JSAI: A Static Analysis Platform for JavaScript

Vineeth Kashyap† Kyle Dewey† Ethan A. Kuefner†
John Wagner† Kevin Gibbons† John Sarracino⋆
Ben Wiedermann⋆ Ben Hardekopf†
† University of California Santa Barbara, USA ⋆ Harvey Mudd College, USA

ABSTRACT

JavaScript is used everywhere from the browser to the server, including desktops and mobile devices. However, the current state of the art in JavaScript static analysis lags far behind that of other languages such as C and Java. Our goal is to help remedy this lack. We describe JSAI, a formally specified, robust abstract interpreter for JavaScript. JSAI uses novel abstract domains to compute a reduced product of type inference, pointer analysis, control-flow analysis, string analysis, and integer and boolean constant propagation. Part of JSAI's novelty is user-configurable analysis sensitivity, i.e., context-, path-, and heap-sensitivity. JSAI is designed to be provably sound with respect to a specific concrete semantics for JavaScript, which has been extensively tested against a commercial JavaScript implementation.

We provide a comprehensive evaluation of JSAI's performance and precision using an extensive benchmark suite, including real-world JavaScript applications, machine generated JavaScript code via Emscripten, and browser addons. We use JSAI's configurability to evaluate a large number of analysis sensitivities (some well-known, some novel) and observe some surprising results that go against common wisdom. These results highlight the usefulness of a configurable analysis platform such as JSAI.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program analysis

General Terms

Languages, Algorithms, Verification

Keywords

JavaScript Analysis, Abstract Interpretation

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 [1]) and server-side applications (e.g., using Node.js [2]), and to develop mobile phone applications (e.g., for Firefox OS [3]). 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 [35] and to experiment with strategies to improve analysis precision [36]. We have also used JSAI to build a static program slicing [48] client and to build a novel abstract slicing [49] 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 2):

- 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 [22] (sans `eval` and family), and various language extensions such as Typed Arrays [4].
- 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

FSE '14, November 16–22, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11 ...$15.00.

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 paper are:

- The design of a JavaScript intermediate language and concrete semantics intended specifically for abstract interpretation (Section 3.1).
- The design of an abstract semantics that enables configurable, sound abstract interpretation for JavaScript (Section 3.2). This abstract semantics represents a 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).
- A discussion of JSAI's configurable analysis sensitivity, including two novel context sensitivities for JavaScript (Section 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 [5], and open-source JavaScript programs (Section 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 2) and conclude with plans for future work (Section 6).

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 [17, 26, 31], 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 [45, 30, 19, 28]. 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 [32, 33, 34]. 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 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.

Various previous works [15, 45, 31, 39, 25, 44, 24] 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 [40, 17, 6, 47, 23] have been proposed, while other works [31, 26] 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 [45, 30, 28, 19] have been proposed to retrofit JavaScript (or subsets thereof) with static types. Guha et. al. [28] propose a novel combination of type systems and flow analysis. Chugh et. al. [19] 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 [25, 20] 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. [42] 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 [41] give a structural smallstep operational semantics directly to the full JavaScript language (omitting a few constructs). Lee et. al [38] 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 [27] 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

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

[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.

code bloat—more than 200× 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 [18] that performs flow-insensitive points-to analysis for Java programs using a declarative specification in Datalog. Several context-sensitive versions [43, 37] 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. 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.

3.1 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 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 **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.

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

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

Figure 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 $n - 1$.

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.

An important design decision we made is to carefully separate the language into pure expressions ($e \in \mathit{Exp}$) that are guaranteed to terminate without throwing an exception, and impure statements ($s \in \mathit{Stmt}$) 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

[3] Guha et al [27] use a similar approach, but our IR design and formal semantics are quite different. See Section 2 for a discussion of the differences between our two approaches.

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

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.

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 [7]. 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.

