior work on JavaScript static analysis has been evaluated only on the standard SunSpider [10] and V8 [11] benchmarks, with a few micro-benchmarks thrown in. We evaluate JSAI on these standard benchmarks, but we also include real-world representatives from a diverse set of JavaScript application domains. We choose seven representative programs from each domain, for a total of 28 programs. We partition the programs into four categories, described below. For each category, we provide the mean size of the benchmarks in the suite (expressed as number of AST nodes generated by the Rhino parser [1]) and the mean translator blowup (i.e., the factor by which the number of AST nodes increases when translating from JavaScript to notJS). The benchmark

names are shown in the graphs presented below; the benchmark suite is included in the supplementary material.

The benchmark categories are: **standard**: seven of the largest, most complex benchmarks from SunSpider [10] and V8 [11] (*mean size: 2858 nodes; mean blowup: 8×*); **addon**: seven Firefox browser addons selected from the official Mozilla addon repository [2] (*mean size: 2597 nodes; mean blowup: 6×*); **generated**: seven programs from the Emscripten LLVM test suite, which translates LLVM bitcode to JavaScript [7] (*mean size: 38211 nodes; mean blowup: 7×*); and finally **opensrc**: seven real-world JavaScript programs taken from open source JavaScript frameworks and their test suites [12, 13] (*mean size: 8784 nodes; mean blowup: 6.4×*).

Our goal is to evaluate the precision and performance of JSAI instantiated with several forms of context sensitivity. However, the different sensitivities yield differing sets of function contexts and abstract addresses, making a fair comparison difficult. Therefore, rather than statistical measurements (such as address-set size or closure-set size), we choose a *client-based* precision metric based on a error reporting client. This metric is a proxy for the precision of the analysis.

Our precision metric reports the number of static program locations (i.e., AST nodes) that might throw exceptions, based on the analysis' ability to precisely track types. JavaScript throws a TypeError exception when a program attempts to call a non-function or when a program tries to access, update, or delete a property of **null** or

undef. JavaScript throws a RangeError exception when a program attempts to update the length property of an array to contain a value that is not an unsigned 32-bit integer. Fewer errors indicate a more precise analysis.

The performance metric we use is execution time of the analysis. To gather data on execution time, we run each experimental configuration 11 times, discard the first result, then report the median of the remaining 10 trials. We set a time limit of 30 minutes for each run, reporting a timeout if execution time exceeds that threshold. We run all experiments on Amazon Web Services [14] (AWS), using M1 XLarge instances; each experiment is run on an independent AWS instance. These instances have 15GB memory and 8 ECUs, where each ECU is equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

We run all 56 analyses on each of the 28 benchmarks, for a total of 1,568 trials (multiplied by an additional 10 executions for each analysis/benchmark pair for the timing data). For reasons of space, we present only highlights of these results. In some cases, we present illustrative examples; the omitted results show similar behavior. In other cases, we deliberately cherry-pick, to highlight contrasts. We are explicit about our approach in each case.

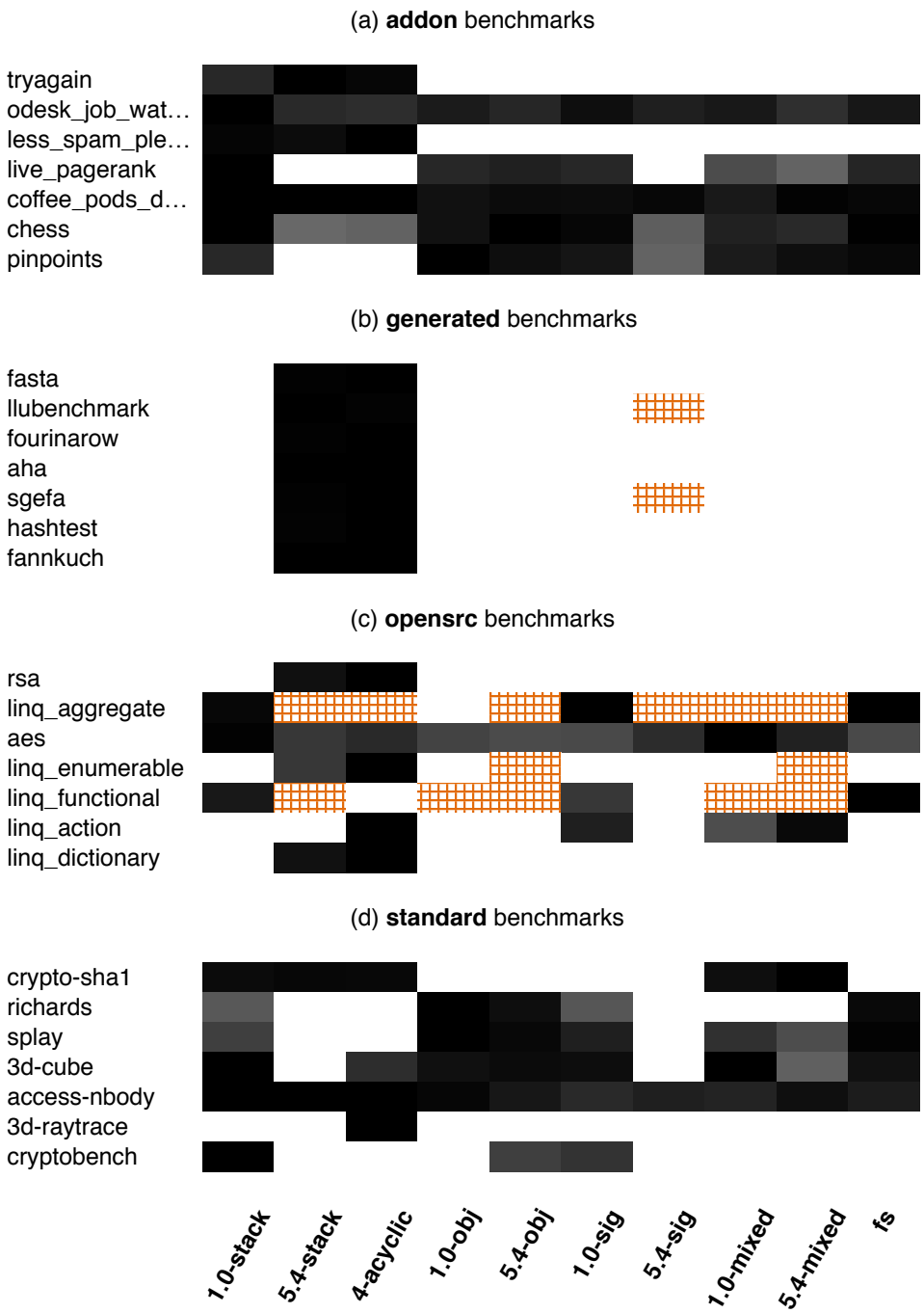

Figure 3.6: A heat map to showcase the performance characteristics of different sensitivities across the benchmark categories. For more details on how to read this map, please refer to the corresponding prose.

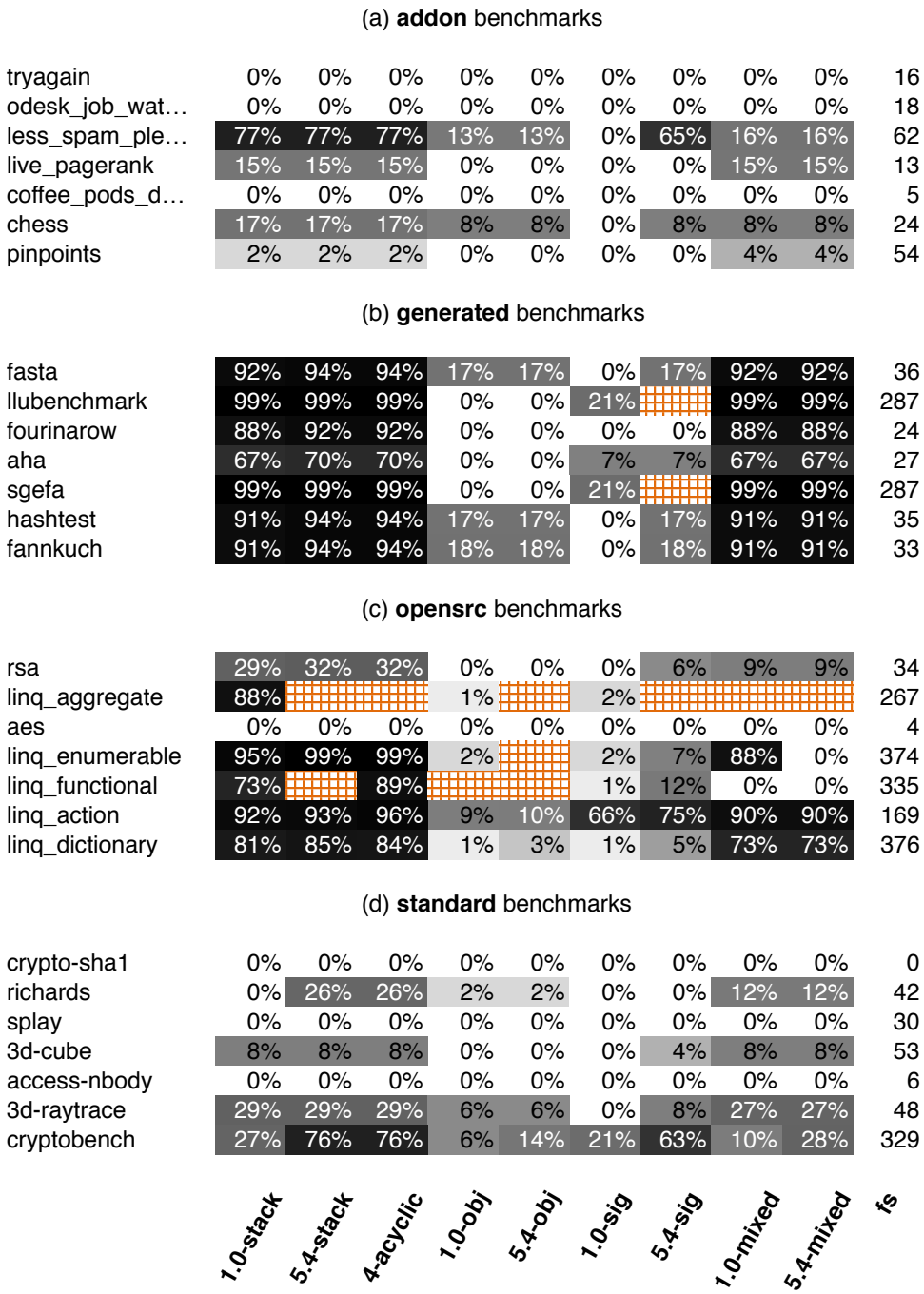

Figure 3.7: A heat map to showcase the precision characteristics (based on number of reported runtime errors) of different sensitivities across the benchmark categories. For more details on reading the heap please refer to the corresponding prose.

3.5.2 Observations

For each main sensitivity strategy, we present the data for two trials: the least precise sensitivity in that strategy, and the most precise sensitivity in that strategy. This set of analyses is: **fs**, **1.0-stack**, **5.4-stack**, **4-acyclic**, **1.0-obj**, **5.4-obj**, **1.0-sig**, **5.4-sig**, **1.0-mixed**, **5.4-mixed**.

Figure 3.6 contains performance results, and Figure 3.7 contains the precision results. The results are partitioned by benchmark category to show the effect of each analysis sensitivity on benchmarks in that category.

Figure 3.6 is setup to easily depict how the sensitivities perform relative to each other. Figure 3.6 is heat map that lays out blocks in two dimensions—rows represent benchmarks and columns represent analyses with different sensitivities. Each block represents relative performance as a color: darker blocks correspond to faster execution time of a sensitivity compared to other sensitivities on the same benchmark. A completely blackened block corresponds to the fastest sensitivity on that benchmark, a whitened block corresponds to a sensitivity that has $\geq 2\times$ slowdown relative to the fastest sensitivity, and the remaining colors evenly correspond to slowdowns in between. Blocks with the red grid pattern indicate a timeout. A visual cue is that columns with darker blocks correspond to better-performing sensitivities, and a row with blocks that have very similar colors indicates a benchmark on which performance is unaffected by varying sensitivities.

Figure 3.7 provides a similar heat map (with similar visual cues) for visualizing relative precisions of various sensitivity strategies on our benchmarks. The final column in this heat map provides the number of errors reported by the **fs** strategy on a particular benchmark, while the rest of the columns provide the percentage reduction (relative to **fs**) in the number of reported errors due to a corresponding sensitivity strategy. The various blocks (except the ones in the final column) are color coded in addition to providing percentage reduction numbers: darker is better precision (that is, more reduction in number of reported errors). Timeouts are indicated using a red grid pattern.

Breaking the Glass Ceiling. One startling observation is that highly sensitive variants (i.e., sensitivity strategies with high k and h parameters) can be far better than their less-sensitive counterparts, providing improved precision at a much cheaper cost (see Figure 3.8). For example, on `linq_dictionary`, **5.4-stack** is the most precise *and* most efficient analysis. By contrast, the **3.2-stack** analysis yields the same result at a three-fold increase in cost, while the **1.0-stack** analysis is even more expensive and less precise. We see similar behavior for the `sgefa` benchmark, where **5.4-stack** is an order of magnitude faster than **1.0-stack** and delivers the same results. This behavior violates the common wisdom that values of k and h above 1 or 2 are intractably expensive.

This behavior is certainly not universal,[6] but it is intriguing. Analysis designers often try to scale up their context-sensitivity (in terms of k and h) linearly, and they

[6]For example, `linq_aggregate` times out on all analyses with $k > 1$.

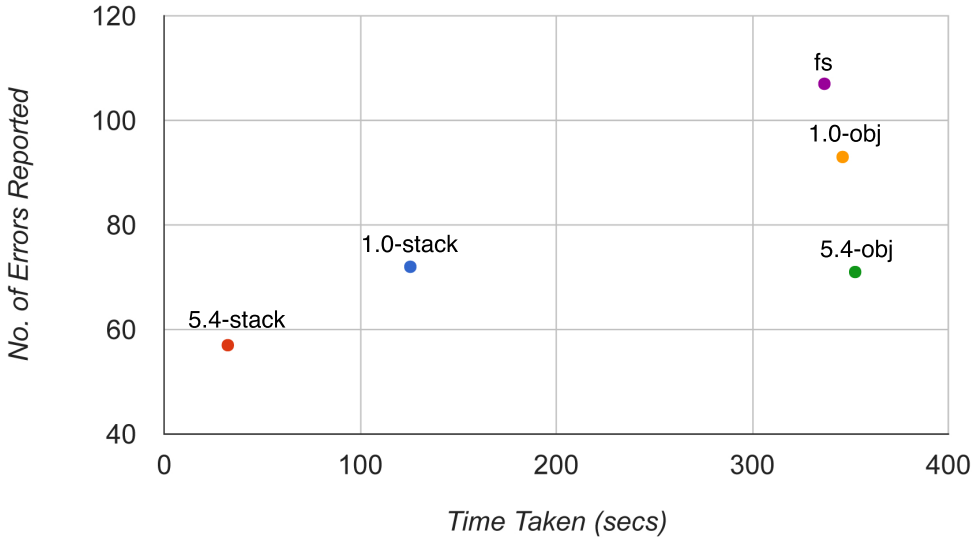

Figure 3.8: Precision *vs.* performance of various sensitivities, on the **opensrc** benchmark `linq_dictionary`. Interestingly, **5.4-stack** (the most sensitive Stack-CFA analysis) is not only tractable, it exhibits the best performance and the best precision.

stop when it becomes intractable. However, our experiments suggest that pushing past this local barrier may yield much better results.

Callstring vs Object Sensitivity. In general, we find that callstring-based sensitivity (i.e., k.h**-stack** and h**-acyclic**) is more precise than object sensitivity (i.e., k.h**-obj**). This result is unintuitive, since JavaScript heavily relies on objects and object sensitivity was specifically designed for object-oriented languages such as Java. Throughout the benchmarks, the most precise and efficient analyses are the ones that employ stack-based k-CFA. Part of the reason for this trend is that 25% of the benchmarks are machine-generated JavaScript versions of procedural code, whose structure yields more

benefits to callstring-based context-sensitivity. Even among the handwritten open-source benchmarks, however, this trend holds. For example, several forms of call-string sensitivity are more efficient and provide more precise results for the open-source benchmarks than object-sensitivity, which often times out.

Benefits of Context Sensitivity. When it comes to pure precision, we find that more context sensitivity sometimes increases precision and sometimes has no effect. The open-source benchmarks demonstrate quite a bit of variance for the precision metric. A context-sensitive analysis almost always finds fewer errors (i.e., fewer false positives) than a context-insensitive analysis, and increasing the sensitivity in a particular family leads to precision gains. For example, **5.4-stack** gives the most precise error report for `linq_enumerable`, and it is an order of magnitude more precise than a context-insensitive analysis. On the other hand, the addon domain has very little variance for the precision metric, which is perhaps due to shorter call sequence lengths in this domain. In such domains, it might be wise to focus on performance, rather than increasing precision.

Summary. Perhaps the most sweeping claim we can make from the data is that there is no clear winner across all benchmarks, in terms of JavaScript context-sensitivity. This state of affairs is not a surprise: the application domains for JavaScript are so rich and varied that finding a silver bullet for precision and performance is unlikely. However,

it is likely that—within an application domain, e.g., automatically generated JavaScript code—one form of context-sensitivity could emerge a clear winner. The benefit of JSAI is that it is easy to experiment with the control flow sensitivity of an analysis. The base analysis has already been specified, the analysis designer need only instantiate and evaluate multiple instances of the analysis in a modular way to tune analysis-sensitivity, without having to worry about the analysis soundness.

3.5.3 Discussion: JSAI *vs.* TAJS

Jensen et al.'s Type Analysis for JavaScript [89, 90] (TAJS) stands as the only published static analysis for JavaScript whose intention is to soundly analyze the entire JavaScript language. JSAI has several features that TAJS does not, including configurable sensitivity, a formalized abstract semantics, and novel abstract domains, but TAJS is a valuable contribution that has been put to good use. An interesting question is how JSAI compares to TAJS in terms of precision and performance.

The TAJS implementation (in Java) has matured over a period of five years, it has been heavily optimized, and it is publicly available. Ideally, we could directly compare TAJS to JSAI with respect to precision and performance, but they are dissimilar enough that they are effectively noncomparable. For one, TAJS has known soundness bugs that can artificially decrease its set of reported type errors. Also, TAJS does not implement some of the APIs required by our benchmark suite, and so it can only run on a subset of

the benchmarks. On the flip side, TAJS is more mature than JSAI, it has a more precise implementation of the core JavaScript APIs, and it contains a number of precision and performance optimizations (e.g., the recency heap abstraction [37] and lazy propagation [90]) that JSAI does not currently implement.

Nevertheless, we can perform a qualitative "ballpark" comparison, to demonstrate that JSAI is roughly comparable in terms of precision and performance. For the subset of our benchmarks on which both JSAI and TAJS execute, we catalogue the number of errors that each tool reports and record the time it took for each tool to do so. We find that JSAI analysis time is $0.3\times$ to $1.8\times$ that of TAJS. In terms of precision, JSAI reports from nine fewer type errors to 104 more type errors, compared to TAJS. Many of the extra type errors that JSAI reports are RangeErrors, which TAJS does not report due to one of the unsoundness bugs we uncovered. Excluding RangeErrors, JSAI reports at most 20 more errors than TAJS in the worst case.

3.6 Conclusion

We have described the design of JSAI, a configurable, sound, and efficient abstract interpreter for JavaScript. JSAI's design is novel in a number of respects which make it stand out from all previous JavaScript analyzers. We have provided a comprehensive evaluation that demonstrates JSAI's usefulness. The JSAI implementation and

formalisms are freely available as a supplement, and we believe that JSAI will provide a useful platform for people building JavaScript analyses.

Our future work includes (1) taking advantage of JSAI's configurability to further investigate what control-flow sensitivities are most useful for JavaScript; (2) writing a number of clients on top of JSAI, including program refactoring, program compression; and (3) extending JSAI to handle language features from the latest ECMA 5 standard.

Chapter 4

Improving Precision of JavaScript Static Analysis via Type Refinement

4.1 Introduction

Dynamic languages have become ubiquitous. For example, Java-Script is used to implement a large amount of critical online infrastructure, including web applications, browser addons/extensions, and interpreters such as Adobe Flash. In response to the growing prominence of dynamic languages, the research community has begun to investigate how to apply *static analysis* techniques in this domain. Static analysis is used to deduce properties of a program's execution behavior; these properties can be used for a variety of useful purposes including optimization [79, 109], error checking [144], verification [46], security auditing [73, 74], and program refactoring [64], among other uses. However, dynamic languages present a unique challenge to static analysis, inher-

ent in their very name: the dynamic nature of these languages makes creating precise, sound, and efficient static analyses a daunting task.

