JSAI: A STATIC ANALYSIS PLATFORM FOR JAVASCRIPT

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

PAPER BACKGROUND

- Symposium on Foundations of Software Engineering (FSE) ’14.
  - 22% Acceptance rate (61/273).
- Significant supplemental material.
  - Includes detailed descriptions and their implementation.
- Co-authored by Dr. Ryder’s ex post-doc.

JSAI BACKGROUND

- What is it?
    - Static analysis platform for ECMA 3 JavaScript.
    - Implemented using Scala 2.10.
- What are its goals?
    - Secure, correct, maintainable, fast JS code.
        - Security auditing, error-checking, debugging, optimization, program understanding, etc.
    - Improve JS static analysis relative to static analysis for other languages like C and Java.
        - Difficult due to JS's dynamic nature.
- How is it different?
    - Formally specified abstract and concrete semantics designed for abstract interpretation.
    - Concrete semantics tested against commercial JS engine, and abstract semantics tested against concrete for soundness.
    - Analysis sensitivity (path, context, heap) is user configurable.

JSAI DESIGN OVERVIEW

- Intermediate representation of JS programs (notJS)
- Abstract semantics for notJS
- Novel abstract domains for JS analysis

WHAT IS NOTJS?

- Intermediate JS program representation using a formally-specified translation from JS.
- Possesses formal concrete semantics.
- Based on abstract syntax tree, not a control flow graph.
    - Higher order functions, implicit exceptions, implicit conversation types.
    - Imprecision and unsoundness.
    - Uses small step abstract machine operational semantics to model control-flow.
- Aims to make abstract interpretation simple, precise, and efficient.

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

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

NOTJS ABSTRACT SYNTAX

WHY USE NOTJS?

- "JavaScript's many idiosyncrasies and quirky behaviors motivate the use of formal specifications for both the concrete JavaScript semantics and our abstract analysis semantics."
- Captures JavaScript behaviors.
- Allows them to test against actual JavaScript implementations.
- Allows for configurable sensitivity.

SEMANTICS FOR NOTJS

- Concrete semantics model JS programs by starting with an initial program state (data structure) and applying transformation rules (functions) to continually generate the next state until the program terminates.
- Abstract semantics provide a static analysis which over-approximates the concrete semantics.
- Differences
  - Concrete State/Transformation Rules: Singleton/Deterministic
  - Abstract State/Transformation Rules: Set/Nondeterministic

put the initial abstract state $\hat{\varsigma}_0$ on the worklist
initialize map `partition` : $\mathit{Trace} \to \mathit{State}^\sharp$ 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

JSAI ANALYSIS ALGORITHM

DOMAINS FOR THE ABSTRACT SEMANTICS

- Abstract state consists of:
  - Term - notJS statement or abstract value after evaluating a statement.
  - Environment - Maps variables to sets of addresses.
  - Store - Maps address to abstract values, abstract objects, or sets of continuations.
  - Continuation Stack - Represents the computations still to be performed.
  - Trace - Allows for configureable context sensitivity.
- Abstract values are:
  - Exception/jump values (handles non-local control flow)
  - Base values (represents JavaScript values)
    - Tuple of abstract numbers, booleans strings addresses, null, and undefined.
    - Each component is a lattice, which represents a type the value cannot contain, and represents an analysis.
    - These lattices result from a reduced product of the individual analyses.
- All the domains in conjunction define a set of simultaneous analyses which include control flow, pointer, type inference and extended boolean, number and string constant propagation.

DOMAINS FOR THE ABSTRACT SEMANTICS (CONT.)

- By default, JSAI's abstract string (*String#*) domain is similar to TAJS's, but is configureable along with the abstract number domain.
- Their abstract object domain models objects as tuples containing:
  - A map from property names to values.
  - A list of definitely present properties.
  - (Novel) A map containing class-specific values as well as a record of which specific class this abstract object belongs to.