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

Algorithm 1 The JSAI worklist algorithm

```
put the initial abstract state $\hat{\varsigma}_0$ on the worklist
initialize map partition : $Trace \to State^\sharp$ 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 partition does not contain trace($\hat{\varsigma}'$) then
            partition(trace($\hat{\varsigma}'$)) = $\hat{\varsigma}'$
            put $\hat{\varsigma}'$ on worklist
        else
            $\hat{\varsigma}_{old}$ = partition(trace($\hat{\varsigma}'$))
            $\hat{\varsigma}_{new} = \hat{\varsigma}_{old} \sqcup \hat{\varsigma}'$
            if $\hat{\varsigma}_{new} \neq \hat{\varsigma}_{old}$ then
                partition(trace($\hat{\varsigma}'$)) = $\hat{\varsigma}_{new}$
                put $\hat{\varsigma}_{new}$ on worklist
            end if
        end if
    end for
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] `next_states` $\in State^\sharp \to \mathcal{P}(State^\sharp)$, and the knob that configures the analysis sensitivity `trace`$(\hat{\varsigma})$.

Abstract Semantic Domains. Figure 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 objects, or sets of continuations (to enforce computability for abstract semantics that use semantic continuations, as per Van Horn and Might [46]); 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

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

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

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

Figure 2: Abstract semantic domains for notJS.

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.

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. Composing 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 [21] 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.

Together, all of these abstract domains define a set of simultaneous analyses: control-flow analysis (for each callsite, 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).

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

Figure 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 $\pi_{\hat{b}}(\llbracket e \rrbracket)$ means to evaluate expression e to an abstract base value, then project out its boolean component.

Abstract Transition Rules. Figure 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 [29]. 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 1). The definition of *Trace* is left to the analysis designer; different definitions yield different sensitivities. For example, 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 5.

3.3 Novel Abstract Domains

JSAI allows configurable abstract number and string do-

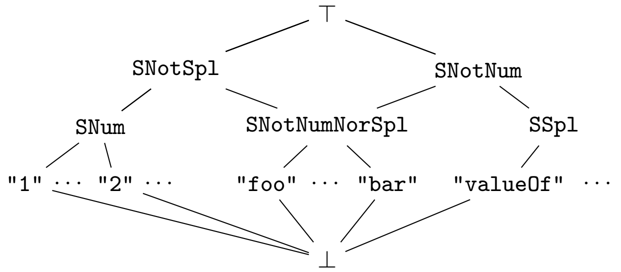


Figure 4: Our default string abstract domain, further explained in Section 3.3.

mains, 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 $String^\sharp$ 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 [33], 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 terminate 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 4 (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.

Abstract Objects. We highlight the abstract domain $Object^\sharp$ given in Figure 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.