In this chapter we focus on the static analysis of JavaScript, though in principle our proposed techniques are applicable to other dynamic languages as well. Our work is complementary to other recent work on JavaScript analysis, which has focused on understanding a program's types by proposing various novel abstract domains to track type information [89, 109]. This focus on types is essential for JavaScript analysis; because JavaScript behavior relies heavily on the runtime types of the values being operated on, understanding types is a necessary prerequisite to understanding many other properties of program behavior. However, with one exception (discussed further in Section 6.7) this prior work on JavaScript analysis has ignored an observation that has been profitably exploited in more traditional static analyses: that branch conditions (i.e., predicates that determine a program's control flow) necessarily constrain the set of values that can flow into the corresponding branches. This observation can be used to *refine* the abstract information propagated by the static analysis within each branch, thus improving the precision of the analysis. The details of how this concept works and how it can be applied to improve the precision of static analysis are explained in Appendix 6.2 (for any analysis in general) and Section 4.2 (for JavaScript analysis specifically).

While this general observation is well-known in the static analysis community, applying it specifically to JavaScript raises several important questions that must be answered to gain any useful benefit: (1) what kinds of conditions provide the most useful information for refinement; (2) how prevalent are these kinds of conditions in realistic JavaScript programs; and (3) how can we best exploit these conditions, based on their prevalence and usefulness, to substantially increase the precision of static analysis?

4.1.1 Key Insight

Our key insight that informs our proposed technique is that the most prevalent and useful conditional branches are not *explicit* in the text of JavaScript program, i.e., these conditions do not show up syntactically as **if** or **while** statements. Rather, they are *implicit* in the JavaScript semantics themselves. As an example, consider the statement `var result = myString.length;`. While syntactically there are no conditional branches in this statement, during execution there are several conditional branches taken by the JavaScript interpreter:

- Is `myString` either **null** or **undefined**? If so then raise a type-error exception, otherwise continue execution.

- Is `myString` a primitive value or an object? If it's a primitive value then convert it to an object first, then access the `length` property; otherwise just access the `length` property.

- Does the object (or one of its prototypes) contain a `length` property? If so then return the corresponding value, otherwise return **undefined**.

Our thesis is that JavaScript static analysis can take advantage of these implicit conditional executions to refine the type information of the abstract values being propagated by the analysis, and that this *type refinement* can provide significant improvement in analysis precision.

4.1.2 Contributions

Our specific contributions are:

- A definition of type refinement for static analysis of JavaScript, including several variations that use different kinds of conditions to refine types (Section 4.2).

- An empirical evaluation of the proposed type refinement variations (Section 6.6). This evaluation is carried out on a more comprehensive set of JavaScript benchmarks than any presented in previous literature on JavaScript static analysis; it includes not only the standard SunSpider and V8 benchmark suites, but also a

number of open-source JavaScript applications [13,26] and a number of machine-generated JavaScript programs created using Emscripten [7].

- A set of recommendations for including type refinement in JavaScript analyses (Section 4.6). Our evaluation shows that taking advantage of implicit conditional branches provides a critical precision advantage for finding type errors, while the explicit `typeof` conditional branches exploited in previous work [78] provide only marginal benefit.

We conclude that type refinement is a promising technique for JavaScript analysis. This technique's design is informed by the semantics of JavaScript, enabling it to take advantage of language features hidden from plain sight and thus gain precision that would be lost by a technique that does not specifically exploit JavaScript semantics. Furthermore, type refinement is orthogonal to the question of designing abstract domains for JavaScript analysis; this means that it can profitably be combined with interesting new abstract domains in the future to achieve even better results.

4.2 The Potential for Refinement in JavaScript

Refinement allows an analysis to safely replace a less-precise answer with a more-precise answer. Appendix 6.2 gives suitable background on static analysis and the concept of refinement; readers unfamiliar with these notions may wish to refer to that

appendix before continuing. Refinement can apply to many different abstract domains for analysis, but we hypothesize that, for JavaScript, the abstract domain of *types* is a particularly fruitful target for refinement. In JavaScript, as with many dynamic languages, the type of a value strongly influences the behavior of a program. Thus, refining type information intuitively would seem likely to improve the precision of JavaScript static analysis (and our empirical results bear out this intuition).

This observation means that we should focus our attention on those conditionals in the JavaScript program that are based on type information, i.e., conditionals whose truth or falsity constrain the set of types allowed in the corresponding branches. An obvious candidate is the set of conditionals that use the **typeof**operator to test value's types. For example, consider the following code:

```
if ( typeof x == "number" ) { x = x + 42; }
```

Suppose that immediately before the conditional, the static analysis has computed that `x` may be a number or a string. Then inside the true branch of the conditional, we can safely refine the type of `x` to be a number. This strategy is similar to the one employed by Guha et al [78] (discussed further in Section 6.7), though they were attempting to typecheck a subset of JavaScript rather than to improve the precision of JavaScript static analysis.

4.2.1 Key Insight

While the **typeof** check is an obvious candidate for refinement, our key insight is that most of the conditionals involving types aren't even syntactically present in the JavaScript program—rather, they are implicit in the semantics of the JavaScript language itself.

Consider the following statement:

```
var x = myString[i];
```

This seemingly simple statement requires a large number of implicit type checks. Example 2 makes all of these checks explicit. None of these checks involve **typeof**. Instead, we see three new kinds of conditions that involve type information.

One condition (e.g., at line `1` in Example 2) checks whether a value is either `null` or `undefined`. JavaScript performs this check whenever a program attempts to access a property of a value; if the value is `null` or `undefined` it is a type-error. JavaScript also performs this check whenever a program attempts to add, update, or delete a property of some value. We abbreviate this condition as **isUndefNull**.

Another condition (e.g., at lines `5`, `12`, `22`, and `31`) checks whether a value is primitive rather than an object, i.e., that it is either a number, a boolean, a string, `undefined`, or `null`[1]. JavaScript performs this check whenever the runtime might need to implicitly convert a value into another type. We abbreviate this condition as **isPrim**.

[1]Confusingly, **typeof** `null` == "object", but `null` is not an object.

Example 2 The semantics of var x = myString[i];

```
 1: if myString is null or undefined then
 2:     type-error

 3: else
 4:     // convert myString to an object first?
 5:     if myString is a primitive then
 6:         obj = toObject(myString)
 7:     else
 8:         obj = myString
 9:     end if

10:     // convert i to a string

11:     // case 1: i is a primitive
12:     if i is a primitive then
13:         prop = toString(i)
14:     else
15:         if i.toString is callable then
16:             tmp = i.toString()
17:         else
18:             goto line 26
19:         end if
20:     end if

21:     // case 2: i is not a primitive, but i.toString() is
22:     if tmp is a primitive then
23:         prop = toString(tmp)

24:     // case 3: i.toString() is not a primitive; try i.valueOf()
25:     else
26:         if i.valueOf is callable then
27:             tmp2 = i.valueOf()
28:         else
29:             type-error
30:         end if

31:         if tmp2 is a primitive then
32:             prop = toString(tmp2)
33:         else
34:             type-error
35:         end if
36:     end if

37:     // retrieve the property from the object
38:     x = obj.prop
39: end if
```

A third condition (e.g., at lines `15` and `26`) checks whether a value is callable (i.e., that it is actually a function). If so, then the runtime calls the function; otherwise it can throw a type error exception. We abbreviate this check for callable as the **isFunc** condition.

The key insight of this work is to focus refinement on those *implicit conditionals*—**isPrim**, **isUndefNull**, and **isFunc**—which abound in JavaScript programs.

4.2.2 Refinement on Implicit Conditions

JavaScript's implicit conditions restrict the types of values that flow along their branches. Refinement can take advantage of these restrictions as follows:

- **isPrim**: On the true branch, the checked value *must* be a primitive value; on the false branch it *must* be an object.
- **isUndefNull**: On the true branch, the checked value *must* be either `undefined` or `null`; on the false branch it *cannot* be `undefined` or `null`.
- **isFunc**: On the true branch, the checked value *must* be a function; on the false branch it *cannot* be a function.

The benefits become evident when we consider a static analysis that does *not* use refinement for these conditions. For **isPrim** the benefit comes from the false branch of the conditional, for example, line `15` in Example 2. Suppose that on line `12` the

analysis computes that `i` may be either `undefined` or an object. In the false branch, `i`'s properties are accessed to make method calls (e.g., the `.toString` and/or `.valueOf` methods used to convert objects to primitives). However, since `i` may be `undefined`, the analysis conservatively computes that these calls may raise a type error exception. If `i` had been refined, then the analysis would know that it cannot be `undefined` on that branch, and hence there cannot be a type error exception.

The benefit for **isUndefNull** and **isFunc** is more subtle. Consider the following program fragment:

```
delete obj.p1;
obj.p2 = 2;
```

Example 3 The semantics of `delete obj.p1; obj.p2 = 2;`

```
1: if obj is null or undefined then
2:     type-error
3: else
4:     delete obj.p1
5:     if obj is null or undefined then
6:         type-error
7:     else
8:         obj.p2 = 2
9:     end if
10: end if
```

The implicit behavior of this fragment is described by the pseudocode in Example 3.This example illustrates how implicit checks and exceptions can lead to spurious type-errors. Concretely, the code performs two sequential property modifications. If

`obj` is `null` or `undefined`, the first statement causes a type-error, and the second statement never executes. Otherwise, both statements execute successfully. An analysis that uses refinement can capture this behavior, while an analysis that does not use refinement cannot, as explained below.

Consider an analysis of this fragment that has imprecise information: `obj` might be `null` or an object. In this case, the **isUndefNull** conditions in lines `1` and `5` of Example 3 are non-deterministic, and the analysis must conservatively propgate `obj`'s type to both branches. An analysis that does not use refinement must then conservatively report that two type-errors might occur: at lines `2` and `6`. In reality, if the first statement of the program fragment successfully executes, so will the second. Refinement can detect this invariant: in the false branch of lines `4`–`9`, the type of `obj` cannot be `null` or `undefined`. The **isUndefNull** condition at line `5` is therefore deterministic, so the analysis will not follow the branch to the error at line `6`. Thus, a refined analysis can give the most precise result for imprecise data: if a type-error occurs, it occurs only as a result of the `delete` statement.

The **isFunc** check is similar to the **isUndefNull** check in that the cost comes from potentially passing unrefined values to successor nodes, causing the analysis to conservatively compute type error exceptions whereas an analysis using refinement would not. In general, the benefit of refinement in the presence of implicit exceptions is potentially tremendous: When exploring the path along which the exception does not occur,

the analysis can refine the type of the value so that it does not cause any more implicit exceptions along that path. Our empirical evaluation demonstrates that, if an analysis focuses on these simple implicit type checks, refinement can dramatically increase the precision of a type-error analysis.

4.3 Refining Types in JavaScript Analyses

The previous section discusses type refinement at a conceptual level. In this section we make the discussion concrete, describing specifically how we perform type refinement for JavaScript. Type refinement takes place in the context of some particular static analysis, however type refinement itself is largely independent of that surrounding context. Therefore we describe type refinement using a generic type-based abstract domain that would be common to any JavaScript static analysis, which in the actual analysis can be augmented to provide whatever additional information is relevant.

4.3.1 Type-based Abstract Domain

We now describe the abstract domain that we will be using to describe type refinement. A JavaScript value can be a primitive value (i.e., `number`, `boolean`, `string`, `undefined`, or `null`), an object, or a function[2]. The abstract domain of Figure 4.1

[2]Functions are also objects, but we distinguish them separately because some implicit checks are specific to functions.

$$\begin{aligned} \sigma \in Store &: Variable \to \mathcal{P}(Type) \\ PropertyMap &: Property \to \mathcal{P}(Type) \end{aligned}$$

$$\begin{aligned} \tau \in Type &: PrimType + ObjType + FuncType \\ PrimType &: \textbf{num} + \textbf{bool} + +\textbf{null} + \textbf{undefined} \\ ObjType &: PropertyMap \\ FuncType &: Closure \times PropertyMap \end{aligned}$$

Figure 4.1: A simplified version of an abstract domain suitable for type refinement. The abstract domain $Store$ maps variables to their abstract types. The abstract domain $ObjType$ maps object properties to a set of possible types. The abstract domain $FuncType$ includes a closure (the function to be called) and a property map (to model the function object).

describes an approximation of these types. This abstract domain is deliberately simpler than one that would be used in an actual analysis, in order to make the exposition more clear by focusing on the aspects relevant to type refinement. A specific static analysis would augment this abstract domain with more information relevant to the purpose of that analysis (for examples of such augmented abstract domains, see [89, 109]).

The abstract domain in the figure represents the relevant type information that is propagated by the analysis from program point to program point. An abstract store $Store$ maps variable names to sets of abstract types. We uses sets of types because, as discussed in Section 6.2, the analysis is approximating the concrete program behavior—e.g., the analysis may be able to determine that a variable is either **num** or **undefined**, but not be able to narrow the type information down any further. An abstract object

$$x \in \mathit{Variable} \qquad p \in \mathit{Property}$$

$$\begin{aligned} a \in \mathit{Access} &::= x \mid x.p \\ c \in \mathit{Condition} &::= \textbf{typeof}(a) = \mathit{tag} \mid \textbf{isFunc}(a) \\ &\quad \mid \textbf{isUndefNull}(a) \mid \textbf{isPrim}(a) \\ \mathit{tag} &::= \textbf{"number"} \mid \textbf{"boolean"} \mid \textbf{"string"} \\ &\quad \mid \textbf{"undefined"} \mid \textbf{"object"} \mid \textbf{"function"} \end{aligned}$$

Figure 4.2: Type-based conditions for refinement. These conditions precisely describe the conditonal expressions that trigger refinement. An *access* is a low-level primitive—the simplest form of a variable or property access. Our analysis can handle any conditional expression that reduces to this form.

type $\mathit{ObjType}$ maps property names to their abstract types. An abstract function type $\mathit{FuncType}$ consists of a *closure* (the function to be called) and a property map (to model the fact that JavaScript functions are also objects).

A type-based static analysis operates over this abstract domain. Abstract stores flow along the program's control-flow graph (where each node is a program statement), and at each statement the analysis interprets the effect of that statement relative to a specific input σ flowing from that statement's predecessors, in order to determine the new σ' that is the output of that statement, which is then passed to that statement's successors. If the analysis encounters a type-based condition, the analysis may be able to increase the precision of the information contained in the store by refining the type information based on the condition, as described below.

4.3.2 Identifying Relevant Type-based Conditions

When using type refinement, the analysis interprets a branch condition as a *filter* along each branch of the conditional; these filters are used to refine the type information of the stores passed to each respective branch. In theory, *any* branch condition that constrains the types contained in the store can be used to perform type refinement. However, in practice some conditions are much more complicated to translate into filters than others. Therefore the analysis designer must make a tradeoff, by syntactically restricting the set of conditions from which the analysis extracts filters. The goal is to balance the additional precision that may be gained by interpreting certain conditions against the complexity of generating filters from those conditions.

Figure 4.2 shows the tradeoff that we have made in our type refinement implementation. The figure gives a restricted syntax for branch conditions (which also makes certain implicit checks explicit in the syntax rather than implicit in the language semantics). Our analysis only attempts refinement using conditions that are contained in this restricted syntax; any other conditions are treated the same as if the analysis were not doing type refinement. We chose this syntax to match the categories of explicit and implicit type checks from Section 4.2. The **typeof**condition corresponds to implicit and explicit checks on type equality. The **isFunc** condition corresponds to the implicit check for whether a value is a function. The **isUndefNull** condition corresponds to the implicit check that the JavaScript interpreter performs when accessing or modifying

an object property. The **isPrim** condition corresponds to the implicit check that the JavaScript interpreter performs as part of implicit type conversion.

Each condition contains exactly one *access*. An access can have two forms: a variable or a *direct* property access. A direct property access $x.p$ gives the precise property name being accessed. The program need not literally contain a direct access; the program might specify the property access using a complex expression. As long as the static analysis can recover the direct access from the expression, the analysis will attempt to apply refinement. In practice, for **typeof**conditions, our analysis handles any conditions of the form **typeof**e_1 == e_2, where e_1 and e_2 are arbitrary JavaScript expressions. Our analysis currently does not handle more complex expressions than these. In particular, it does not handle logical combinations of these conditions. In Section 6.6, we demonstrate that the conditions in Figure 4.2 are sufficient to achieve significant increases in the precision of a type-based analysis; creating useful filters for more complicated expressions is left to future work.

4.3.3 Filtering Type Information

Each condition induces a *filter* that captures the types described by that condition. Figure 4.3 defines the filter for each possible condition. These filters match the description of the explicit and implicit information encoded in the scenarios from Section 4.2.

$c \in Condition$	$filter(c)$
typeof$(a) =$ **"number"**	{**num**}
typeof$(a) =$ **"boolean"**	{**bool**}
typeof$(a) =$ **"string"**	{**str**}
typeof$(a) =$ **"undefined"**	{**undefined**}
typeof$(a) =$ **"object"**	{**null**} $\cup$ $ObjType$
typeof$(a) =$ **"function"**	$FuncType$
isFunc(a)	$FuncType$
isUndefNull(a)	{**undefined**, **null**}
isPrim(a)	$\{\tau_p \in PrimType\}$

Figure 4.3: Filters for refinement conditions. The analysis uses these filters to refine information along a condition's branches.

The analysis uses a condition's filters to refine the values that flow along the condition's branches. Specifically, for each type-based condition c:

1. The analysis interprets condition c relative to an input store σ, to determine which branches (i.e, the true and false branches) to execute.

2. The analysis uses the input store σ to retrieve the abstract type τ of the condition's access a.

3. When the analysis executes the true branch, it computes a new abstract type for a as follows: $\tau \cap filter(c)$. In other words, it intersects the current set of possible types for a with the set of possible types for a allowed by the branch condition. The analysis updates the type for a in σ and sends the updated store along the true branch.

4. When the analysis executes the false branch, it computes a new abstract type for a as follows: $\tau - \mathit{filter}(c)$. In other words, it removes from the current set of possible types for a those types which would have meant that the branch condition was true. The analysis updates the type for a in σ and sends the updated store along the false branch.

For example, suppose the analysis reaches a condition **isUndefNull**(a) with a store that maps a to the abstract type $\{$**undefined**, **num**$\}$. In this case the condition evaluates to both **true** and **false**, and so the analysis must execute both the true and false branches. Along the true branch, the analysis sends a store that assigns a the type $\{$**undefined**, **num**$\} \cap \mathit{filter}($**isUndefNull**$(a)) = \{$**undefined**$\}$. Along the false branch, the analysis sends a store that assigns a the type $\{$**undefined**, **num**$\} - \mathit{filter}($**isUndefNull**$(.)) = \{$**num**$\}$.

4.3.4 Sound Type Refinement

Type refinement is *sound* if and only if the filtered set of types sent to a branch is a superset of all types that might ever be seen at that branch over all possible concrete executions. Our refinement rules are sound as long as they are only applied to accesses that correspond to a single concrete access. This is the standard static analysis issue of *strong* vs *weak* updates: a strong update can replace a value with a completely new value (potentially more precise than the previous value), while a weak update can only

replace a value with a weaker (i.e., less precise) value. This issue is best explained by example.

Suppose that a variable x is mapped by the abstract store to the set of types $\{[\text{foo} \mapsto \{\textbf{num}, \textbf{str}\}], [\text{foo} \mapsto \{\textbf{num}, \textbf{bool}\}]\}$. This abstract value means that the type of x may be one or the other of the two object types, but the analysis does not know which one. Now suppose that the analysis is considering a branch condition $\textbf{typeof}(x.\text{foo}) = \textbf{"number"}$. In the true branch, the type of $x.\text{foo}$ must be **num**. However, the analysis does not know *which* of the two possible object types for x is correct, so it cannot determine which of the two object types has been constrained by the condition. Thus, the analysis cannot refine either object type because if it refines the wrong one, the analysis becomes unsound. When this is the case, any update to the abstract value for x must be weak: an analysis cannot replace the abstract value of x with a more precise version.

If, on the other hand, x refers to only a single possible object type, e.g., $\{[\text{foo} \mapsto \{\textbf{num}, \textbf{str}\}]\}$, then an analysis knows that this is the type constrained by the condition. Thus, the analysis can safely refine x's value in the true branch to $\{[\text{foo} \mapsto \{\textbf{num}\}]\}$. When this is the case, an update to the abstract value for x can be strong: the analysis *can* replace x's abstract value with a more precise value. Our analysis applies type refinement—which overwrites abstract values—only when the refinement corresponds to a strong update.

4.4 Evaluation

We have implemented our proposed ideas and evaluated their effect on a static analysis for JavaScript that detects potential type-error exceptions. We find that an analysis that performs type refinement on all of the conditions described in Section 4.3.2 can achieve a significant increase in analysis precision, with a minimal impact on the analysis performance. In this section we demonstrate the effectiveness of our ideas by comparing the type-error analysis *with* type refinement relative to the same analysis *without* type refinement, for a variety of JavaScript programs.

4.4.1 JavaScript Analysis Framework

We use the JSAI JavaScript static analyzer for our experiments. The source code for JSAI can be found at `http://www.cs.ucsb.edu/~pllab` under Downloads. JSAI is implemented in Scala version 2.10.1 using the Rhino parser as a front-end [1]. JSAI does not currently handle `eval` and related mechanisms for dynamic code injection, however none of our benchmarks use these mechanisms.

4.4.2 Benchmark Suite

JavaScript can be written in a number of different styles, and these styles can affect the usefulness of our type refinement technique. In order to explore this issue, we select

benchmarks from a variety of application domains. We group the selected bencharks into three categories:

- **standard**: These are the standard benchmark suites, SunSpider [**?**] and Octane [27], that are used by browser vendors to test the correctness and performance of their JavaScript implementations.

- **opensrc**: These are real-world, handwritten JavaScript programs taken from various open source projects such as LINQ for JavaScript [26] and Defensive JS [13].

- **emscripten**: These are machine-generated JavaScript code, obtained by compiling C/C++ programs using the Emscripten [7] LLVM→JavaScript compiler.

We select seven benchmarks from each category for our evaluation. This benchmark suite is available for download[3]. The benchmarks exercise a wide range of JavaScript features, including core objects and APIs, typed arrays, etc. However, none of these benchmarks contain `eval` or equivalent features that allow dynamic code injection.

Figure 4.4 shows the distribution of program sizes in each benchmark category, based on the number of AST nodes created for the programs by the Rhino parser. We use AST nodes as the metric for program size because it correlates with the amount of work done by the analysis, which operates over AST nodes. The standard benchmarks, while not large, exercise several key features of the language, and we use them to test

[3]Available with the rest of the repository under Downloads at `http://www.cs.ucsb.edu/~pllab`.

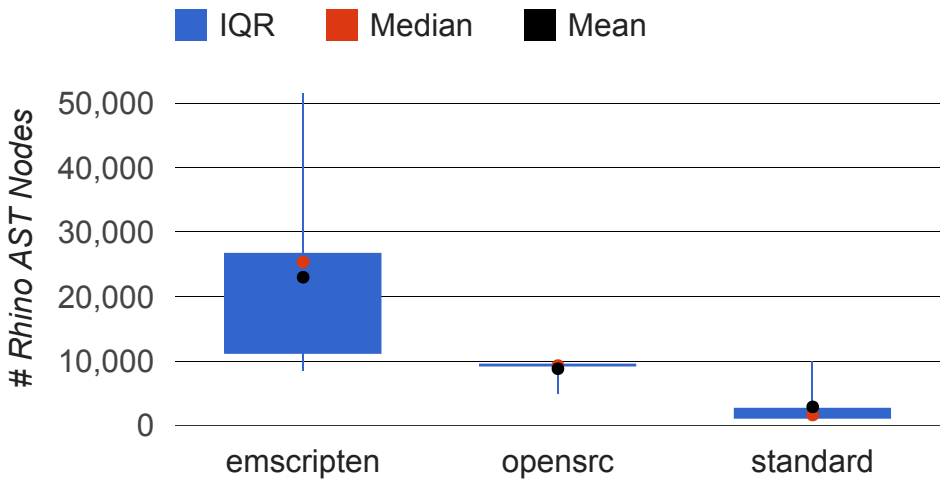

Figure 4.4: Graph to show size distribution (along y-axis) of benchmarks in each category (x-axis). Size is measured in terms of number of JavaScript AST nodes created by the Rhino parser [1]. For each benchmark category, the blue box gives the 25%-75% quartiles, the blue line gives the range of sizes, and median and mean are denoted by red and black dots respectively.

the correctness of our implementation. The opensrc and emscripten benchmarks are significantly larger, however it should be noted that the emscripten benchmarks contain a large amount of unreachable code because the Emscripten compiler automatically includes a large amount of unused library code.

4.4.3 Experimental Methodology

Our base analysis is flow-sensitive and context-sensitive, using a stack-based 1-CFA context-sensitivity strategy (i.e., it distinguishes function contexts by the callsite from which the function was invoked). The heap model uses static allocation sites to

model abstract addresses. Starting from this base type-error analysis, we implement new analyses that incrementally add support for refining various kinds of conditions. We implement and evaluate a total of four type-error analyses:

- **B**: a base flow-sensitive, context-sensitive type-error analysis that does not perform refinement.
- **T**: the **B** analysis, extended with type refinement for conditionals that contain **typeof**checks.
- **TP**: the **T** analysis, extended to include type refinement for conditionals that contain **isPrim** checks.
- **TPUNF**: the **TP** analysis extended to include type refinement for conditionals that contain **isUndefNull** and **isFunc** checks.

We compare the precision and performance of these analyses for all the benchmarks in our suite. The metric we use to measure precision is the number of program locations (i.e., AST nodes) that the analysis computes may potentially throw type-error exceptions. The analysis that reports the fewest locations is the most precise. This metric correlates with the usefulness of a static type-error reporting tool: although false positives are inherent in a static analysis, the fewer the number of reported errors (i.e., the fewer the false positives) the more useful the tool is.

The metric we use to measure performance is execution time in seconds. We perform a trial for each (analysis, benchmark) pair. Each trial runs in its own invocation of the JVM. A trial starts with a warm-up run whose results are discarded. We then perform 10 runs in sequence and report the mean execution time of these 10 runs. All our experiments execute on an Ubuntu 12.04.2 LTS machine with CPU speed of 1.9GHz and 32GB RAM on JVM version 1.7.

4.4.4 Potential Opportunity for Type Refinement

In this section, we explore the potential benefits of type refinement across our benchmark categories. Type refinement is potentially useful for a given branch condition when the analysis treats that condition as non-deterministic—i.e., the analysis cannot determine for certain which branch is taken, and so must execute both branches. To gain an understanding of how many opportunities various flavors of type refinement can take advantage of in these benchmarks, we distinguish three kinds of branches:

- **T**: Branches with a **typeof**check in them.
- **P**: Branches with a **isPrim** check in them.
- **UNF**: Branches with a **isUndefNull** or **isFunc** check in them.

We also qualify each kind of branch to be:

- **D**: a deterministic branch.
- **NDC**: a non-deterministic branch where the branch condition follows our restricted syntax for type refinement, and therefore is a candidate for our type refinement.
- **NDNC**: a non-deterministic branch where the branch condition does not match our restricted syntax for type refinement, and therefore is not a candidate for our type refinement.

Deterministic branches **D** provide no opportunity for type refinement at all. Non-deterministic, non-candidate branches **NDNC** could potentially benefit from type refinement if we extended our technique to include more complicated branches, but do not benefit from our current type refinement implementation. The non-deterministic candidate branches **NDC** are the branches that can benefit from our current implementation of type refinement.

We provide the above information about each qualified kind of branch for each benchmark category, using the **B** version of the analysis, and summarize the data in Table 4.1. The data shows that the deterministic branches far exceed the non-deterministic ones. This might seem surprising, but the reason is because the analysis must perform a number of checks that are almost always trivially true. For example, in the code `a[0] = 0`, the analysis checks that `a` is not `undefined` or `null`, and in a vast major-

category	branch kind	D	NDC	NDNC
standard	**T**	469	87	104
	P	2408	141	0
	UNF	5692	571	0
opensrc	**T**	408	82	50
	P	2048	80	0
	UNF	9456	374	0
emscripten	**T**	149	14	9
	P	595	3	0
	UNF	7120	12	0

Table 4.1: The table that shows for each category of benchmarks, the kind of branches that the analysis encounters. The numbers represent number of program locations. The abbreviations are further detailed in Section 4.4.4. The way to interpret this table is as follows: for example, the number under column **NDC**, and row **T** represents the number of program locations with branches that have **typeof** checks in them, and are non-deterministic and match our grammar for type refinement.

ity of cases this is true, making this conditional deterministic. This is particularly true of the emscripten benchmarks, which makes sense because they were generated from statically-typed languages. Although most branch points are deterministic, there are still a significant number of non-deterministic branches that can be exploited by type refinement.

4.4.5 Effects of Various Type Refinements

Table 4.2 presents the raw data of our evaluation, including the total number of type errors reported for each benchmark and the mean runtimes for the **B** and **TPUNF**

analyses. Figure 4.5 extracts and summarizes the precision results from Table 4.2. For each benchmark, we provide the number of type-errors reported by the analysis under the four configurations (**B**, **T**, **TP** and **TPUNF**—columns 2–5), the percentage reduction in number of type errors reported when run with **TPUNF** version over the **B** version (column 6), the mean runtime in seconds when run with **B** and **TPUNF** versions (columns 7 and 8, respectively), and the percentage increase in time taken of **TPUNF** version over **B** version (column 9). We also summarize for each benchmark category, the total number of error reports across benchmarks in that category for each version of the analysis, and the mean runtime for running the analysis under **B** and **TPUNF** versions across the benchmarks in that category. The mean performance data has a relative standard deviation of at most 30%.

For the standard and opensrc benchmarks, the **T** configuration yields almost no benefit over the **B** configuration, meaning that doing type refinement over this kind of condition is not useful for reducing type-error exceptions. However, the **TP** configuration does yield a fair amount of benefit, and the **TPUNF** configuration yields significant benefit. For example, on the `cryptobench` benchmark in the standard category the **TPUNF** configuration reports 253 fewer type errors than the **B**, and for `rsa` in opensrc category, 124 fewer errors are reported by **TPUNF** configuration when compared to **B**. In general, across benchmarks in the standard and opensrc categories, most of

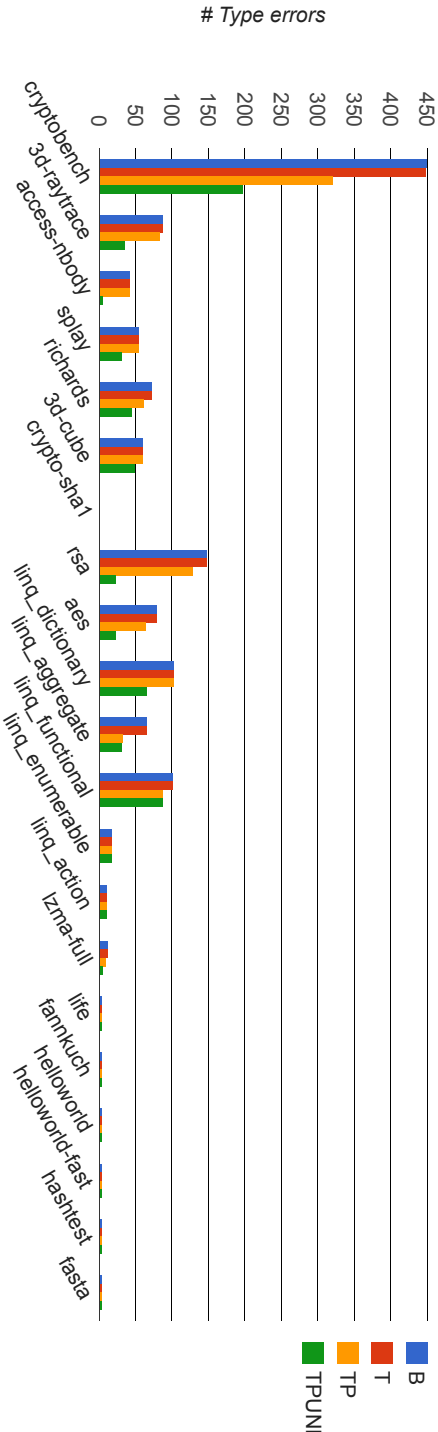

Figure 4.5: Analysis precision (in number of reported type-errors) with and without refinement; lower is better. Benchmarks are grouped by category. Under the **TPUNF** analysis with refinement, many of the standard benchmarks (`cryptobench` to `crypto-sha1`) and the opensrc benchmarks (`rsa` to `linq_action`) achieve more than 50% improvement in precision, relative to the **B** analysis without refinement.

the benefits arise from type refinement for **isPrim**, **isUndefNull** and **isFunc** implicit checks.

For the emscripten benchmarks, type refinement does not make a significant impact on precision—this is expected based on the low potential available in these benchmarks for type refinement (see Figure 4.1), and because there are very few type-errors reported in the **B** version already. Because emscripten benchmarks are generated by compiling from a statically-typed language, the values in the program tend to be very monomorphic.

From Table 4.2, specifically the column giving the percentage increase in time, we observe that type refinements have a negligible effect on performance.

4.5 Related Work

Static analysis of dynamic languages is an active research area. Recent innovations include: a type analysis that relies on an abstract domain that is highly tuned for JavaScript [89]; various static type inference algorithms [34, 69] and hybrid type inference algorithms [33], including those that prevent access to undefined fields [144], enable program optimizations [79, 109], or are suitable for IDEs [124]; static analyses to secure the Web [72–74]; alias analyses for JavaScript [85, 133] and Python [71]; an analysis to support JavaScript refactoring [64]; and analysis frameworks for dy-

Benchmark	Number of TypeErrors				Reduction in	Mean Runtime (s)		Slowdown
	B	T	TP	TPUNF	TypeErrors	B	TPUNF	
cryptobench	450	448	321	197	56%	47.46	46.30	−2.4%
3d–raytrace	87	87	84	35	60%	3.92	4.07	4%
access–nbody	43	43	43	6	86%	0.41	0.41	0%
splay	55	55	55	31	44%	0.78	0.78	0%
richards	73	73	62	45	38%	2.23	2.44	9%
3d–cube	60	60	60	49	18%	3.77	3.72	−1%
crypto–sha1	0	0	0	0	0%	0.30	0.29	−3%
standard	**768**	**766**	**625**	**363**	**53%**	**8.41**	**8.29**	**−1.4%**
rsa	148	148	129	24	84%	10.73	10.46	−3%
aes	79	79	64	23	71%	0.82	0.81	−1%
linq_dictionary	102	102	102	66	35%	81.00	79.35	−2%
linq_aggregate	66	66	33	31	53%	308.29	314.23	2%
linq_functional	101	101	88	87	14%	530.94	536.41	1%
linq_enumerable	17	17	17	17	0%	85.55	84.49	−1%
linq_action	11	11	11	11	0%	21.61	21.99	2%
opensrc	**524**	**524**	**444**	**259**	**51%**	**148.42**	**149.68**	**−1%**
lzma–full	12	12	9	5	58%	0.39	0.39	0%
life	4	4	4	4	0%	1.53	1.61	5%
fannkuch	4	4	4	4	0%	1.97	2.01	2%
helloworld	4	4	4	4	0%	1.46	1.47	1%
helloworld–fast	4	4	4	4	0%	1.44	1.38	−4%
hashtest	4	4	4	4	0%	3.66	3.73	2%
fasta	4	4	4	4	0%	3.21	3.50	9%
emscripten	**36**	**36**	**33**	**29**	**19%**	**1.95**	**2.01**	**3%**

Table 4.2: Table summarizing the precision and performance benefits of various type refinement optimizations.

namic languages [28, 104]. All these techniques are orthogonal to type refinement. As such, they all may benefit from the idea of refinement in general, and the analyses for JavaScript would benefit directly from our contribution.

Two existing techniques for type inference of dynamic languages rely in particular on a notion of refinement: flow typing [78] and occurrence typing [136, 137]. Flow typing is a technique for JavaScript type inference that uses type tags in explicit `typeof` conditionals to filter type information [78]. Occurrence typing is another technique for refining types in branches based on the conditionals that govern those branches [136, 137]. Occurrence typing takes into consideration a more complex set of filters than flow typing and type refinement, including the effects of selectors (e.g., `car`). These filters are encoded in a propositional logic that forms the basis of a type system for Scheme.

Our work, type refinement, leverages a similar insight as does flow typing and occurrence typing, namely that runtime types in a dynamically typed language are constrained by branch conditions. Perhaps the most important distinction between our work and prior work is the focus on *which* branch conditions are mined by the analysis to obtain more precise information. Flow and occurrence typing focuses on branches that occur in common coding idioms: explicit behavior, encoded by a programmer, that follows a particular pattern. The reasoning behind this focus is that the programmers are communicating information by using an idiom, and automated understanding can

take advantage of this high-level semantic information. Type refinement focuses on implicit behavior driven by the runtime: behavior encoded not by the programmer, but by the semantics of the language itself. The reasoning behind this focus is that implicit behavior (i.e., the semantics of the language) appears in *every* program, and if the analysis can take advantage of this behavior the potential benefit can be huge. In the context of type errors, our study shows that a focus on the implicit JavaScript behavior provides more benefit that the focus on only explicit JavaScript behavior.

Type refinement also differs from prior work in its goals and methods. The goal of flow and occurrence typing is a sound static type system for a dynamic language. Hence, Guha et al. and Tobin-Hochstadt et al. focus on type soundness (although Guha et al. also validate their work against a corpus of Scheme code). In contrast, the goal for our work is to provide more precise information for JavaScript static analyses, which we validate by performing an evaluation of several variants against a large corpus of JavaScript programs. Operationally, our work differs from prior work in that prior work encodes the extra type information in the abstract domain, explicitly representing the information and essentially delaying the refinement. Ours work filters the results along a path, essentially applying the refinement immediately.

Type refinement should not be confused with *refinement types* [46, 68]. Although they have similar names and share other superficial similarities, they are different concepts. Type refinement is an action that occurs during program analysis: it filters the

analysis values that flow along paths in the program's abstract execution. An analysis with type refinement is more precise than one without it. A refinement type is an entity, placed in a program by its author to express a restriction on the set of values that can be computed by a particular expression. A type system with refinement types can prove stronger properties about programs than one without them.

4.6 Conclusion and Future Work

We show in this work that type refinement is a useful precision optimization for static analysis of JavaScript. In particular, type refinement for implicit conditionals in JavaScript can have a significant impact on precision (upto 86% for a static type-error client). We also show that type refinement does not cause any adverse performance impact.

We can do type refinements only if the refinement can be a strong update, thus, we can increase the precision due to type refinement by implementing orthogonal techniques that can increase the number of strong updates. We are currently exploring recency abstraction [37, 84], which is a technique that can increase the number of strong updates in the analysis. It would be interesting to study the effect of combining type refinement with recency abstraction.

We would also like to extend our type refinement to more complicate conditionals, and explore how far we can increase precision benefits without affecting adversely performance.

Chapter 5

Improving Performance of Static Analysis via Parallelization

5.1 Introduction

JavaScript is prevalent on a wide variety of platforms, including the web, mobile phones, desktops, and servers. Static analysis for JavaScript is a necessity to help build developer tools to construct and review secure, fast, maintainable, and correct JavaScript code. In order to be useful, such JavaScript analyses need to be precise and to run within a reasonable amount of time. However, JavaScript's inherently dynamic nature makes precise static analysis very expensive. As an anecdotal example, we have observed a particular 2,800 line JavaScript program[1] on which a sequential JavaScript analysis that computes data and control dependencies takes over 22 hours to complete. While in the early days of its introduction JavaScript programs tended to be small, sim-

[1] `linq_aggregate` with `stack-5-4`

ple scripts, today there are many complex JavaScript applications with tens to hundreds of thousands of lines of code. Thus, there is a need to increase JavaScript analysis performance while maintaining high levels of precision.

A heretofore unexplored option is to *parallelize* the JavaScript analysis, thus exploiting the prevalence of modern multicore architectures. The idea of parallel program analysis is not novel; there are many existing parallel program analysis frameworks [30, 61, 63, 103, 105, 113, 114, 121, 140]. However, most of these efforts are aimed at first-order, statically-typed, highly imperative languages such as C or Fortran; JavaScript presents new challenges that must be addressed. Our novelty lies not only in the first parallel JavaScript static analysis, but also in the approach with which we design our parallel analysis, which potentially could benefit parallel analysis of other languages.