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 5; here we define the six main strategies. Four of the six strategies are known in the literature, while two are novel to this paper. 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 hypotheses 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. [43]). 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.

5. EVALUATION

In this section we evaluate JSAI's precision and performance for a range of context-sensitivities as described in Section 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 [33], the most comparable existing JavaScript analysis.

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 [8] and V8 [9] 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 [10]) 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 [8] and V8 [9] (*mean size: 2858 nodes; mean blowup: 8*×); **addon**: seven Firefox browser addons selected from the official Mozilla addon repository [11] (*mean size: 2597 nodes; mean blowup: 6*×); **generated**: seven programs from the Emscripten LLVM test suite, which translates LLVM bitcode to JavaScript [5] (*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.

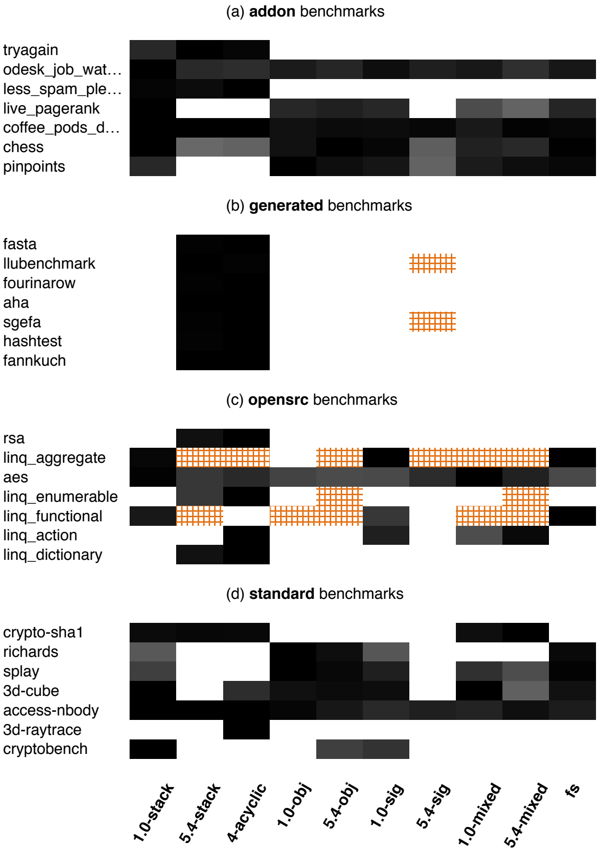


Figure 6: A heat map to showcase the performance characteristics of different sensitivities across the benchmark categories. The above figure is a two-dimensional map of blocks; rows correspond to benchmarks, and columns correspond to analysis run with a particular sensitivity. The color in a block indicates a sensitivities' relative performance on the corresponding benchmark, as compared to fastest sensitivity on that benchmark. Darker colors represent better performance. Completely blackened blocks indicate that the corresponding sensitivity has the fastest analysis time on that benchmark, while completely whitened blocks indicate that the corresponding sensitivity does not time out, but has a relative slowdown of at least 2×. The remaining colors are of evenly decreasing contrast from black to white, representing a slowdown between 1× to 2×. The red grid pattern on a block indicates a timeout.

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**.

Figures 5 and 6 contain performance results, and Figure 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. The performance graphs in Figure 5 plot the median execution time in milliseconds, on a log scale, giving a sense of actual time taken by the various sensitivity strategies. Lower bars are better; timeouts extend above the top of the graph.

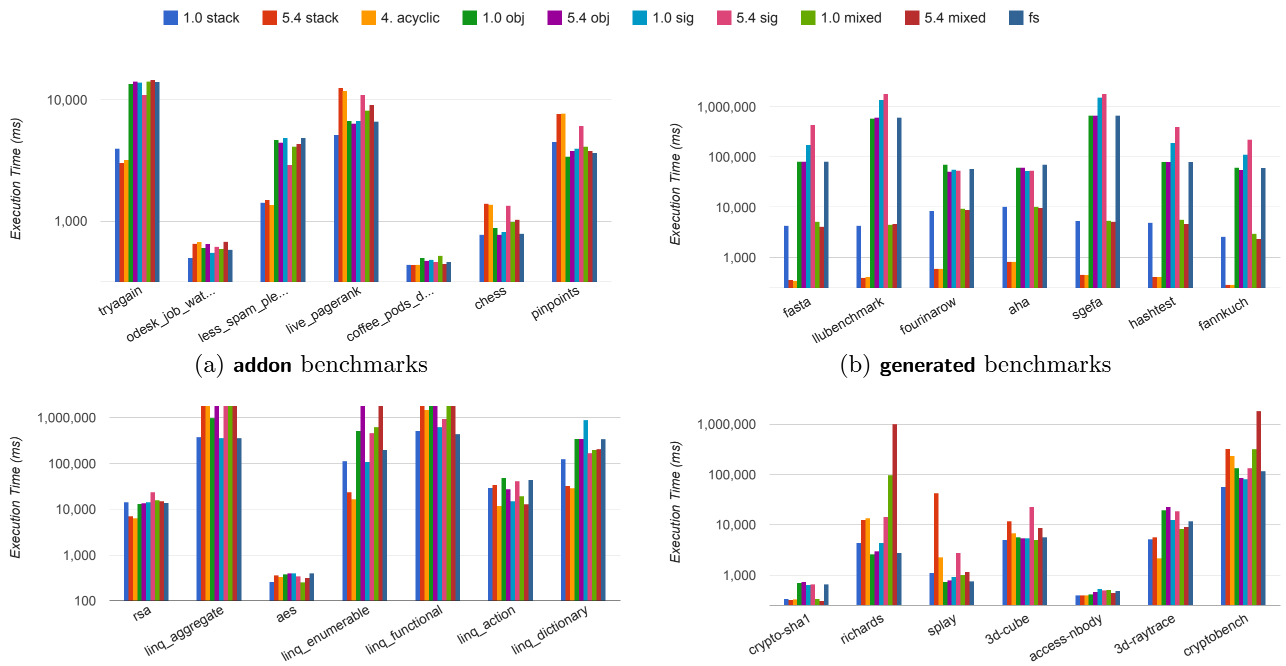
We provide an alternate visualization of the performance data through Figure 6 to easily depict how the sensitivities perform relative to each other. Figure 6 is heat map that lays out blocks in two dimensions—rows represent benchmarks and columns represent analyses with different sensitivities. Each 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 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 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 stop when it becomes intractable. However, our experiments suggest that pushing past this local barrier may yield much better

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



(a) **addon** benchmarks

(b) **generated** benchmarks

(c) **opensrc** benchmarks

(d) **standard** benchmarks

Figure 5: Performance characteristics of different sensitivities across the benchmark categories. The x-axis gives the benchmark names. The y-axis (log scale) gives for each benchmark, the time taken by the analysis (in milliseconds) when run under 10 different sensitivities. Lower bars mean better performance. Timeout (30 minutes) bars are flush with the top of the graph.