Key Insight. We focus on parallel analyses that are flow- and context-sensitive, because we need a high level of precision to successfully analyze JavaScript. Almost all such precise parallel analyses in existing work are based on traditional dataflow analysis (DFA) [93, 100]. Our key insight is that the DFA framework inextricably mixes decisions about synchronization and granularity with the definition of the analysis itself, thus limiting opportunities to fully exploit possible parallelism in such analyses. We identify an alternate approach to program analysis more amenable to paralleliza-

tion, based on ideas from abstract interpretation. Using this approach, we can phrase an analysis as two separate and independent components:

- An abstract semantics that represents the static analysis as a state transition system (STS). The analysis is defined as a reachable-states computation on the STS: given a program and its initial state, the analysis finds all abstract program states potentially reachable from that initial state. This reachable-states computation is embarrassingly parallel in nature, because each state is inherently independent from all other states.

- A separate mechanism for selectively merging multiple abstract states into a single abstract state by over-approximating the information in the states being merged together. This merging takes place during the reachability computation and is used to bound the reachable state space in a sound manner. This mechanism effectively merges branches of the computation tree formed by the STS, turning it into a DAG and thus adding sequential behavior (and synchronization points) into the otherwise parallel reachable-states computation.

From this perspective the static analysis itself is trivially a massively parallel problem; this parallelism is then limited by a strategy that determines how and when states are merged (introducing synchronization into the analysis). Using this approach, opportunities for parallelizing the analysis become more obvious than the previous DFA-

based approaches. In fact, existing approaches can be re-defined as limited instances of our framework. While it is possible to derive this overall insight purely from a DFA standpoint, it is not possible to act upon it within the DFA framework because DFA intertwines and conflates the two above-described components in an inseparable way.

Our new perspective provides a useful framework for designing parallel analyses, but there is still a large design space to be explored. The strategy for merging states controls the level of synchronization required by the analysis, as well as the size of the state space being explored; thus it has a strong impact on parallelism. In addition, while the normal reachable-states computation is embarassingly parallel, that does not necessarily mean that taking full advantage of its inherent parallelism is the best course—there are many different possible levels of granularity which may provide performance benefits and tradeoffs.

In essence, in our approach the problem of parallelizing an analysis boils down to two decisions: determining a strategy for merging states, and selecting a particular level of granularity at which to operate. In this work we explore several such design points, discussing their rationales and implications. We include a novel parallelization strategy based on function contexts.

Contributions. The specific contributions of this work are the following:

1. A new perspective on the design of parallel program analyses, based on formulating the analysis as a state transition system plus a separate state merging strategy. (Section 5.3.1)

2. A language-agnostic exploration of the design space of this parallelization framework, including a novel parallelization strategy based on function contexts. (Section 5.3.2)

3. Our implementation of these ideas for JSAI, an abstract interpreter for JavaScript that computes a fundamental analysis for JavaScript—performing a combination of type inference, alias analysis, control-flow analysis, and string, numeric, and boolean value analysis. (Section 5.4)

4. An evaluation of our resulting parallel JavaScript abstract interpreter. Speedups are typically in the 2-4$\times$ range on 12 hardware threads, ranging as high as 36.9$\times$. (Section 6.6)

5. A publicly available implementation[2].

[2]In the `Downloads` section of `http://cs.ucsb.edu/~pllab`

5.2 Background and Related Work

In this section we provide a brief background on sequential dataflow analysis (DFA) and describe related work on parallelizing program analysis, much of which is based on DFA.

5.2.1 Sequential Dataflow Analysis

DFA-based analysis is carried out on the program's *control-flow graph* (CFG), which is a directed graph $G = \langle N, E \rangle$ where N is a finite set of nodes corresponding to program statements and $E \subseteq N \times N$ is a set of edges corresponding to the possible control-flow between statements. The possible analysis solutions are structured into a lattice $\mathcal{L} = (Solns, \sqsubseteq, \sqcup, \sqcap)$, where the most-precise solution is at the bottom $\perp$ of the lattice and the least-precise solution is at the top $\top$ of the lattice.[3]

Each node k of the CFG maintains two lattice elements corresponding to the analysis solutions immediately before and immediately after that statement: IN_k represents the incoming solution, and OUT_k represents the outgoing solution. At the beginning of the analysis $\text{IN}_k = \text{OUT}_k = \perp$ for every k. Each node k has a transfer function $\mathcal{F}_k$ that transforms IN_k to OUT_k. For all nodes k, the analysis iteratively computes the following two functions until the analysis reaches a fixpoint:

[3]This is actually opposite of the convention usually used by DFA, which reverses the lattice described above; we do this to be consistent with the abstract interpretation convention used later in the chapter.

$$\text{IN}_k = \bigsqcup_{x \in pred(k)} \text{OUT}_x \tag{5.1}$$

$$\text{OUT}_k = \mathcal{F}_k(\text{IN}_k) \tag{5.2}$$

In other words, for each node merge the outgoing information from all immediate predecessor nodes (using the lattice join operator) to get that node's incoming solution, and then apply that node's transfer function to get that node's outgoing solution. The fixpoint computation is usually performed using a worklist. The worklist is initialized to contain the program's entry node; the analysis iteratively performs the following actions until the worklist is empty (signaling the fixpoint has been reached): pop a node k from the worklist; compute IN_k and OUT_k; if OUT_k is changed from its previous value then put all successor nodes of k onto the worklist.

The Importance of Node Ordering. The order in which the worklist processes nodes is irrelevant in terms of correctness, i.e., the analysis will compute the same solution regardless of node ordering. However, it turns out to have significant impact on analysis performance. Intuitively, a bad node ordering can cause paths in the CFG to be redundantly recomputed many times. Suppose a node k is computed to have $\text{OUT}_k = \ell$ where lattice element $\ell \in Solns$, and this information is propagated by the worklist down the

CFG paths starting from k. Later the worklist processes a node that is a predecessor to k, causing k to be processed again, and now $\text{OUT}_k = \ell'$ where $\ell \sqsubseteq \ell'$. Then this new information must be propagated down the CFG again, subsuming the previous solutions along those paths. In the worst case those paths could be recomputed h times where h is the height of the lattice. Thus, a good node ordering is important for performance.

5.2.2 Parallelizing Program Analysis

We categorize the related work on parallelizing flow- and context-sensitive program analysis into three general approaches. We leave out work on parallelizing flow- or context-*insensitive* analysis, such as that by Méndez-Lojo et al. [113, 114], Edvinsson et al. [63], and Nagaraj et al. [122].

Worklist-Based Parallelism. This parallelization strategy operates by processing all nodes currently enqueued on the analysis worklist in parallel. Dwyer et al. [61] discuss a worklist-parallel implementation of the FLAVERS DFA toolset [60] for C. They start a new thread for each node in a global worklist and each thread enqueues its result back in that worklist. The authors report average speedups of 3.8× on 6–9 hardware threads. However, the paper's evaluation is problematic in two respects, making it difficult to interpret the results: (1) the sequential analysis they compare against used an arbitrary node ordering for the worklist, which in our experience can cause slow-downs from 2–

5× relative to a more optimized node ordering strategy; and (2) their evaluation reports analysis runtimes rather than speedups.[4]

Nondeterminism-Based Parallelism. This parallelization strategy looks for nondeterministic branch points in the analysis (e.g., conditional guards with indeterminate truth values) and executes the branches in parallel until control-flow merges again (e.g., after the conditional is finished). This approach is taken by Monniaux [121], who describes a parallel implementation of the Astrée static analyzer [53] for embedded controller code written in C. The parallel implementation exploits the fact that, in this particular application domain, programs often contain dispatch loops over a `switch` statement, and each `case` within the `switch` requires significant analysis effort and is independent of all other cases. Thus each case is analyzed in parallel, achieving speedups between 2–3× on five processors. The usefulness of this method is highly specific both to C and to idioms common in the C programs that Astrée targets. Monniaux claims that a version for general-purpose programs was attempted which parallelized at arbitrary nondeterministic points, and the results were dissappointing [121].

Partition-Based Parallelism. This parallelization strategy partitions the analysis in some way and computes the analysis of each partition in parallel. This strategy is extremely general, with a number of distinct instantiations in the literature.

[4] Speedups speak of both speed and scalability whereas runtimes tell us *only* about how fast something went.

Lee et al. [105, 106] partition their parallel Fortran analysis by strongly-connected components (SCC) in the program's CFG. Each SCC is analyzed in parallel using separate worklists; the SCC solutions are combined using elimination-based techniques [126]. They achieve an average speedup of 4.6× in 8 threads. However, the speedups were relative to their parallel analysis running on a single thread rather than to a specialized sequential version of the analysis.

Weeks et al. [140] partition a parallel analysis for a custom purely-functional language (used to write concurrent applications) using dynamically-discovered dependencies. If statement s_1 is found to be dependent on statement s_2, then s_1 will be put into s_2's partition (unless this would increase s_2's partition size beyond some threshold). The authors report runtimes, but we were able to compute speedup from the provided data. These average 9.4× on 16 threads on two trivial benchmarks handcrafted by the authors.

Albarghouthi et al's parallel C analysis. [30] is query-based (i.e., they do not compute a solution for the entire program, only enough to answer a specific query). They frame the analysis in terms of MapReduce [56], with a parallel map phase and a sequential reduce phase. During the map phase, multiple functions are analyzed intraprocedurally in parallel. If a function call is encountered, then the call is enqueued to be analyzed later. During the reduce phase, sequential dependencies are accounted for.

The process is repeated on the enqueued function calls until a fixpoint is reached. They achieve an average speedup of 3.71× on 8 hardware threads.

5.2.3 Problems with the DFA Approach for Parallelism

A number of the existing approaches to parallelizing analysis, as described above, require a CFG as input. For languages such as C and Fortran this is a reasonable assumption; however, for a language like JavaScript it is not reasonable at all. Javascript's higher-order functions, prototype-based inheritance, implicit type conversions and implicit exceptions, and other language features mean that a computing a useful CFG requires extensive, precise, and costly analysis—the very kind of analysis we are trying to optimize via parallelization.

In addition, the DFA approach itself can make it more difficult to see opportunities for parallelization. The traditional formulation of DFA is inherently sequential. We observe that equations (1) and (2) in Section 5.2.1 implicitly impose synchronization points into the very definition of the analysis itself, as they require multiple nodes to cooperate in order to merge and propagate information between themselves. Synchronization is (almost) unavoidable for a tractable analysis, but the DFA framework makes it difficult to separate synchronization out as a separate concern from the analysis itself.

5.3 Designing for Parallelism

Our key insight is that by designing and implementing the program analysis in a certain way, the design space of parallelization strategies becomes clearer and implementing the parallelization strategies effectively becomes easier. In particular, we take advantage of an approach to program anaysis based on abstract interpretation which we call $\text{STS}_\triangledown$; this approach divides the analysis into two separate components: an embarrassingly parallel reachability computation on a state transition system, and a strategy for selectively merging states during that reachability computation. We describe this program analysis approach below, and then discuss the parallelism design space exposed by this anaysis perspective.

5.3.1 The $\text{STS}_\triangledown$ Approach to Program Analysis

The basis for the $\text{STS}_\triangledown$ approach to program analysis is described in [82]; we summarize the approach in this section. Note that everything in this section refers to a completely sequential definition of program analysis; there is no parallelism. Fundamentally, the $\text{STS}_\triangledown$ model specifies a static analysis in two parts: (1) the underlying analysis itself, described as a state transition system (STS); and (2) a strategy for when to merge states together, used to bound the reachable state space while maintaining the soundness of the analysis.[5] The solution to the analysis is the set of reachable states in

[5] As described in [82] this strategy is a widening operator $\triangledown$ in the abstract interpretation sense.

the STS from some given initial state; the state merging strategy specifies the *control flow sensitivity* of the analysis, i.e., its path-, flow-, context-, and heap-sensitivity. Thus, the analysis and its sensitivity are treated as two separate and independent concerns. The key insight of this work, as opposed to [82], is that this separation of concerns can greatly benefit parallelism in a way described in later sections of this chapter.

An abstract machine-based smallstep operational semantics is a useful way to describe a static analysis [138], and can easily be seen as a STS. Such a semantics defines a notion of *abstract state* (e.g., a program point together with the current abstract values of all variables in scope at that program point) and a set of *transition rules* that uses the semantics of the program statement at that program point to map an abstract state to a new abstract state. For example, if the abstract state is $\langle \mathtt{pp3}, [x \mapsto 1] \rangle$ and the statement at program point 3 is "`x += 1`", then the next state would be $\langle \mathtt{pp4}, [x \mapsto 2] \rangle$. The exact definition of an abstract state and the transition rules would vary depending on the language being analyzed and the analysis being defined. Without going into details on the exact state definition and transition rules for a particular language and analysis, we can formalize this idea as the following:

$\hat{\varsigma} \in \Sigma^\sharp$	abstract states
$\mathcal{F}^\sharp \subseteq \Sigma^\sharp \times \Sigma^\sharp$	transition relation

The abstract states form a lattice $\mathcal{L} = (\Sigma^\sharp, \sqsubseteq, \sqcap, \sqcup)$, where $\sqsubseteq$ is the ordering relation, $\sqcap$ is the meet operator, and $\sqcup$ is the join operator. The solution to the program analysis is defined as the least fixpoint (**lfp**) of the abstract semantics from some set of initial states $\Sigma^\sharp_{\mathcal{I}}$. Define the operator $\mathring{\cdot}$ so that for set S and any function on sets $\mathcal{F}$, $\mathring{\mathcal{F}}(S) = S \cup \mathcal{F}(S)$. Then the analysis solution $[\![P]\!]^\sharp$ for program P is defined by:

$$[\![P]\!]^\sharp = \textbf{lfp}_{\Sigma^\sharp_{\mathcal{I}}} \mathring{\mathcal{F}}^\sharp$$

An operational view of this least fixpoint definition is as a worklist algorithm: it initializes the worklist with the states in $\Sigma^\sharp_{\mathcal{I}}$, then iteratively it (1) removes the current states from the worklist; (2) applies $\mathcal{F}^\sharp$ to them to get a set of new states; (3) filters out any states it has seen already; and (4) puts the remaining states into the worklist. This continues until the worklist is empty, at which point it has computed the entire set of possible states, thus concluding the program analysis.

However, the analysis as defined is intractable (in fact, potentially uncomputable). The issue is control-flow—specifically, the nondeterministic choices that must be made because of the analysis' over-approximations: which branch of a conditional should be taken, whether a loop should be entered or exited, which (indirect) function should be called, etc. The number of abstract states grows exponentially with the number of

nondeterministic choices, and is potentially unbounded. We must extend the analysis to control this behavior.

Therefore, we apply a *widening operator* $\triangledown$ to the analysis which bounds the abstract state space by selectively merging abstract states, thus losing precision but making the analysis tractable. This widening operator will, at each step of the fixpoint computation: (1) partition the current set of reachable states into disjoint sets; (2) for each partition, merge all of the abstract states in that partition into a single abstract state that over-approximates the entire partition; (3) union the resulting abstract states together into a new set that contains only a single abstract state per partition. This allows us to limit the number of states by fixing a particular number of partitions. By defining different strategies for partitioning the abstract states, we can control how states are merged and thus control the precision and performance of the analysis. As shown in [82], this partitioning strategy is actually synonymous with the analysis control flow sensitivity.

Operationally, this means that we modify the worklist algorithm so that it maintains a memoization table with one entry (i.e., abstract state) per partition. At each step the algorithm selects a state from the worklist, uses $\mathcal{F}^\sharp$ to compute a new set of states, merges them into the appropriate partition entries using $\triangledown$, and if any of those partition entries have changed due to the newly-merged information, adds them back into the worklist. In pseudocode, this operational view of the $\text{STS}_\triangledown$ model looks like the following:

Example 4 The sequential worklist algorithm

put the initial abstract state $\hat{\varsigma}_0$ on the `worklist`
initialize map `memo` : $Partition \rightarrow State^\sharp$ to empty
repeat
  remove an abstract state $\hat{\varsigma}$ from the `worklist`
  for all abstract states $\hat{\varsigma}'$ in `next_states`($\hat{\varsigma}$) **do**
    if `memo` does not contain `partition`($\hat{\varsigma}'$) **then**
      `memo`(`partition`($\hat{\varsigma}'$)) = $\hat{\varsigma}'$
      put $\hat{\varsigma}'$ on `worklist`
    else
      $\hat{\varsigma}_{old}$ = `memo`(`partition`($\hat{\varsigma}'$))
      $\hat{\varsigma}_{new} = \hat{\varsigma}_{old} \sqcup \hat{\varsigma}'$
      if $\hat{\varsigma}_{new} \neq \hat{\varsigma}_{old}$ **then**
        `memo`(`partition`($\hat{\varsigma}'$)) = $\hat{\varsigma}_{new}$
        put $\hat{\varsigma}_{new}$ on `worklist`
      end if
    end if
  end for
until `worklist` is empty

The `next_states` function applies the state transition rules to determine the next abstract state(s) reachable from the given abstract state—this entails the computational core of the analysis logic. The `partition` function maps an abstract state to its partition as defined by the state merging strategy. The algorithm computes the analysis fixpoint exactly as described earlier.

5.3.2 Parallelism Design Space

The $\text{STS}_\triangledown$ program analysis model provides a useful perspective for parallelizing analysis, because it boils the problem down to two questions: (1) what strategy should

we use to merge states during the reachability computation (thus injecting synchronization points); and (2) what granularity should we use to parallelize the reachability computation itself?

Recall that the state merging strategy is synonymous with the flow- and context-sensitivity of the analysis—merging fewer states means greater sensitivity and thus greater precision, while merging more states means less sensitivity and thus less precision. With respect to parallelization, there is a tradeoff between merging strategies that merge fewer states (reducing synchronization but increasing the number reachable states), versus strategies that merge fewer states (increase synchronization but reducing the number of reachable states). We explore a small part of this space in our evaluation, however, there is interesting future work in exploring this trade-off further.

Besides state merging, the remaining question is granularity, which we explore in the rest of this section. We first discuss an obvious point in this space, the worklist-parallel strategy, and why it is not a satisfactory solution. We then introduce a novel point in this space, the per-context strategy, that has not been explored before.

Worklist-Parallel Strategy. The most straightforward granularity strategy is to parallelize the worklist loop by processing each node on the worklist in parallel. In essence, we explore the reachability of each node independently until the various states reach some merge point specified by the merge strategy (but not necessarily the same merge

point for all states), whereupon the merged states are inserted back into the global worklist for the process to be repeated. The pseudocode of the analysis for this strategy looks like the following:

Example 5 The worklist parallel algorithm

```
put the initial abstract state ς̂0 on the worklist
initialize templist to empty
initialize map memo : Partition → State♯ to empty
repeat
    for all abstract states ς̂ in the worklist do in parallel
        for all abstract states ς̂′ in next_states(ς̂) do
            begin thread-safe
                if memo does not contain partition(ς̂′) then
                    memo(partition(ς̂′)) = ς̂′
                    put ς̂′ on templist
                else
                    ς̂old = memo(partition(ς̂′))
                    ς̂new = ς̂old ⊔ ς̂′
                    if ς̂new ≠ ς̂old then
                        memo(partition(ς̂′)) = ς̂new
                        put ς̂new on templist
                    end if
                end if
            end thread-safe
        end for
    end parallel for
    swap worklist and templist
until worklist is empty
```

The **thread-safe** block is run atomically using synchronization primitives.

There are three major drawbacks to this strategy. First, it can cause a great deal of redundant computation because of node ordering issues (as described in Section 5.2.1). If multiple states are being processed in parallel but one subsumes the others, then

the parallel computations are not actually useful and there is no gain in performance. Second, all of the parallel computations must be synchronized together, even those that reach different merge points (and hence are independent). This is because the analysis doesn't know which threads will reach which merge points, and thus must wait until all threads reach some merge point before it can continue at any one merge point. Finally, this strategy introduces a large number of short-lived threads, which can be detrimental to performance.

Per-Context Parallel Strategy. We propose a novel point in the granularity design space based on function contexts, one that attempts to address some of the issues of the worklist-parallel strategy and is motivated by empirical observation. We want to reduce node ordering issues, limit synchronization between independent parts of the analysis, and increase the granularity of the thread computations. Context-sensitive analyses have desirable properties which can be exploited for these goals. Context-sensitive analyses *clone* functions based on some notion of abstract calling context (the exact definition of "context" defines the particular type of context-sensitivity used by the analysis). Each clone is specialized to a particular context and, most importantly, analyzed separately. Different clones can be analyzed in parallel, while analysis of a single clone can be done sequentially. This strategy allows a more optimal node ordering, because within each context we can sequentially analyze nodes in reverse

postorder (the best possible node ordering). Different contexts are independent of each other, which limits synchronization. Finally, threads now compute an entire function rather than a single statement or basic block, increasing work granularity per thread and reducing thread management overhead. The pseudocode of the analysis using this strategy is as follows:

In the above algorithm, a unique thread is used to run ANALYSISTHREAD per context. The global map `backlog` maps each context to a synchronized queue, while `worklist` is local to each thread. The function `context` extracts the context under which an abstract state needs to be analyzed. Note that no synchronization is required on access to `memo` (because each thread is run sequentially and multiple threads do not access same parts of `memo`). The procedure PROCESS checks if no thread corresponding to context is running, which can happen under two circumstances: (1) the context has never been seen before, thus a new thread is used to run ANALYSISTHREAD with that context (2) the thread corresponding to the context has marked itself as potentially done, in which case the thread is unmarked and woken up back again to run ANALYSISTHREAD. The analysis begins by calling MAIN, and the analysis ends when each of the threads mark themselves as potentially done and each of the backlog queues are empty.

While a per-context strategy has been previously mentioned in the literature [63], to our knowledge this is the first time it has ever been detailed and implemented. Addition-

Example 6 The per-context parallel algorithm

```
procedure ANALYSISTHREAD(ctxt)
    move abstract states from backlog(ctxt) to worklist
    repeat
        remove an abstract state ς̂ from the worklist
        for all abstract states ς̂′ in next_states(ς̂) do
            if context(ς̂′) ≠ ctxt then
                PROCESS(context(ς̂′), ς̂′)
            else if memo does not contain partition(ς̂′) then
                memo(partition(ς̂′)) = ς̂′
                put ς̂′ on worklist
            else
                ς̂_old = memo(partition(ς̂′))
                ς̂_new = ς̂_old ⊔ ς̂′
                if ς̂_new ≠ ς̂_old then
                    memo(partition(ς̂′)) = ς̂_new
                    put ς̂_new on worklist
                end if
            end if
        end for
    until worklist is empty
    if backlog(ctxt) is empty then
        mark this thread as potentially done
    else
        ANALYSISTHREAD(ctxt)
    end if
end procedure

procedure PROCESS(ctxt, ς̂)
    begin thread-safe
        enqueue ς̂ into backlog(ctxt)
        if no thread corresponding to ctxt is running then
            ANALYSISTHREAD(ctxt)
        end if
    end thread-safe
end procedure

procedure MAIN
    initialize map memo : Partition → State♯ to empty
    ANALYSISTHREAD(context(ς̂_0))
end procedure
```

ally, thanks to the $\text{STS}_{\triangledown}$ representation of the analysis, using the per-context strategy is simple. Instead of a global worklist, use one worklist per context encountered during the analysis. Each worklist has a dedicated thread computing a fixpoint. When a thread processes a function call leading to a new context, it passes the resulting state on to the appropriate thread and continues processing its own worklist. The only synchronization required is this thread communication. When all worklists are empty, the analysis has reached a global fixpoint.

5.4 Parallel JavaScript Analysis

In this section we briefly describe the JavaScript language and the existing sequential JavaScript analysis that we adapted for our parallel analysis. We then describe the modification to that sequential analysis necessary to implement our parallel analysis design.

5.4.1 JavaScript Features

JavaScript is an imperative, dynamically-typed language with objects, prototype-based inheritance, higher-order functions, implicitly applied type-conversions, and exceptions. JavaScript programs only have two scopes (global scope and function scope), though variables and functions are allowed to be defined anywhere; these declara-

tions (but not the corresponding initializations, except for functions) are automatically hoisted to the appropriate scoping level. JavaScript is designed to be as resilient as possible: when a program performs some action that doesn't make sense (e.g., accessing a property of a non-object, or adding a boolean and a function together) JavaScript uses implicit conversions and default behaviors when possible in order to continue the execution without errors rather than raising an exception.

Objects are the fundamental JavaScript data structure.[6] Object properties can be dynamically inserted and deleted, and when performing a property access the specific property being accessed can be computed at runtime. JavaScript features such as the `for..in` loop and the `in` operator allow for reflective introspection of an object's contents. Object inheritance is handled via delegation: when accessing a property that is not present in a given object *obj*, the property lookup algorithm determines whether *obj* has some other object *proto* as its prototype; if so then the lookup is recursively propagated to *proto*.

These features have two important implications for static analysis: (1) computing a precise CFG requires careful and costly analysis, because higher-order functions, prototype-based inheritance, implicit type-conversions, and implicit exceptions make control-flow non-obvious, thus analysis techniques based on the CFG are problematic;

[6] Even functions and arrays are just special kinds of objects, and can be used in the same ways as other objects.

and (2) JavaScript's inherent dynamism means that high precision is important to get useful results, implying that any useful analysis will be expensive.

5.4.2 Sequential JSAI

We build on an existing sequential abstract interpreter for JavaScript called JSAI [96]. The analysis performs type inference, control-flow analysis, pointer analysis, and numeric, string, and boolean value analysis. JSAI is designed and implemented as an abstract machine-based smallstep operational semantics, which can be thought of as a state transition system. Rather than baking in a specific flow-, context-, and heap-sensitivity strategy, JSAI is designed around the $\text{STS}_{\triangledown}$ model in order to have configurable control flow sensitivity [82]: the basic analysis computes the reachable states of the STS defined by the abstract semantics, while a separate modular component determines a strategy for selectively merging states. States are represented as tuples holding relevant components such as the values on the stack and heap, the current continuation, and a trace recording the execution history. The set of states forms a lattice; states are merged using the lattice join operator which operates pointwise on the state components. The choice of which states to merge and when is determined by JSAI's merging strategy, which can be chosen independently from the rest of the analysis. A given merging strategy determines the flow-, context-, and heap-sensitivity of the analysis; indeed, merging strategies and sensitivities are synonymous.

JSAI is formally specified and the code is designed to have a close correspondence with the formalisms, using immutable states and written using mostly pure functional style, making it easy to follow and manipulate. Alternatively, we could have used TAJS [87, 89, 90], a competing sequential JavaScript analysis framework, whose runtimes are in the same order of magnitude as JSAI (between $0.3\times$ and $1.8\times$). However, TAJS does not use the $\text{STS}_{\triangledown}$ model, does not offer configurable sensitivity, and lacks a formal specification.

5.4.3 Parallelism Strategies

We implement two specific parallelism strategies as discussed in Section 5.3: the worklist-parallel strategy and the per-context strategy. We describe for each one the necessary changes to JSAI, which were minimal in both cases. Our experience is that implementing different strategies is a simple task, making it easy to explore the design space of the $\text{STS}_{\triangledown}$ model. For each strategy we use a single global thread pool [15, 35] of a fixed size and create new thread tasks for the pool on demand. We also replace JSAI's memoization table (which holds the computed solution as the analysis executes, mapping program points to the abstract states computed at that program point so far) with a thread-safe version that requires no locking for table lookups [16].

Worklist-Parallel Strategy. This strategy is the simplest to implement. Rather than iteratively popping elements off of the worklist and processing them sequentially, instead we pop *all* elements of the worklist and process each element in parallel, having them enqueue the resulting abstract states back onto the global worklist. This strategy is, in concept, the same strategy used by Dwyer et al [61], and we implement it to use as a comparison point for our novel proposed strategy given below.

Per-Context Parallel Strategy. We observe that for the JavaScript benchmarks we have tested, if N states are on a non-empty worklist, and those N states are members of M contexts (where $1 \leq M \leq N$), then typically $M >> 1$. In other words, many contexts are typically enqueued for processing at once. This indicates that the per-context strategy described in Section 5.3 has promise for anayzing JavaScript. Instead of a global worklist, we use one worklist per context, with one thread for each worklist. Each worklist has an associated non-blocking, lock-free [17, 116] *backlog* queue that other threads use to enqueue work for that thread; whenever a thread runs out of elements in its worklist, it puts its backlog queue into its worklist and continues. When a thread processes a function call that belongs to a new context, it puts the resulting abstract state into that context's backlog queue. The analysis has reached a fixpoint when all worklists and backlog queues are empty. The memoization table is global; because contexts are independent from each other, there will never be a conflict between threads

when updating the memoization table. We also tried an alternative to the backlog queue strategy for thread communication, wherein threads directly enqueued work into other threads' worklists; we saw results ranging from 30% faster to 18% slower performance relative to the backlog queue implementation, with most benchmarks being slower; thus we only use the backlog queue implementation in our evaluation.

5.5 Evaluation

We evaluate our parallel JSAI implementation using a set of real-world JavaScript benchmarks, detailed in Section 5.5.2. We decribe the benchmarks and our experimental methodology, then present and discuss our results for the worklist-parallel strategy and the per-context strategy.

5.5.1 Experimental Methodology

System Under Test. Our testbed is equipped with two 6-core Intel Xeon processors running at 1.9 Ghz with hyperthreading enabled. We only report data for 1-12 threads, with one thread per core. While utilizing hyperthreading with 13-24 threads usually did yield better speedups, these tended to be minimal and with high variability. The machine is equipped with a total of 32 GB of memory, and we ran with a maximum

JVM heap size of 25,600 MB for all experiments. During the course of the experiments, we had exclusive access to the machine, and all non-essential services were disabled.

Calculating Speedups. The speedups we report are relative to the sequential JSAI implementation, as per the usual definition of speedup. This is an important point for comparing against related work. In several cases, authors have instead focused on runtimes [140], speedups relative to the framework itself [105, 106, 114], percent improvement in performance [103], or atypical presentations of speedups [121].

Configuration Focus. Previous experiments [96] have shown that stack-based CFA tends to work well for JavaScript, both in terms of precision and performance. We specifically use `stack-k-h` CFA, in which the top k callsites on the call stack are used as the context (this is the standard "callstring context sensitivity" used in DFA). The parameter h controls the heap sensitivity, which distinguishes each abstract object allocation by a context of depth h, in addition to its program location. We show results for `stack-5-4` (most precise) in this section.

Testing. In order to test the correctness of our implementation, we annotated the benchmarks with special statements to print out the final abstract values for certain program points. In all cases, the solutions from our parallel implementation were equivalent to the solutions produced by the sequential interpreter. In addition, we ran several

hundred handcrafted tests on both the sequential and parallel analyzers to compare their results; in all cases they agree.

5.5.2 Benchmarks

We focus on ECMA3-compliant JavaScript programs which do not exercise the document object model (DOM). While SunSpider and other concrete performance benchmark suites meet the above criteria, most of their constituents complete analysis within seconds. Given that short-running benchmarks can be improved little by the addition of parallelism, such benchmarks have been omitted from our evaluation.

In an attempt to derive more complex benchmarks which are more time-consuming for our analysis to handle, we have turned to open source JavaScript programs in the wild. This allows us to benchmark against a more realistic suite. Additionally, we have intentionally selected benchmarks representing a variety of coding styles, with both imperative and functional code. This allows us to determine whether or not our parallel analysis performance is dependent on coding style, which is important considering that JavaScript allows for very different styles to be used and to coexist. A complete description of our benchmark suite is given in Table 5.1.

Benchmark	Derived From	General Kind	Number of Functions	LOC	Sequential Runtime Under `stack-5-4` (s)
`cryptobench`	[18] (SunSpider origin)	imperative	132	1699	508.082
`md5`	[19]	imperative	37	365	778.061
`buckets`	[20]	mixed	168	2472	73.801
`numbers`	[21]	mixed	90	1870	145.082
`jsparse`	[22]	functional	74	878	515.239
`linq_action`	[?]	functional	381	2783	32.097
`linq_aggregate`	[?]	functional	396	2830	80722.088
`linq_functional`	[?]	functional	378	2790	4516.588

Table 5.1: A summary of our benchmark suite. The `linq*` benchmarks all execute different APIs from the same common library in a manner that causes vastly different code paths to be analyzed between the three benchmarks. Benchmarks of the *mixed* kind have both imperative and functional characteristics based on subjective observation.

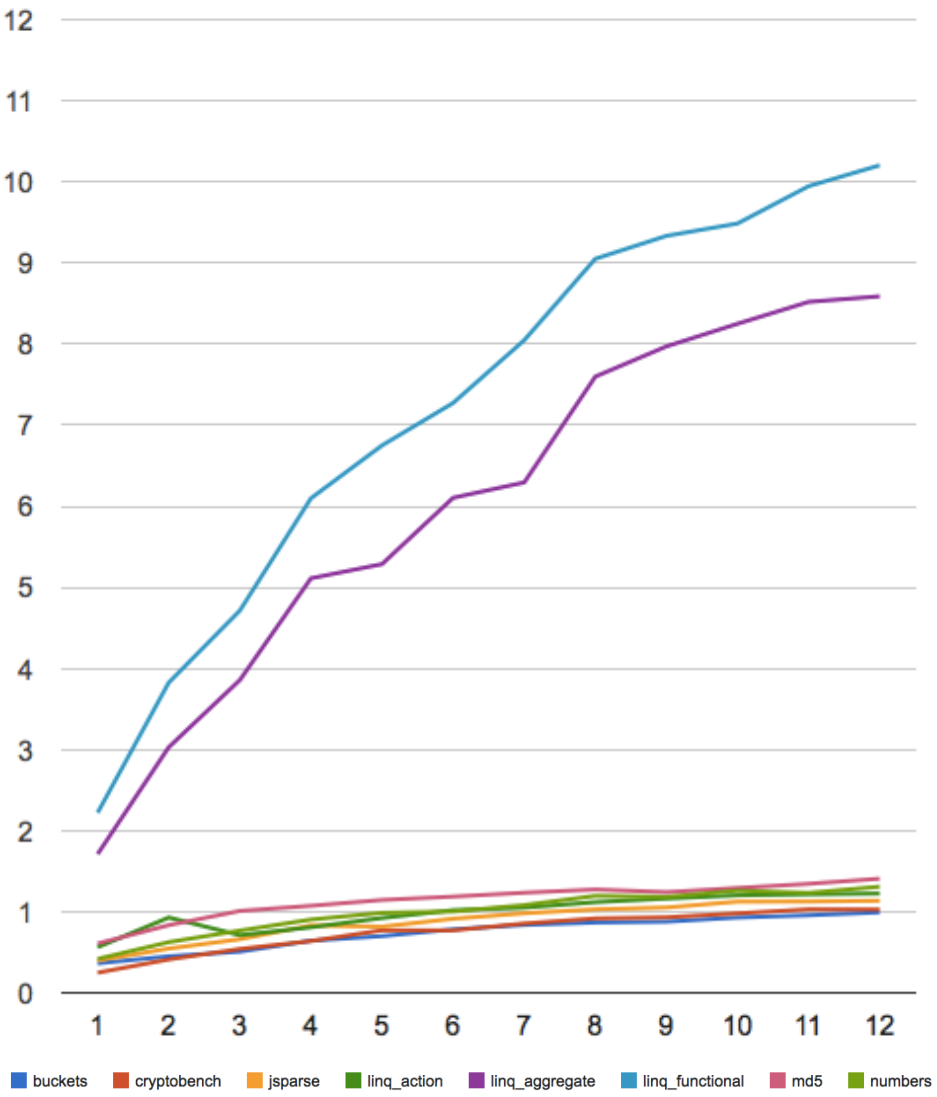

Figure 5.1: Worklist-parallel speedups for the trace `stack-5-4`. The number of hardware threads used is on the `x` axis, and the speedup is on the `y` axis.

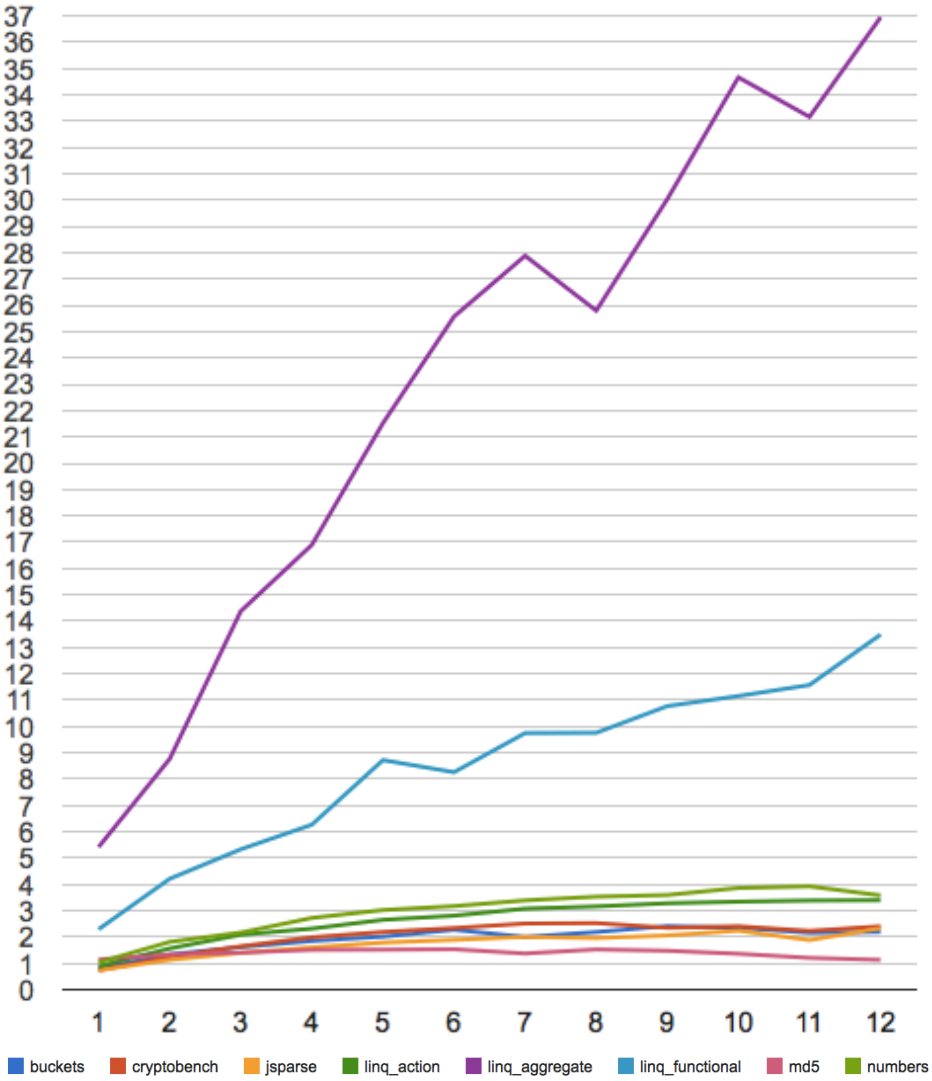

Figure 5.2: Per-context parallel speedups for the trace `stack-5-4`. The number of hardware threads used is on the `x` axis, and the speedup is on the `y` axis.

5.5.3 Worklist-Parallel Results

The performance results for the worklist-parallel strategy on the configuration `stack-5-4` is given in Figure 5.1. Under `stack-5-4` the worklist-parallel results show real promise with the `linq_aggregate` and `linq_functional` benchmarks. Both `linq_aggregate` and `linq_functional` benchmarks show superlinear speedup for less than 10 hardware threads. The superlinear speedup seems to be the result of two factors. First, both the `linq_aggregate` and `linq_functional` benchmarks are highly functional in implementation style, containing more than one function per ten lines of code. Moreover, many of these functions are used in a higher-order style, are largely independent of each other, and are of small to moderate length. Intuitively, this leads to many contexts of a granularity level well-suited to parallel processing. Second, node ordering is also probably a factor. The sequential analyzer enforces an arbitrary, possibly far-from-optimal ordering between states in different contexts (it uses reverse post-order *within* a context, but without the results of the analysis it isn't possible to order *between* contexts). It is possible that our worklist-parallel implementation just happens to choose better node orderings on these benchmarks. The somewhat erratic nature of our speedup curves serves as further evidence of these ordering issues. It is unclear how to measure the effects of ordering in a structured way, therefore, we do not have any experiments that can backup our conjecture. There are many examples from the literature where either superlinear or otherwise better than predicted performance has

been recorded [30, 61, 63, 105], and different node orderings were commonly cited as the reason.

In stark contrast to the excellent speedups of `linq_aggregate` and `linq_functional` on `stack-5-4`, the rest of the benchmarks see rather dismal performance. No other benchmark was able to reach a speedup higher than 1.45×, irrespective of the number of hardware threads used. This implies that for these benchmarks, the worklist-parallel approach duplicates a significant amount of work. One exception to this seems to be `linq_action`, which was derived from the same codebase as `linq_aggregate` and `linq_functional`. Given that `linq_action` has a fairly short sequential runtime at around 32 seconds, it seems that it is simply too short to see much improvement from the worklist-parallel strategy.

5.5.4 Per-Context Parallel Results

Speedups for the per-context parallel implementation with our `stack-5-4` trace on our benchmark suite are shown in Figure 5.2. Once again, `linq_aggregate` and `linq_functional` stand out, unconditionally showing higher speedups in all cases than any other benchmark. Moreover, both benchmarks show superlinear behavior for less than 10 hardware threads, presumably for the same reasons as detailed in the previous section. Of particular interest is that based on the performance results, it appears that there are three buckets in which data can be distributed based on their

relative performance to each other. This bucketing shows that functional programs tend to perform better than non-functional programs, with all of our functional benchmarks being either in the top-performing or moderate-performing bucket. However, this is not to say that non-functional programs can not perform well; the moderately-performing bucket holds `cryptobench`, a highly imperative benchmark.