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 callstring sensitivity are more efficient and provide more precise results for the open-source benchmarks than object-sensitivity, which often times out.

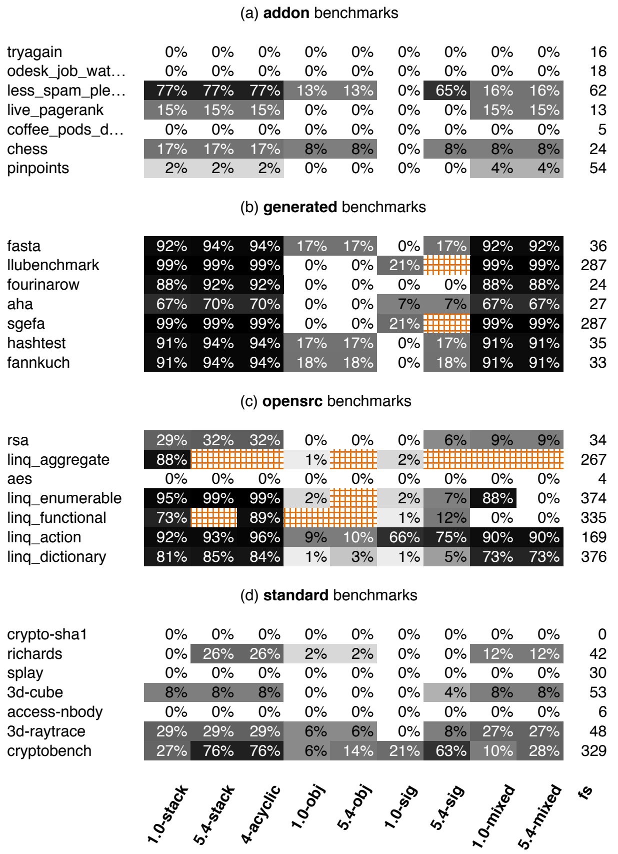
Benefits of Context Sensitivity. When it comes to pure precision, we find that more context sensitivity sometimes increases precision and sometimes has no effect. The open-source benchmarks demonstrate quite a bit of variance for the precision metric. A context-sensitive analysis almost always finds 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.

5.3 Discussion: JSAI *vs.* TAJS

Jensen et al.'s Type Analysis for JavaScript [33, 34] (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,



(a) **addon** benchmarks

	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	fs
tryagain	0%	0%	0%	0%	0%	0%	0%	0%	0%	16
odesk_job_wat…	0%	0%	0%	0%	0%	0%	0%	0%	0%	18
less_spam_ple…	77%	77%	77%	13%	13%	0%	65%	16%	16%	62
live_pagerank	15%	15%	15%	0%	0%	0%	0%	15%	15%	13
coffee_pods_d…	0%	0%	0%	0%	0%	0%	0%	0%	0%	5
chess	17%	17%	17%	8%	8%	0%	8%	8%	8%	24
pinpoints	2%	2%	2%	0%	0%	0%	0%	4%	4%	54

(b) **generated** benchmarks

	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	fs
fasta	92%	94%	94%	17%	17%	0%	17%	92%	92%	36
llubenchmark	99%	99%	99%	0%	0%	21%	▦	99%	99%	287
fourinarow	88%	92%	92%	0%	0%	0%	0%	88%	88%	24
aha	67%	70%	70%	0%	0%	7%	7%	67%	67%	27
sgefa	99%	99%	99%	0%	0%	21%	▦	99%	99%	287
hashtest	91%	94%	94%	17%	17%	0%	17%	91%	91%	35
fannkuch	91%	94%	94%	18%	18%	0%	18%	91%	91%	33

(c) **opensrc** benchmarks

	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	fs
rsa	29%	32%	32%	0%	0%	0%	6%	9%	9%	34
linq_aggregate	88%	▦	▦	1%	▦	2%	▦	▦	▦	267
aes	0%	0%	0%	0%	0%	0%	0%	0%	0%	4
linq_enumerable	95%	99%	99%	2%	▦	2%	7%	88%	0%	374
linq_functional	73%	▦	89%	▦	▦	1%	12%	0%	0%	335
linq_action	92%	93%	96%	9%	10%	66%	75%	90%	90%	169
linq_dictionary	81%	85%	84%	1%	3%	1%	5%	73%	73%	376

(d) **standard** benchmarks

	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	fs
crypto-sha1	0%	0%	0%	0%	0%	0%	0%	0%	0%	0
richards	0%	26%	26%	2%	2%	0%	0%	12%	12%	42
splay	0%	0%	0%	0%	0%	0%	0%	0%	0%	30
3d-cube	8%	8%	8%	0%	0%	0%	4%	8%	8%	53
access-nbody	0%	0%	0%	0%	0%	0%	0%	0%	0%	6
3d-raytrace	29%	29%	29%	6%	6%	0%	8%	27%	27%	48
cryptobench	27%	76%	76%	6%	14%	21%	63%	10%	28%	329

Figure 7: A heat map to showcase the precision characteristics (based on number of reported runtime errors) of different sensitivities across the benchmark categories. The above figure is a two-dimensional map of blocks; rows correspond to benchmarks, and columns corresponds to analysis run with a particular sensitivity. The rightmost column corresponds to the context insensitive analysis **fs**, and the blocks in this column give the number of errors reported by the analysis under **fs** (which is an upper bound on the number of errors reported across any sensitivity). The color (which ranges evenly from black to white) in the remaining blocks indicate the percentage reduction in number of errors reported by the analysis under the corresponding sensitivity, compared to **fs** on the same benchmark. Darker colors represent more reduction in errors reported, and hence better precision. In addition to the colors, the percentage reduction in errors is also given inside the blocks (higher percentage reduction indicates better precision). The red grid pattern on a block indicates a timeout.