With the trend of functional programs generally performing better, speedups underneath `stack-5-4` in Figure 5.2 is particularly interesting. The `linq_aggregate` and `linq_functional` benchmarks both show superlinear speedup, with `linq_aggregate` seeing speedups higher than 34× on 10 cores. `linq_aggregate` shows strong evidence that node ordering is to blame, given the sudden drop in performance at 8 hardware threads under `stack-5-4`. It seems likely that the sequential interpreter chooses a particularly poor context ordering for `linq_aggregate` underneath the `stack-5-4` trace. This same sort of performance drop is also seen with the `numbers` benchmark underneath `stack-5-4`. As such, context ordering plays a significant role, even for the parallel abstract interpreter.

5.6 Conclusions

We have presented a alternative program analysis model to the more usual DFA approach, called $\text{STS}_{\triangledown}$. This framework makes it easy to reason about and explore

different parallelization strategies, as well as being more applicable than DFA to languages with difficult control-flow that make the program CFG hard to compute. Using this framework we have implemented a parallel analysis for JavaScript and explored two points in the parallel design space: a naive worklist-parallel strategy and a novel per-context strategy. Our results show that our parallel implementation provides speedups comparable or better than the speedups reported in our related work for realistic JavaScript programs.

Chapter 6

Application of JSAI to Security of JavaScript-based Browser Addons

6.1 Introduction

The web-browser addon framework is a powerful and popular mechanism for extending browser behavior—thousands of third-party developers are creating addons, and browser users have downloaded billions of copies [23]. These addons have almost complete access to a user's information: browser history, cookies, passwords, clipboard, geo-location, mouse and keyboard actions, the local filesystem, and more. Malicious addons are trivially easy to write, and yet can be difficult to detect. Thus, vetting third-party addons is critical both for users (whose information is at risk) and for browser providers (whose reputations are at risk). However, the current vetting process for addons submitted to official addon repositories is mostly manual and completely ad-hoc.

Our goal is to help automate this vetting process by creating an analysis to automatically infer *security signatures* for JavaScript-based browser addons. A security signature captures both (1) information flows between interesting sources and sinks, for example, *from* the current browser URL *to* a network message; and (2) interesting API usage, for example, to detect deprecated or unsafe APIs. API usage inference is treated as a special case of information flow—in essence, can any information potentially flow to a use of that API. This signature inference analysis can be used by official addon repositories upon addon submission (and also by third-party developers prior to submission) to detect potential security problems, thus reducing the vetting burden and increasing addon security.

6.1.1 Key Challenges

We must address three key challenges to enable security analysis of browser addons:

1. **Flexible Security Policies**: Naively, we might expect to use a standard information flow analysis [127] to establish the security of an addon. For such an analysis, we would use a security lattice to specify a security policy describing allowable information flows, and report any information flows in the addon that violate the specified policy. Unfortunately, there is no single security policy (and hence no single security lattice) that is suitable for all addons. Whether an addon's information flow is secure or not depends largely on that addon's intended

purpose. For example, the current URL being browsed by the user should usually be private. However, if an addon's intended purpose is send URLs over the network to an URL shortener service, then this information flow is expected and allowed. There are many other examples of information flows that would usually be considered insecure, but that are allowable given the intended purpose of the addon. Thus, we need a more flexible solution than traditional information flow analysis.

2. **Classifying Information Flows**: Traditional information flow analysis simply reports whether a *leak* (a flow violating the given security policy) might occur. However, this information alone is not useful for our purpose—there are many possible ways for information flow to happen, with varying levels of importance and concern. We must be able to classify the information flows to aid the addon vetter in their task and enable them to understand exactly what the addon is doing. This requires a more discriminating analysis than traditional information flow.

3. **Inferring Network Domains**: A large part of addon security concerns the network domains that the addon communicates with. In JavaScript, these domains are created and passed around in the form of strings. It requires careful and precise analysis to recover the actual network domains from these strings.

6.1.2 Our Contributions

To meet these challenges, we present the following contributions:

1. **Annotated Program Dependence Graph**: We base our analysis on the Program Dependence Graph (PDG) [66]. Defining and implementing a PDG for JavaScript is novel; moreover, we extend the classic definition with a novel set of graph annotations that allow us to classify various information flows according to their natures. We use the annotated PDG specifically for information flow in this work, but it can be more generally useful, e.g., for program slicing, code obfuscation, code compression, and various code optimizations. The annotated PDG definition and construction algorithm are described in Section 6.3.

2. **Security Signatures**: To accomodate the fluid nature of addon security policies, we develop a novel notion of *addon security signatures*. Rather than attempting to enforce a specific policy, instead we infer interesting flows and API usages and present them to the vetter, allowing them to compare the inferred signature against the addon description to decide whether the addon should be accepted. We define what constitutes a signature and how to construct a signature from the annotated PDG. The definition of security signatures and their construction are described in Section 6.4.

3. **Prefix String Analysis**: Inferring network domains from strings requires precise analysis, but that analysis must also remain tractable. We have defined a sweet-spot in this space by developing a prefix string analysis that is precise enough to compute most of the statically-determinable network domains while still retaining practical performance. This analysis is described in Section 6.5.

4. **Evaluation**: Finally, we evaluate the usefulness and practicality of our work by inferring security signatures for a set of ten real browser addons taken from the Mozilla Firefox offical addon repository. The evaluation and results are described in Section 6.6.

6.2 Background

In this section we provide necessary background information on addons, as well as illustrative examples (taken from real addons) of how addons can violate user privacy.

Addon Security Context Modern web-browsers offer the ability to extend browser behavior with user-installed *addons* (also called *extensions*). Addons[1] are written in JavaScript by third-party developers; they have much higher privileges than client-side JavaScript programs, and they are *not* subject to the sandboxing and other security re-

[1]Extensions to browsers written in native code are referred to as browser plugins, and they are not the focus of our work.

strictions that exist for client-side programs. Proof-of-concept malicious addons have been developed that demonstrate how easily such privileges can be misused [24, 25], and other researchers have demonstrated that even non-malicious addons can be exploited to break security [40, 108]. These are not just theoretical problems; for example, the Mozilla vetting team has seen a number of submitted addons that contain malicious code copied from these published exploits [32].

Addon Execution Addons use an event-driven programming model: they continuously execute a loop responding to *events* such as mouse movement and clicks, keyboard entry, page loads, network responses, timeouts, etc. When the browser first starts up the addon code is fully evaluated, during which a set of event handlers are registered. Then the addon enters an loop in which the following two steps are executed infinitely often: (1) if the event queue is not empty, then an event is pulled off the event queue; (2) if there is an event handler corresponding to the given event, then the handler is invoked and evaluated to completion. More event handlers can be registered and existing handlers can be de-registered during the event handling phase.

Addon Vetting The current addon vetting process for official addon repositories employs volunteers who manually inspect addon code. There are no fully documented or precisely specified security policies, rather, the vetters look for "code smells". Any addon that does not pass the sniff test is rejected. Dynamic code injection is particu-

larly discouraged, given the difficulty in statically determining what the dynamically-injected code will do. This fact is encouraging for static analysis, because unlike client-side JavaScript (which uses `eval` and related APIs heavily) we can safely disallow addons from using dynamic code. Our analysis reports any potential use of these restricted APIs.

Privacy Leaks An addon's elevated privileges make it trivial to leak private user information. We concern ourselves with two kinds of information flows: *explicit flows* (due to data dependencies) and *implicit flows* (due to control dependencies). Timing and termination flows are beyond the scope of this work. We give two examples derived from actual information flows discovered by our analysis in real addons that have been downloaded millions of times. In these examples, the current URL being browsed by the user is accessed by the addon via `content.location.href`, and the call `XHRWrapper(publicServer)` sets up a cross site request to the network domain `publicServer`. Consider the code:

```
function ajax(params) {
  var data = params["data"];
  request = XHRWrapper(publicServer);
  request.send("url is: " + data);
```

```
 }
 ajax({ data: content.location.href });
```

Here, the current URL is used to construct the `data` field of an object literal passed as an argument to `ajax`. The function `ajax` creates a network request to `publicServer` over which the `data` field of its formal parameter is sent, thus explicitly leaking the private URL information. Now consider the code:

```
 window.addEventListener("load", check, false);
 function check(e) {
  var seen = false;
  if (content.location.href == "sensitive.com")
    seen = true;
  var request = XHRWrapper(publicServer);
  request.send(seen);
}
```

Here `check` is registered as an event handler for page load events, thus, whenever the user loads a new page `check` is executed. `check` sets `seen` to true only if the current URL is `sensitive.com`, and then sends `seen` over the network to `publicServer`. This code implicitly leaks private information about whether the user visits `sensitive.com`.

These are just two—certainly non-exhaustive—ways in which privacy can be breached by addons. Our main goal in this work is to develop a static analysis for JavaScript addons that can reliably and precisely detect these kinds of privacy leaks, as well as distinguish between various kinds of leaks.

6.3 Annotated PDGs for JavaScript

A *Program Dependence Graph* (PDG) [31, 38, 66] is an explicit representation of a program's data and control dependencies. We use an novel extended variant of PDGs as a basis for our security signature inference (described in Section 6.4). The relation between information flow and program dependencies has been noted before (e.g., Abadi et al [29]) and has previously been exploited for information flow analysis of Java bytecode [80]. Our novel contributions are (1) defining PDG construction for JavaScript; and (2) a set of annotations for the PDG that allow us to classify the various types of information flows found in a program.

We assume we are given a base analysis for JavaScript that is flow- and context-sensitive and computes a reduced product of pointer analysis (what objects a reference may point to), string analysis (what set of strings a value may represent), and control-flow analysis (what functions a call may refer to). Any such base analysis can be used for our technique; two existing analyses that meet these requirements are JSAI [96]

and TAJS [89]. From this information we compute the following as input to our PDG construction:

1. A context-sensitive interprocedural control flow graph (CFG), with one node per statement per context.

2. Read and write sets for each statement under each context, consisting of the set of variables and the set of (object, property) pairs that the statement may read from or write to. JavaScript uses computable property accesses, i.e., an object property name is a string that can be computed at runtime, unlike languages such as Java where object fields are statically known. Therefore, the object properties in the read/write sets are actually abstract strings (elements from the abstract string domain used in the base analysis) representing potentially multiple possible concrete property names.

In the rest of this section, we explain how to use this information to construct an annotated PDG. We first define the annotated PDG, then describe the two stages of PDG construction: constructing the annotated data dependence graph (DDG) and constructing the annotated control dependence graph (CDG).

6.3.1 Defining the Annotated PDG

A classic PDG is a graph (V, E) such that $v \in V$ are the program statements and there is an edge $v_1 \rightarrow v_2 \in E$ if there is a data or control dependence from v_1 to v_2. Statement v_2 is *data dependent* on statement v_1 if v_1 writes to a location in memory, v_2 reads from that location in memory, and the value read by v_2 could potentially be the value written by v_1. Statement v_2 is *control dependent* on statement v_1 if the execution of v_1 controls the number of times that v_2 is executed (e.g., v_1 is the guard of a conditional and v_2 is contained in one branch of that conditional). Information can flow from statement v_1 to statement v_2 if there is a path in the PDG from v_1 to v_2.

In order to classify information flows, we annotate the edges of the PDG to denote how each particular edge was derived from the program. We can broadly classify edges based on whether they correspond to data or control dependencies, but an even finer granularity of classification is useful. We describe and motivate the different possible annotations here.

Data Dependence Annotations We can classify data dependence edges as *strong* or *weak*. A strong data dependence arises between v_1 and v_2 if v_1 writes to a single memory location, v_2 definitely reads from that exact same memory location, and the value it reads is definitely the value written by v_1. A weak data dependence arises between v_1 and v_2 if either v_2 only possibly reads from the same memory location as v_1

writes to, or if that memory location was possibly over-written by another value during the execution between v_1 and v_2. The idea behind this classification is that information flow along strong data dependence edges is more likely to be interesting/relevant than that along weak data dependence edges.

Control Dependence Annotations We can classify control dependence edges as *local* or *non-local*. We can further subdivide non-local control edges into *explicit* or *implicit*. A local control edge arises from structured local control flow, such as conditionals or loops; all other control edges are classified to be non-local. An explicit non-local control edge arises from explicit (i.e., syntactically visible) control-flow jumps in the code, such as a `break` or `continue` instruction inside a loop, or an exception thrown using the `throw` instruction, or returning from a function using a `return` instruction. An implicit non-local control edge arises from implicit (i.e., syntactically invisible) exceptions that can be thrown by various JavaScript instructions (e.g., accessing a property of the `undefined` value, or attempting to call a non-function). It is useful to distinguish these categories; for example, consider line 20 in Figure 6.1, and suppose that the analysis infers `obj` to be potentially `undefined` at this line. Since this statement may raise an implicit exception, the statement at line 21 and all the statements that follow inside the try block are control dependent on the conditional on line 19 (because the conditional evaluation of statement 19 dictates whether or not these

statements execute), causing many additional edges to be added to the PDG. Most of these additional edges are a gross over-approximation of the actual control-flow during program execution. Thus, information flow along local control edges is likely to be more interesting/relevant than that along non-local control edges, and information flow along explicit non-local control edges are in turn likely to be more interesting/relevant than that along implicit non-local control edges.

Amplified Control Finally, we can also classify control edges (independently from the classifications above) as *amplified* or *unamplified*. An amplified control edge is contained within a cycle of the CFG, whereas an unamplified control edge is not. This is interesting for information flow because an unamplified control edge can convey at most one bit of information (i.e., whether a statement is executed or not), whereas an amplified control edge could potentially convey an arbitrary number of bits of information (one for each iteration of the loop or recursive call).

Annotation Grammar From these various classifications, we define the following annotation grammar:

$$ann \in Annotation ::= data \mid control$$

$$data \in DataDep ::= \textbf{data}_{strong} \mid \textbf{data}_{weak}$$

$$control \in CtrlDep ::= ctrl \mid ctrl^{amp}$$

$$ctrl \in Ctrl ::= \textbf{local} \mid \textbf{nonloc}_{exp} \mid \textbf{nonloc}_{imp}$$

The annotated PDG is then a graph (V, E) such that $v \in V$ are the program statements and there is an edge $v_1 \xrightarrow{ann} v_2 \in E$ if there is a data or control dependence from v_1 to v_2 that matches the criteria of annotation ann. The remaining subsections describe how we construct the PDG and assign the appropriate annotations to its edges.

6.3.2 Constructing the Annotated DDG

The first phase of PDG construction creates the Data Dependence Graph, which contains all of the data dependence edges of the eventual PDG. In JavaScript, data dependencies arise from reads and writes to variables and to object properties. For statement v, let $ReadVar(v)$ be the set of variables that v can read from, $ReadProp(v)$ be the set of (object, property) pairs that v can read from, $WriteVar(v)$ be the set of variables that v can write to, and $WriteProp(v)$ be the set of (object, property) pairs that v can write to; these sets are computed from the base analysis described earlier.

Dynamically adding, updating, or deleting a property are all considered object property writes. Recall that the properties in these (object, property) pairs are actually abstract strings representing possibly multiple concrete property names.

Each element of these sets is qualified as *strong* (a definite read or write) or *weak* (a possible read or write). Definite reads/writes occur for a variable when its associated abstract memory location is guaranteed to correspond to a single concrete memory location. Definite reads/writes occur for a (object, property) pair when a similar criterion holds for the object *and* the property abstract string corresponds to a single, exact concrete string. Note that definite writes correspond to strong updates in static analysis, and thus write sets that are qualified to be strong are singleton sets. We use normal set intersection for the $ReadVar(\cdot)$ and $WriteVar(\cdot)$ sets, but for the $ReadProp(\cdot)$ and $WriteProp(\cdot)$ sets we must define a new set intersection operator that accounts for the abstract string property names (which abstractly represent sets of concrete strings). We define the operator $\Cap$ as: $S_1 \Cap S_2 = \{(obj, prop) \mid (obj, prop_1) \in S_1,\ (obj, prop_2) \in S_2,\ prop = prop_1 \sqcap prop_2,\ prop \neq \bot\}$.

There is a DDG edge $v_1 \xrightarrow{\textbf{data}_{strong}} v_2$ if there is a CFG path from v_1 to v_2 and both of the following conditions hold:

- $WriteVar(v_1) \cap ReadVar(v_2) = \{var\}$ and *var* is strong in both sets, or $WriteProp(v_1) \cap ReadProp(v_2)$

$= \{(obj, prop)\}$ and $(obj, prop)$ is strong in both sets. In other words, v_2 definitely reads from the memory location written by v_1.

- There is no statement v_3 along any path from v_1 to v_2 such that $WriteVar(v_1) \cap WriteVar(v_3) \neq \emptyset$ or $WriteProp(v_1) \Cap WriteProp(v_3) \neq \emptyset$, i.e., the value read by v_2 is definitely the value written by v_1.

There is a DDG edge $v_1 \xrightarrow{\textbf{data}_{weak}} v_2$ if there is a CFG path from v_1 to v_2, there is not an edge $v_1 \xrightarrow{\textbf{data}_{strong}} v_2$, and both of the following conditions hold:

- $WriteVar(v_1) \cap ReadVar(v_2) \neq \emptyset$ or $WriteProp(v_1) \Cap ReadProp(v_2) \neq \emptyset$. In other words, v_2 possibly reads from the memory location written by v_1.
- There at least one path from v_1 to v_2 such that for any statement v_3 on that path, $WriteVar(v_1) \cap WriteVar(v_3)$ is empty or contains only weak elements and $WriteProp(v_1) \Cap WriteProp(v_3)$ is empty or contains only weak elements. In other words, the value read by v_2 is possibly the value written by v_1.

Figure 6.1 and Figure 6.2 give an example program and the associated PDG to illustrate these points. The edge $1 \xrightarrow{\textbf{data}_{strong}} 2$ exists because we can determine definitely that the call argument at line 2 refers to the (object, property) pair created at line 1. The edge $1 \xrightarrow{\textbf{data}_{weak}} 3$ exists because (assuming the analysis cannot exactly determine the return value of `getString`) we don't know which property of the object defined at line 1 is being accessed.

```
 1   var data = { url: doc.loc };
 2   send(data.url);
 3   send(data[getString()]);
 4   func();
 5   if (doc.loc == "secret.com")
 6       send(null);
 7   var arr = ["covert.com", "priv.com"/*,...*/];
 8   var i = 0, count = 0;
 9   while(arr[i] && doc.loc != arr[i]) {
10      i++;
11      count++; } // end while
12   send(count);
13   try {
14       if (doc.loc != "hush-hush.com")
15           throw "irrelevant";
16       send(null);
17   } catch(x) {};
18   try {
19       if (doc.loc != "mystic.com")
20           obj.prop = 1;
21       send(null);
22       /* ..... */
23   } catch(x) {}
```

Figure 6.1: An example program to show the various annotations of the PDG. We assume the following for this example: `doc.loc` is the current browser url; the `send` method sends it arguments over the network; the base analysis infers `obj` to either reference an object or `null`; `func` is inferred to be either a callable function or `undefined`; and the call to `getString()` returns an unknown string.

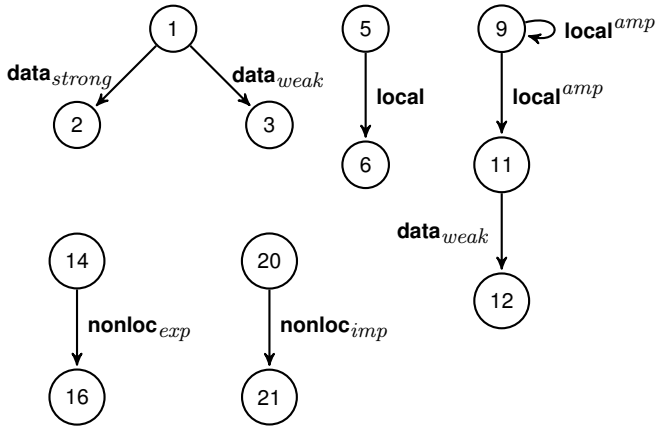

Figure 6.2: A subset of the annotated PDG for the example program in Figure 6.1, to illustrate the interesting edges and nodes.