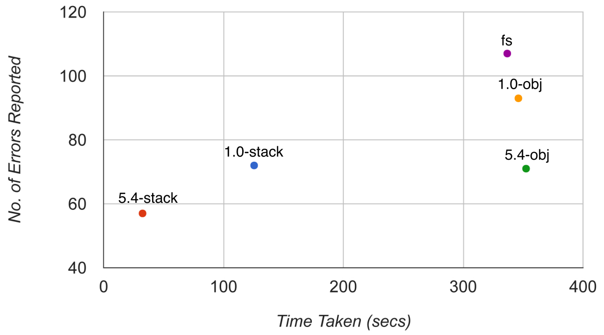


Figure 8: Precision *vs.* performance of various sensitivities, on the **opensrc** `linq_dictionary` benchmark. Interestingly, **5.4-stack** (the most sensitive Stack-CFA analysis) is not only tractable, it exhibits the best performance and the best precision.

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 [16] and lazy propagation [34]) 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× to 1.8× 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.

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.

Acknowledgements. This work was supported by NSF CCF-1319060 and CCF-1117165.

7. REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[15] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In *European conference on Object-oriented programming*, 2005.

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

[17] 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.

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

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

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

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

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

[23] 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.

[24] P. A. Gardner, S. Maffeis, and G. D. Smith. Towards a program logic for javascript. In *ACM Symposium on Principles of programming languages*, 2012.

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

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

[27] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of javascript. In *European conference on Object-oriented programming*, 2010.

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

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

[30] P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. *European conference on Object-oriented programming*, 2010.

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

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

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

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

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

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

[37] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*. ACM, 2013.

[38] 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.

[39] 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.

[40] 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.

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

[42] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Dynamic determinacy analysis. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*. ACM, 2013.

[43] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *ACM Symposium on Principles of programming languages*, 2011.

[44] 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.

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

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

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

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

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