6.3.3 Constructing the Annotated CDG

The final phase of PDG construction creates the Control Dependence Graph (CDG); the PDG is the union of the DDG and CDG. The CDG is constructed using standard techniques [66], but we stage its construction in order to properly annotate the CDG edges. We also omit from the CDG all edges due to uncaught exceptions (for example, in Figure 6.1, we omit edges due to a potential implicit exception at line 4). If we included those edges, then for all statements that may throw an exception outside of a try/catch block we would need an edge to every other reachable statement in the CFG. For our purposes omitting these edges is sound because uncaught exceptions result in termination, and we are not considering termination leaks in our security analysis.

We construct the annotated CDG in four stages in the following order:

1. Create a pruned CFG by removing all edges arising from non-local control-flow (i.e., exceptions and jumps). Compute CDG_1 from the pruned CFG using standard techniques, and annotate all edges with **local**.

2. Create another pruned CFG from the original CFG by removing all non-local control-flow edges arising from implicit exceptions. Compute CDG_2 from this pruned CFG, subtract any edges present in CDG_1, and annotate all remaining edges with $\textbf{nonloc}_{exp}$.

3. Compute CDG_3 from the full CFG, subtract any edges present in CDG_1 or CDG_2, and annotate all remaining edges with $\textbf{nonloc}_{imp}$.

4. Update all three CDGs so that any annotation $ctrl$ for an edge whose source node is contained within a CFG cycle is updated to $ctrl^{amp}$. The final CDG is $\text{CDG}_1 \cup \text{CDG}_2 \cup \text{CDG}_3$.

When creating a pruned CFG some nodes may become unreachable from the CFG entry node; we add a new edge in the pruned CFG from the entry to any such node before computing the CDG.

In the previous example, the edge $5 \xrightarrow{\textbf{local}} 6$ exists because line 6's execution depends on line 5 but there is no loop, and $9 \xrightarrow{\textbf{local}^{amp}} 11$ exists because line 11's execution depends on line 9 and there is a containing loop. Line 16's execution is control dependent on line 14, because along its `true` branch, the explicit non local control flow at

line 15 can cause line 16 to not execute. Hence the edge $14 \xrightarrow{\textsf{nonloc}_{exp}} 16$. Line 20 can potentially throw an implicit exception, because the base analysis is assumed to infer `obj` to either be a reference an object or `null`. Hence the edge $20 \xrightarrow{\textsf{nonloc}_{imp}} 21$.

6.4 Generating Security Signatures

From the annotated PDG described in the previous section, we can infer interesting information flows to report to the addon vetter and classify them according to types based on the annotations. In this section we describe the form of the signature and how we infer signatures from the annotated PDG.

6.4.1 Description of Security Signatures

Figure 6.3 gives the formal description of a security signature. A signature consists of zero or more entries, where each entry describes either a particular information flow from an interesting source to an interesting sink, or an interesting API usage. API usage is a special case of information flow that indicates there exists some source (interesting or not) that may flow to an instance of that API. The set of interesting sources, sinks, and APIs is given to the analysis; in our implementation we have used the sources, sinks, and APIs considered interesting by the Mozilla vetting team (where the interesting APIs include various script injection APIs such as `Services.scriptloader` and

$$
\begin{aligned}
sign \in Signature &::= \overrightarrow{entry} \\
entry \in Entry &::= src \xrightarrow{type} sink \mid sink \\
type \in FlowType &::= \mathbf{type1} \mid \ldots \mid \mathbf{type8} \\
src \in Source &::= \mathbf{url} \mid \mathbf{key} \mid \mathbf{geoloc} \mid \ldots \\
sink \in Sink &::= \mathbf{send}(Pre) \mid \mathbf{scriptloadr} \mid \ldots
\end{aligned}
$$

Figure 6.3: Grammar for a security signature $sign$. Pre is the prefix string domain described in Section 6.5; it is used to indicate the network domain being communicated with. We give a subset of the complete list of interesting sources and sinks. The eight flow types are described in the text and Figure 6.4.

various deprecated APIs), but they are easily configurable if desired. The **send** sink (corresponding to a network send using `XMLHttpRequest`) takes a parameter indicating the network domain being communicated with. Each information flow entry also has one of eight types, described further below.

An information flow between source and sink is derived from a path in the PDG from the source to the sink. The type of flow is derived from the annotations on the PDG edges along that path. We order the flow types by the kinds of edges (i.e., edges with particular annotations) we allow the associated flow to traverse in the PDG: the more kinds of edges allowed, the weaker the flow type. We have structured the set of flow types into a lattice, pictured in Figure 6.4. Each flow type is associated with an annotation from $Annotation$; the meaning is that a flow of a given type only traverses PDG edges annotated with the given annotation or some annotation at a higher level in

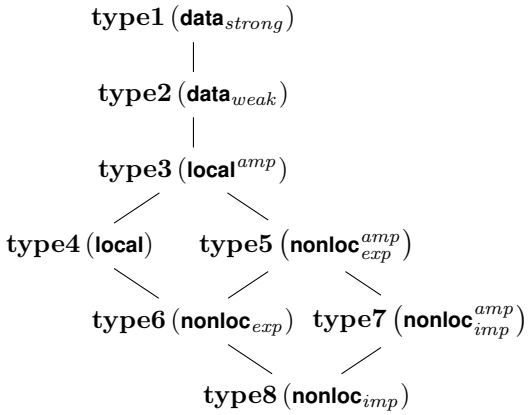

Figure 6.4: Flow types ordered in a lattice of perceived strength. Higher in the lattice indicates a more important type of flow. Each flow type is associated with an annotation from the PDG. A flow has a given type if there is a path from source to sink using only PDG edges annotated with any annotation at a level equal or higher in the lattice.

the lattice. This lattice is based on our perceived strength of the type of flow—obtained by manually examining the commonly intended and commonly accidental kinds of flows. This lattice is the one we use in our analysis, but the lattice is independently configurable to accommodate changes in perceived strength of the flow types.

Consider the examples below to better interpret the various flow types in the lattice in the Figure 6.4. The strongest flow type, **type1**, is assigned to information flows that only traverse PDG edges annotated with **data**$_{strong}$. The **type4** flow type is assigned to information flows that only traverse PDG edges annotated with **local**, **local**^{amp}, **data**$_{weak}$, or **data**$_{strong}$. The weakest flow type, **type8**, is assigned to information flows that traverse any kind of PDG edge. One can think of a particular flow type as corresponding to a sub-graph of the PDG containing only the allowed kinds of edges; an information flow

is assigned that flow type if (1) there is a path from the source to the sink contained in that sub-graph; and (2) there is not a path from the source to the sink in the sub-graph of any higher flow type.

6.4.2 Inferring Signatures

Given an annotated PDG, we must infer signatures of the form described above. Inferring the API usage part of a signature (i.e., is there any information flow to an interesting API) is straightforward: if there a reachable call statement in the CFG whose call expression is data dependent on any node (including itself) with a $ReadProp(\cdot)$ set containing a designated interesting sink *snk*, then the *snk* API may be used. Note that for inferring API usage we consider all call expressions that are data dependent on reads to APIs because functions can be copied and passed around in JavaScript. Inferring the information flow entries of the signature is more involved; the rest of this subsection explains how this is done.

We wish to characterize the set of paths between interesting sources and sinks with a flow type. For each (source, sink) pair there is a set of paths between them in the PDG; we need to compute the strongest flow type(s) possible that are consistent with that set of paths and their edge annotations (because some flow types are noncomparable in strength, there may not be a single strongest flow type). To describe this computation, we first define two helper functions:

$$extend : (FlowType \times Annotation) \to FlowType$$

$$max : \mathcal{P}(FlowType) \to \mathcal{P}(FlowType)$$

The $extend$ function takes a flow type t and extends it with an annotation ann—the function returns the strongest flow type t' which includes all the edge annotations corresponding to the flow type t as well as ann. For example, $extend(\mathbf{type4}, \textsf{nonloc}^{amp}_{exp}) = \mathbf{type6}$, and $extend(\textsf{local}^{amp}, \textsf{nonloc}^{amp}_{exp}) = \mathbf{type5}$. The max function takes a set of flow types and returns the strongest flow types in that set (again, since there are noncomparable flow types there may not be a single strongest flow type in the set). For example, $max(\{\mathbf{type4}, \mathbf{type5}, \mathbf{type6}\}) = \{\mathbf{type4}, \mathbf{type5}\}$.

For each source we will compute a set of flow types for each statement in the PDG reachable from that source; the final set of flow types are taken from the statements corresponding to interesting sinks. Let $FlowType(v)$ be the set of flow types assigned to statement v, and initialize $FlowType(v) = \{\mathbf{type1}\}$ for all statements v. Then compute the fix point over all v of the following equation:

$$FlowType(v) = \max \left(\bigcup_{\substack{v' \xrightarrow{ann} v \in E \\ t \in FlowType(v')}} \{extend(t, ann)\} \right)$$

Intuitively, $FlowType(v)$ gives the strongest set of flow types using which the source under consideration can reach v. To compute this, we look at all the predecessors v' of v, and extend of the flow types computed at v' with the edge annotation ann between v' and v, and keep only the strongest flow types amongst these. Because we consider all the predecessors v' of v and edges between v' and v, we account for all the possible paths from the source to v. Due to the presence of cycles in PDG, we compute a fixpoint of these equations.

Consider the following PDG example to illustrate the flow type equation. Let the PDG include the edges $v_1 \xrightarrow{\textsf{nonloc}^{amp}_{exp}} v_3$ and $v_2 \xrightarrow{\textsf{nonloc}^{amp}_{exp}} v_3$, with $FlowType(v_1) = \{\mathbf{type4}, \mathbf{type5}\}$ and $FlowType(v_2) = \{\mathbf{type3}\}$. To compute $FlowType(v_3)$, we first extend the flow types at predecessors v_1 and v_2 with the corresponding edge annotations, and take their union to obtain $\{\mathbf{type6}, \mathbf{type5}\}$. We then pick the strongest flow types from these to obtain $FlowType(v_3) = \{\mathbf{type5}\}$.

We compute the above fixpoint for the various statements with respect to each interesting source in turn; the signature is created by taking the flow types at each interesting sink. If for source src the sink snk has flow types $\{type_1, type_2\}$, then the signature contains the entries $src \xrightarrow{type_1} snk$ and $src \xrightarrow{type_2} snk$.

6.5 Inferring Network Domains

The most common way in which addons communicate with network domains is to create a network request object `XMLHttpRequest` and pass it a string that contains the desired URL. To generate precise signatures, our analysis should statically infer as many of these URL strings as possible. However, a string constant analysis (analogous to the traditional integer constant analysis) is insufficient to determine many of these strings. Often an addon will communicate with the same domain, but dynamically extend that domain's URL with different suffixes, e.g., different arguments to the same web application. Consider the following code which exemplifies a common pattern found in addons:

```
var baseURL = "www.example.com/req?";
if (...) baseURL += "name"; else baseURL += "age";
// communicate with baseURL
```

A string constant analysis would infer `baseURL` to be an unknown string after the conditional. Our insight is that, for inferring the network domain contained in the string, we only need the URL's prefix rather than the entire URL; e.g., in the example above we need to infer only the base domain `www.example.com/req?` and not the two URLs constructed from that base domain.

Therefore, we augment the base JavaScript analysis (which uses a constant string analysis) with a *prefix string* analysis in order to infer these network domain prefixes. Our abstract prefix string domain is similar in concept to the prefix domain described by Costantini et al. [48], except that we also track exact strings whenever possible—because we use the same string domain for inferring URLs as well as object properties, this is an important distinction for precision. We describe our abstract prefix string domain and one example abstract string operation for that domain, string concatenation. The complete prefix domain formalization and proof sketches of soundness are contained in the supplemental materials.[2]

The prefix string abstract domain is a lattice $\mathcal{L}_p^\sharp = (Pre, \sqsubseteq, \sqcup, \sqcap)$. Let $\preceq$ mean string prefix and let $\oplus$ mean the greatest common prefix; then:

- *Pre* is a set of (string, boolean) pairs augmented with a bottom element: $(str, b) \in Pre = (String \times Boolean) \cup \{\bot\}$, such that $b =$ **true** means str is an exact string and $b =$ **false** means str is a prefix of an unknown string.

- The bottom of the lattice $\bot$ represents an uninitialized string value, and the top of the lattice $\top = (\epsilon,$ **false**$)$ represents all possible strings.

- $\bot \sqsubseteq (str, b) \sqsubseteq \top$ for all $(str, b) \in Pre$, and $(str_1, b_1) \sqsubseteq (str_2, b_2)$ iff either $b_2 =$ **false** and $str_2 \preceq str_1$, or $b_1 =$ **true**, $b_2 =$ **true**, and $str_1 = str_2$

[2] Available under the `Downloads` link at `http://www.cs.ucsb.edu/~pllab`.

- $(str_1, b_1) \sqcup (str_2, b_2) =$

$$\begin{cases} (str_1, b_1) & \text{if } str_1 = str_2, b_1 = b_2 = \textbf{true} \\ (str_1 \oplus str_2, \textbf{false}) & \text{otherwise} \end{cases}$$

- $(str_1, b_1) \sqcap (str_2, b_2) =$

$$\begin{cases} (str_1, b_1) & \text{if } b_2 = \textbf{false},\ str_2 \preceq str_1 \\ (str_2, b_2) & \text{if } b_1 = \textbf{false},\ str_1 \preceq str_2 \\ \bot & \text{otherwise} \end{cases}$$

The lattice is noetherian, i.e., it meets the finite ascending chain condition. We describe the abstract string concatenation operation $+$ on the prefix domain as a representative example of the set of required abstract operations. Let X be any element of $\mathcal{L}_p^\sharp$; then:

- $\bot + X = X + \bot = \bot$
- $(str_1, \textbf{true}) + (str_2, b_2) = (str_1 \cdot str_2, b_2)$
- $(str_1, \textbf{false}) + (str_2, b_2) = (str_1, \textbf{false})$

6.6 Evaluation

In this section we first briefly describe our analysis implementation and our benchmarks and experimental methodology; we then describe and discuss our evaluation results.

6.6.1 Implementation

We implement our signature inference analysis on top of JSAI [96], a flow- and context-sensitive abstract interpreter for JavaScript. JSAI, and hence our analysis, is implemented in Scala. The analysis is performed in three passes: (1) use JSAI to compute the CFG and read/write sets; (2) construct the annotated PDG as described in Section 6.3; and (3) infer the signature as described in Section 6.4.

We extend JSAI in two ways for our analysis. First, we augment JSAI's abstract string domain with the prefix string domain described in Section 6.5. Second, we extend JSAI to handle browser-embedded code: we provide manually-written stubs for the native APIs (e.g., DOM and XPCOM APIs) used by our benchmarks, and simulate the addon event-handling loop by adding a loop at the end of the addon that non-deterministically executes all registered event handlers. Our implementation is available under the `Downloads` link at `http://www.cs.ucsb.edu/~pllab`.

6.6.2 Benchmarks and Methodology

Our benchmark suite consists of real addons taken from the Mozilla addon repository [2]. All of these addons were vetted manually by Mozilla before being added to the repository, and have been present in the repository for years. Table 6.1 lists the addons, their intended purpose, and their size. The size is given as the number of AST nodes parsed by Rhino [1], a more accurate representation than number of lines of code. All of these addons, along with a set of tests showing various kinds of information flows, are bundled with our implementation.

For expository purposes, we classify the addons into three categories based on each addon's summary submitted by its developer:

Category A: Addons intended to explicitly send the current URL information to a specified domain. For example, `LivePageRank`, which sends the active URL over the network to find out its page rank.

Category B: Addons intended to implicitly send information about the current URL or user key presses to a specified domain. For example, `YoutubeDownloader` will check whether the current URL is in fact `youtube.com` before attempting to download a video.

Category C: Addons intended to communicate with a specified domain, but without sending any interesting information. For example, `Chess.comNotifier`

Addon Name	Listed Purpose	Category	Size
`LivePagerank`	Display PageRank for active URL	A	3,900
`LessSpamPlease`	Generates a reusable anonymous real mail address	A	3,696
`YoutubeDownloader`	Youtube video downloader	B	3,755
`VKVideoDownloader`	Downloads videos from sites	B	2,016
`HyperTranslate`	Translates selected text when key shorts are pressed	B	3,576
`Chess.comNotifier`	Notifies your turn on `chess.com`	C	1,079
`CoffeePodsDeals`	Indicates coffee pods for sale	C	1,670
`oDeskJobWatcher`	Indicates oDesk job opening	C	609
`PinPoints`	Save clips (addresses) from web text	C	2,146
`GoogleTransliterate`	Allows user to type in Indian languages	C	4,270

Table 6.1: Real addons from Mozilla addon repository [2] used as benchmarks for our evaluation. We manually sort addons into categories based on their behavior, the category descriptions are given in Section 6.6.2. The size of the benchmarks give the number of AST nodes parsed by Rhino [1].

will communicate with `chess.com` to find out whose turn it is to play. These addons exemplify API usage discovery, using network communication as the API of interest.

In order to check the precision of our inferred signatures, we first manually write a signature for each addon based on its developer-provided summary (this is done *before* we automatically infer any signatures). We can then use the manual signatures to compare against the automatically inferred signatures: if the inferred signatures are weaker (allow more flows) than the manual signature, it indicates either a false positive or a misleading addon summary. We give an example manual signature for one addon in each category:

- `LivePageRank` (A): $\mathbf{url} \xrightarrow{\mathbf{type1}} \mathbf{send}(\texttt{toolbarqueries.google.com})$. Rationale: its stated purpose is to display the page rank of the active URL, computed by sending the URL to `toolbarqueries.google.com`.

- `HyperTranslate` (B): $\mathbf{key} \xrightarrow{\mathbf{type3}} \mathbf{send}(\texttt{translate.google.com})$. Rationale: it translates selected text by using a web service, but only if the keys pressed by the user match its defined keyboard shortcuts. Thus, the addon can implicitly reveal information about key presses to the domain `translate.google.com`. Because the addon continuously listens for key presses, this information flow can be amplified.

- `Chess.comNotifier` (C): $\mathbf{send}(\texttt{chess.com})$. Rationale: it does not reveal information about any interesting sources over the network, but it does communicate with `chess.com` about game status.

We also measure the time taken by the analysis to infer signatures for each benchmark. Our main purpose is to show that the analysis time is reasonable; our prototype implementation is written with emphasis on correctness rather than performance, and there are multiple opportunities for improving the performance of our implementation. We divide the time taken into three phases:

Phase 1 (P1): time taken by the base analysis to compute information assumed as input to our annotated PDG construction.

Phase 2 (P2): time taken to construct the annotate PDG as described in Section 6.3.

Phase 3 (P3): time taken to convert the annotated PDG into a signature as described in Section 6.4.2.

To compute the timing results we run the analysis 11 times on each benchmark, discard the first result, and report the median of the remaining runs. The timing information is obtained on a Mac OS X 2.3 GHz Intel Core i7 machine with 8GB of RAM.

6.6.3 Results and Discussion

Table 6.2 summarizes the result of signature inference analysis on the benchmarks. For each addon, the analysis result is summarized as *pass* (the inferred signature matches the manual signature); *fail* (the inferred signature has more flows than the manual signature, and manual inspection determined they were false positives); or *leak* (the inferred signature has more flows and manual inspection determined they were real). The times are given separately for each analysis phase, as described in Section 6.6.2. The total time taken by the analysis for each of the addons is under one minute.

Five of the addons passed. Of the remainder, two failed and three had unintended leaks. We discuss the failures and leaks in more detail below.

Failed Addons The inferred signatures for `LessSpamPlease` and `VKVideoDownloader` fail simply because the analysis was not able to determine the exact network domain being communicated with. For example, VKVideoDownloader checks whether the current URL is one of three different video player domains, and communicates with the corresponding domain. Our prefix abstract string domain is not expressive enough to precisely represent all three domains, and hence infers the final domain to be unknown. It is worth noting that in the remaining eight out of the ten addons, our prefix string analysis can determine the exact domains with which the

addons communicate. Both failed signatures had the correct information flow sources, sinks, and flow types; the only imprecision was in the network domain.

Leaky Addons `YoutubeDownloader` computes a video id taken directly from the current URL and sends it to `youtube.com`; this is a real explicit information flow. While this is probably an acceptable flow, it was not described in the developer's addon summary and hence was unexpected. `GoogleTransliterate` communicates with the transliterate web API only if the current URL is not `about:blank` (i.e., the empty page); this is an real implicit information flow, though again probably harmless. These examples highlight the usefulness of using security signatures rather than checking against a fixed policy: rather than a simple pass/fail result, the signature allows the addon vetter to easily determine what types of flows are present and whether they are acceptable or not.

`Pinpoints` is an interesting case. Besides communicating with `yourpinpoints.com` (as indicated in the developer summary), it also communicates with `maps.google.com`. It required careful reading of the extended addon description and the addon code to determine that this was actually intended behavior that should have been included in the addon summary (the addon uses information from the Google Maps API to improve the information it saves). This illustrates another benefit of our signature inference, by

Addon Name	Result	Time Taken(s)		
		P1	P2	P3
LivePagerank	pass	15.9	30.3	0.5
LessSpamPlease	*fail*	4.0	24.0	0.1
YoutubeDownloader	**leak**	13.2	22.4	0.2
VKVideoDownloader	*fail*	0.7	8.7	0.1
HyperTranslate	pass	9.6	30.9	0.3
Chess.comNotifier	pass	0.8	2.1	0.1
CoffeePodsDeals	pass	0.4	2.7	0.1
oDeskJobWatcher	pass	0.4	0.9	0.1
PinPoints	**leak**	3.6	16.9	0.1
GoogleTransliterate	**leak**	1.8	10.87	0.1

Table 6.2: Addon signature inference result summary. An addon is marked *pass* if the inferred signature has no more flows than the manual signature; *fail* if it has more flows and they are false positives; and *leak* if it has more flows and they are real. The last three columns indicate the time taken by the inference analysis, divided into three phases as outlined in Section 6.6.2. All times are given in seconds.

highlighting flows that are undocumented or only documented in the addon's fine print.

6.7 Related Work

There have been a number of previous efforts targeting either information flow security, security analysis specific to JavaScript, or browser addon security. In this section we discuss those efforts most relevant to our own work.

Secure Information Flow There are decades of work on secure information flow; for details see the survey by Sabelfeld and Myers [127]. Most of this work is based on type

systems. There is some existing work on using abstract interpretation [43, 55], however they do not target any language nearly as complex and difficult to analyze as JavaScript. Abadi et al. [29] establish a close connection between secure information flow and program slicing using dependencies. Hammer et al. [80] present an information flow analysis for Java bytecode using PDGs. They use a traditional lattice-based approach for their analysis, and apply it to a different language and domain than we do. They also do not attempt to distinguish between the different kinds of information flows.

Security Analysis for JavaScript There have been both static and dynamic (e.g., [83, 86]) approaches to JavaScript analysis; here we focus specifically on those that contain some static component (e.g., [34, 85, 88, 89, 134, 142]), as well as some security component. These analyses target client-side webpage JavaScript programs rather than JavaScript-based browser addons, which present different challenges and opportunities.

Justet al. [91] blend static and dynamic analyses; they track information flow dynamically as much as possible, but resort to static analysis to capture implicit flows. Because of dynamic tracking, their approach requires changes to the JavaScript runtime and incurs an average overhead of 150%.

Guarnieri and Livshits [72] define a statically analyzable subset of JavaScript and implement a tool to enforce certain security and reliability policies on JavaScript widgets. They use dynamic checks to make certain the executing widget code is within the

defined subset language. Their security policy is not formally specified and it is not clear whether they handle only explicit flows or also track implicit flows.

Chugh et al. [47] propose a hybrid mechanism to check certain specific types of malicious information flow in client-side JavaScript. Since client-side JavaScript (unlike browser addons) are allowed to dynamically load new code, they cannot perform a whole-program analysis. Instead, their tool performs a static analysis on all available code and infers a set of dynamic checks necessary to enforce security. Their technique does not scale to more general information flow policies.

Keil and Theimann [99] propose a type-based dependency analysis for JavaScript, and formalize their analysis for a subset of JavaScript. Their analysis can be viewed as static counterpart to data tainting, and they build a tool over the TAJS [89] framework. While not a security analysis, they claim that their analysis could be used as a basis for investigating various security properties.

Browser Addon Security Browser addon security has also attracted much attention. Barth et al. [44] propose a new browser addon architecture (which is now adopted by the Chrome web browser) that reduces the attack surface of addons. They achieve this by separating out addons into components with different privileges and isolating the components by running them in different processes. While Chrome requires the addon to explicitly request access for different privileges, it does not perform any information-

flow based reasoning to figure out what the addons do with accessible information and whether any confidential information is being leaked.

Guha et al. [77] describe IBEX, a framework to develop and verify secure browser addons. IBEX requires developers to write browser addons in a dependently-typed language called Fine. Their tool can statically check if addons conform to policies specified in a Datalog-like policy language, but only if the addons are written in Fine, requiring extensive developer effort.

Dhawan and Ganapathy [58] describe SABRE, a system that guards against Firefox addon security flaws by performing in-browser dynamic information flow tracking of JavaScript addons. SABRE requires extensive modifications to the browser and the execution-time cost of SABRE is high. Djeric and Goel [59] present another dynamic taint tracking analysis for Firefox addons which has similar characteristics. In contrast, we perform a static analysis of the addons; this means that there is no runtime cost and that reviewers can use their discretion to ignore warnings that turn out to be false positives.

Bandhakavi et al. [40] describe VEX, a static tool for highlighting potential security vulnerabilities in Firefox addons. VEX performs an unsound (by design) static taint analysis of JavaScript code (tracking only explicit leaks) with the intent of finding certain types of vulnerability bugs.

Beacon [94] is a static analysis tool to detect capability leaks in Firefox Jetpack extensions (which is a library of modules that makes writing addons much easier). While Beacon detects capability leaks between modules and over-privileged modules, their analysis is unsound by design, and cannot perform information-flow reasoning.

Lerner et. al. [107] present a type-system based approach to verify compliance of JavaScript-based addons with Private-Browsing mode. This requires some annotation effort and cannot perform information-flow reasoning.

6.8 Conclusion

Browser addons written using JavaScript are extremely popular, but they can be easily exploited by malicious developers. We develop a static analysis to automatically infer security signatures for browser addons. Security signatures summarize uses of security critical APIs, as well as interesting information flows augmented with how they occur in addons. These signatures can be used to understand the behavior of addons with regard to security much more easily than having to go through the entire addon source code manually. Inference of security signatures can be employed to automate addon vetting with very little manual intervention. In our evaluation, we demonstrate the usefulness of our strategy by applying our analysis to ten real browser addons from the official Mozilla addon repository.

Chapter 7

Conclusions

JavaScript's popularity has virtually spread to every platform, no longer making it just a language for the web. This has created an urgent need for semantic tools for JavaScript—a static analysis platform for JavaScript can fuel such tools. This thesis shows that our open-source artifact JSAI is a sound, configurable, fast and precise static analysis platform for JavaScript, with formally specified concrete and abstract semantics for JavaScript. We show how to systematically construct abstract interpreters that are widely configurable with arbitrary control-flow sensitivities in a modular fashion, and use these insights in building JSAI. We use type refinement to improve JSAI's precision, and novel parallelization techniques to improve JSAI's performance. We build multiple clients for JavaScript using JSAI, including a security auditing client for vetting browser addons written in JavaScript.

We envision JSAI to form a research platform for easy experimentation with abstract domains, context-, heap-, path-sensitivities and other control-flow sensitivities

that might be novel to JavaScript. JSAI also computes rich amount of semantic information needed to fuel a number of clients. We are currently working on multiple clients (some in collaboration with other teams), including, amongst others, a DOM-based XSS detection client for the web, and a program understanding client with the intent to identify analysis false-posties quickly.

Bibliography

[1] https://developer.mozilla.org/en-US/docs/Rhino.

[2] https://addons.mozilla.org/en-US/firefox/.

[3] http://www.drdobbs.com/windows/microsofts-javascript-move/240012790.

[4] http://nodejs.org/.

[5] http://www.mozilla.org/en-US/firefox/os/.

[6] http://www.khronos.org/registry/typedarray/specs/latest/.

[7] http://www.emscripten.org/.

[8] http://doctorjs.org/.

[9] https://developer.mozilla.org/en-US/docs/SpiderMonkey.

[10] http://www.webkit.org/perf/sunspider/sunspider.html.

[11] http://v8.googlecode.com/svn/data/benchmarks/v7/run.html.

[12] http://linqjs.codeplex.com/.

[13] http://www.defensivejs.com/.

[14] http://aws.amazon.com/.

[15] http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html.

[16] http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentHashMap.html.

[17] http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html.

[18] http://octane-benchmark.googlecode.com/svn/latest/crypto.js.

[19] https://github.com/blueimp/JavaScript-MD5.

[20] https://github.com/mauriciosantos/buckets.

[21] http://github.com/sjkaliski/numbers.js.

[22] https://github.com/doublec/jsparse.

[23] https://blog.mozilla.org/blog/2012/07/26/firefox-add-ons-cross-more-than-3-billion-downloads/.

[24] http://azurit.elbiahosting.sk/ffsniff/.

[25] http://www.subhashdasyam.com/2011/04/mozilla-firefox-strategies-mozilla.html.

[26] LINQ for JavaScript. http://linqjs.codeplex.com/. Accessed: 2013-06-05.

[27] Octane JavaScript Benchmark. http://code.google.com/p/octane-benchmark/. Accessed: 2013-06-05.

[28] T.J. Watson Libraries for Analysis (WALA). http://wala.sf.net. Accessed: 2013-06-05.

[29] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Symposium on Principles of Programming Languages*, 1999.

[30] A. Albarghouthi, R. Kumar, A. V. Nori, and S. K. Rajamani. Parallelizing top-down interprocedural analyses. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[31] M. Allen and S. Horwitz. Slicing java programs that throw and catch exceptions. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, 2003.

[32] AMO Team at Mozilla. Personal Communication, 2011.

[33] J.-h. D. An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic inference of static types for ruby. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2011.

[34] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2005.

[35] G. R. Andrews. *Concurrent programming: principles and practice*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

[36] J. M. Ashley and R. K. Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4), July 1998.

[37] G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. In *International conference on Static Analysis*, 2006.

[38] T. Ball and S. Horwitz. Slicing programs with arbitrary control-flow. In *International Workshop on Automated and Algorithmic Debugging*, 1993.

[39] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2001.

[40] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. VEX: Vetting Browser Extensions for Security Vulnerabilities. In *USENIX Conference on Security*, 2010.

[41] S. Bandhakavi, N. Tiku, W. Pittman, S. T. King, P. Madhusudan, and M. Winslett. Vetting browser extensions for security vulnerabilities with vex. *Commun. ACM*, 54(9), Sept. 2011.

[42] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 1979.

[43] R. Barbuti, C. Bernardeschi, and N. De Francesco. Abstract Interpretation of Operational Semantics for Secure Information Flow. *Inf. Process. Lett.*, 2002.

[44] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting Browsers from Extension Vulnerabilities. In *Annual Network & Distributed System Security Symposium*, 2010.

[45] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2009.

[46] R. Chugh, D. Herman, and R. Jhala. Dependent types for javascript. In *International Conference on Object Oriented Programming Systems Languages and Applications*, 2012.

[47] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[48] G. Costantini, P. Ferrara, and A. Cortesi. Static analysis of string values. In *ICFEM*, 2011.

[49] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM Press, New York, NY, 1977.

[50] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *ACM Symposium on Principles of Programming Languages*, 1979.

[51] P. Cousot and R. Cousot. Invited Talk: Higher Order Abstract Interpretation (and Application to Comportment Analysis Generalizing Strictness, Termination, Projection, and PER Analysis. In *IEEE Computer Society International Conference on Computer Languages*, 1994.

[52] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In *European Symposium on Programming (ESOP)*, 2005.

[53] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astrée analyzer. In *European Symposium on Programming (ESOP)*, 2005.

[54] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[55] N. De Francesco and L. Martini. Abstract interpretation to check secure information flow in programs with input-output security annotations. In *International Conference on Formal Aspects in Security and Trust*, 2006.

[56] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. 2004.

[57] K. Dewey, V. Kashyap, and B. Hardekopf. A parallel abstract interpreter for javascript. In *International Symposium on Code Generation and Optimization*, 2015.

[58] M. Dhawan and V. Ganapathy. Analyzing Information Flow in JavaScript-Based Browser Extensions. In *Annual Computer Security Applications Conference*, 2009.

[59] V. Djeric and A. Goel. Securing Script-based Extensibility in Web Browsers. In *USENIX Conference on Security*, 2010.

[60] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. 1994.

[61] M. B. Dwyer and M. Martin. Practical parallelization: Experience with a complex flow analysis. Technical Report KSU CIS TR 99-4, Kansas State University, 1999.

[62] ECMA. *ECMA-262: ECMAScript Language Specification*. Third edition, Dec. 1999.

[63] M. Edvinsson, J. Lundberg, and W. Löwe. Parallel points-to analysis for multicore machines. 2011.

[64] A. Feldthaus, T. D. Millstein, A. Møller, M. Schäfer, and F. Tip. Tool-supported refactoring for JavaScript. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[65] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for javascript ide services. In *International Conference on Software Engineering*. IEEE Press, 2013.

[66] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, July 1987.

[67] J. Fischer, R. Jhala, and R. Majumdar. Joining dataflow with predicates. In *European Software Engineering Conference*, 2005.

[68] T. Freeman and F. Pfenning. Refinement types for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1991.

[69] M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks. Static type inference for ruby. In *ACM symposium on Applied Computing*, 2009.

[70] P. A. Gardner, S. Maffeis, and G. D. Smith. Towards a program logic for javascript. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2012.

[71] M. Gorbovitski, Y. A. Liu, S. D. Stoller, T. Rothamel, and T. K. Tekle. Alias analysis for optimization of dynamic languages. 2010.

[72] S. Guarnieri and B. Livshits. Gatekeeper: mostly static enforcement of security and reliability policies for javascript code. In *Conference on USENIX security symposium*, 2009.

[73] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable javascript. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2011.

[74] A. Guha, S. Krishnamurthi, and T. Jim. Static analysis for ajax intrusion detection. 2009.

[75] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *World Wide Web Conference*, 2009.

[76] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of javascript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2010.

[77] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *European conference on Programming languages and systems*, 2011.

[78] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *European Symposium on Programming (ESOP)*, 2011.

[79] B. Hackett and S. Guo. Fast and precise hybrid type inference for javascript. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[80] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Secur.*, Oct. 2009.

[81] M. Handjieva and S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *Symposium on Static Analysis (SAS)*, 1998.

[82] B. Hardekopf, B. Wiedermann, B. Churchill, and V. Kashyap. Widening for control-flow. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2014.

[83] D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *IEEE Computer Security Foundations Symposium*, 2012.

[84] P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. *European Conference on Object-Oriented Programming (ECOOP)*, 2010.

[85] D. Jang and K.-M. Choe. Points-to analysis for javascript. In *Symposium on Applied Computing*, 2009.

[86] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *Conference on Computer and Communications Security*, 2010.

[87] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the Eval that Men Do. In *International Symposium on Software Testing and Analysis*, 2012.

[88] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In *European Conference on Foundations of Software Engineering*, 2011.

[89] S. H. Jensen, A. Møller, and P. Thiemann. Type Analysis for Javascript. In *International Symposium on Static Analysis*, 2009.

[90] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural Analysis with Lazy Propagation. In *International Symposium on Static Analysis*, 2010.

[91] S. Just, A. Cleary, B. Shirley, and C. Hammer. Information flow analysis for javascript. In *International Workshop on Programming Language and Systems Technologies for Internet Clients*, 2011.

[92] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7, 1977.

[93] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:309–317, 1977.

[94] R. Karim, M. Dhawan, V. Ganapathy, and C.-c. Shan. An analysis of the mozilla jetpack extension framework. In *European Conference on Object-Oriented Programming*, 2012.

[95] V. Kashyap, K. Dewey, E. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. JSAI: A static analysis platform for javascript. 2014.

[96] V. Kashyap and B. Hardekopf. Security signature inference for javascript-based browser addons. In *Symposium on Code Generation and Optimization*, 2014.

[97] V. Kashyap, J. Sarracino, J. Wagner, B. Wiedermann, and B. Hardekopf. Type refinement for static analysis of javascript. In *Symposium on Dynamic Languages*, 2013.

[98] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2013.

[99] M. Keil and P. Thiemann. Type-based Dependency Analysis for JavaScript. In *ACM Workshop on Programming Languages and Analysis for Security*, 2013.

[100] G. A. Kildall. A unified approach to global program optimization. In *Symposium on Principles of Programming Languages (POPL)*, 1973.

[101] G. A. Kildall. A unified approach to global program optimization. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 1973.

[102] A. Lakhotia, D. R. Boccardo, A. Singh, and A. Manacero. Context-sensitive analysis of obfuscated x86 executables. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 2010.

[103] W. Le and M. L. Soffa. Parallel path-based static analysis. Technical Report CS-2010-6, University of Virginia, 2010.

[104] 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*, 2012.

[105] Y.-F. Lee and B. G. Ryder. A comprehensive approach to parallel data flow analysis. 1992.

[106] Y.-F. Lee, B. G. Ryder, and T. J. Marlowe. Experiences with a parallel algorithm for data flow analysis. *The Journal of Supercomputing*, 1991.

[107] B. S. Lerner, L. Elberty, N. Poole, and S. Krishnamurthi. Verifying web browser extensions' compliance with private-browsing mode. In *ESORICS*, 2013.

[108] R. S. Liverani and N. Freeman. Abusing Firefox Extensions. *Defcon 17*, 2009.

[109] F. Logozzo and H. Venter. Rata: Rapid Atomic Type Analysis by Abstract Interpretation – Application to Javascript Optimization. In *Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, 2010.

[110] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *ACM Symposium on the Foundations of Software Engineering*, Aug. 2013.

[111] S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for javascript. In *Asian Symposium on Programming Languages and Systems*, 2008.

[112] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *European conference on Programming Languages and Systems*, 2005.

[113] M. Méndez-Lojo, M. Burtscher, and K. Pingali. A gpu implementation of inclusion-based points-to analysis. 2012.

[114] M. Méndez-Lojo, A. Mathew, and K. Pingali. Parallel inclusion-based points-to analysis. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2010.

[115] D. L. Metayer and D. Schmidt. Structural operational semantics as a basis for static program analysis. *ACM Computing Surveys*, 28:340–343, 1996.

[116] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. 1996.

[117] J. Midtgaard and T. Jensen. A calculational approach to control-flow analysis by abstract interpretation. In *Symposium on Static Analysis (SAS)*, 2008.

[118] J. Midtgaard and T. P. Jensen. Control-flow analysis of function calls and returns by abstract interpretation. *Information and Computation*, 211(0):49 – 76, 2012.

[119] M. Might and P. Manolios. A posteriori soundness for non-deterministic abstract interpretations. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 260–274, 2009.

[120] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1), Jan. 2005.

[121] D. Monniaux. The parallel implementation of the astrée static analyzer. 2005.

[122] V. Nagaraj and R. Govindarajan. Parallel flow-sensitive pointer analysis by graph-rewriting. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, PACT '13, pages 19–28, Piscataway, NJ, USA, 2013. IEEE Press.

[123] F. Nielson and H. R. Nielson. Interprocedural control flow analysis. In *European Symposium on Programming (ESOP)*, 1999.

[124] F. Pluquet, A. Marot, and R. Wuyts. Fast type reconstruction for dynamically typed programming languages. 2009.

[125] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5), Aug. 2007.

[126] B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. 1986.

[127] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 2003.

[128] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Dynamic determinacy analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2013.

[129] D. A. Schmidt. Natural-Semantics-Based abstract interpretation. In *International Static Analysis Symposium (SAS)*, 1995.

[130] D. A. Schmidt. Abstract interpretation of small-step semantics. *Lecture Notes in Computer Science*, 1192:76–99, 1997.

[131] O. Shivers. *Control-Flow Analysis of Higher-Order Languages, or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

[132] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2011.

[133] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of javascript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2012.

[134] A. Taly, U. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated analysis of security-critical javascript apis. In *IEEE Symposium on Security and Privacy*, 2011.

[135] P. Thiemann. Towards a Type System for Analyzing Javascript Programs. In *European Conference on Programming Languages and Systems*, 2005.

[136] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2008.

[137] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *ACM SIGPLAN International Conference on Functional programming (ICFP)*, 2010.

[138] D. Van Horn and M. Might. Abstracting abstract machines. In *International Conference on Functional Programming*, 2010.

[139] D. Vardoulakis. *CFA2: Pushdown Flow Analysis for Higher-Order Languages*. PhD thesis, Northeastern University, 2012.

[140] S. Weeks, S. Jagannathan, and J. Philbin. A concurrent abstract interpreter. *Lisp and Symbolic Computation*, 1994.

[141] M. Weiser. Program slicing. In *International Conference on Software Engineering*. IEEE Press, 1981.

[142] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript Instrumentation for Browser Security. In *Symposium on Principles of Programming Languages*, 2007.

[143] D. Zanardini. The semantics of abstract program slicing. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, 2008.

[144] T. Zhao. Polymorphic type inference for scripting languages with object extensions. 2011.