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

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

ABSTRACT SEMANTIC DOMAINS

ABSTRACT TRANSITION RULES

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* $\textbf{true} \in \pi_{\hat{b}}([\![e]\!])$
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* $\textbf{false} \in \pi_{\hat{b}}([\![e]\!])$

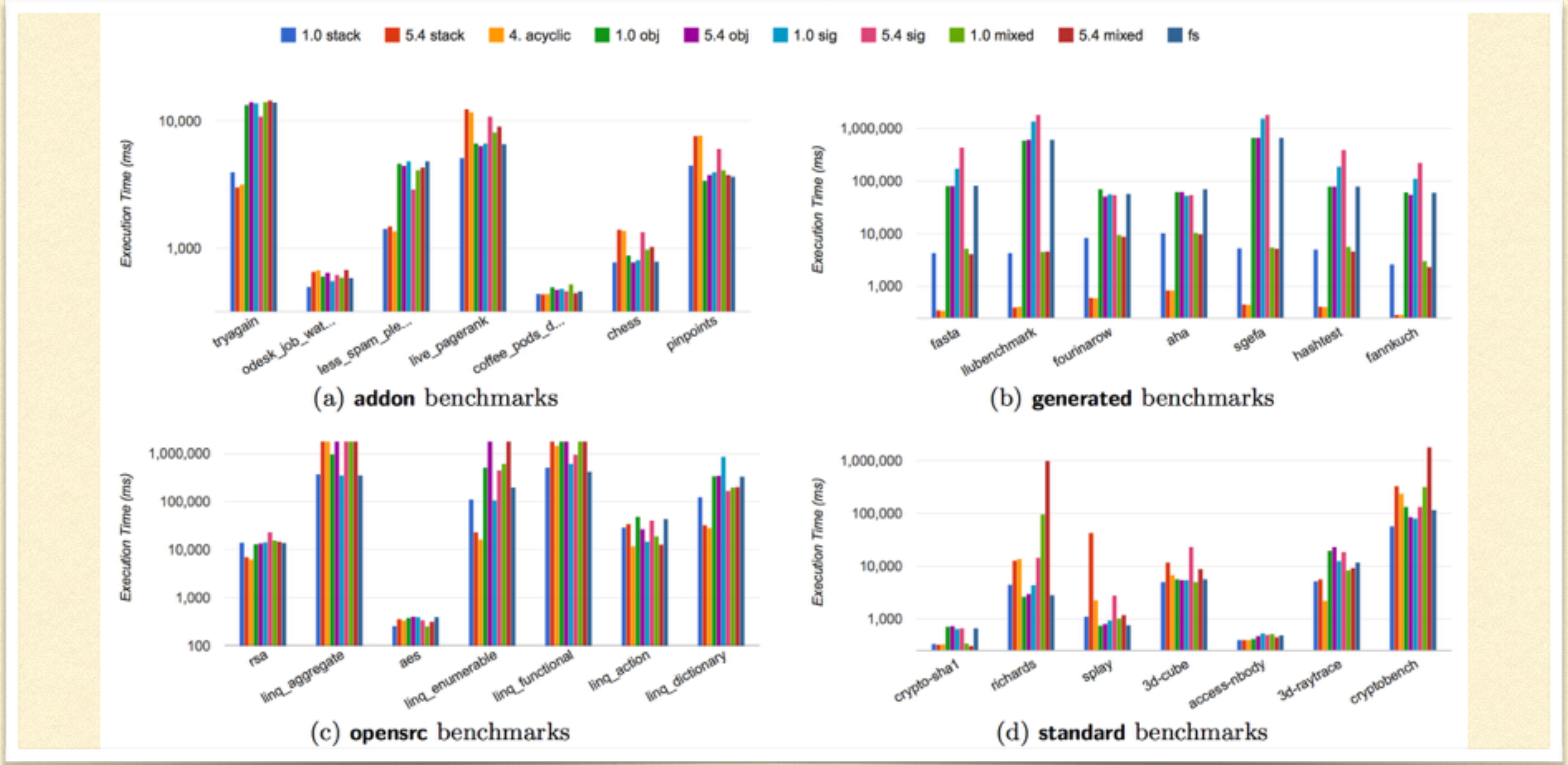
- Describe how to get from a current state to a successor state.
- Nondeterministic.

JSAI CONFIGURABILITY

- Path-, context-, and heap- sensitivities configurable.
- Implemented six main parameterized context sensitivities:
  - Context Insensitive
  - Stack CFA
  - Acyclic CFA
  - Object Sensitive
  - Signature CFA (Novel)
  - Mixed CFA (Novel)

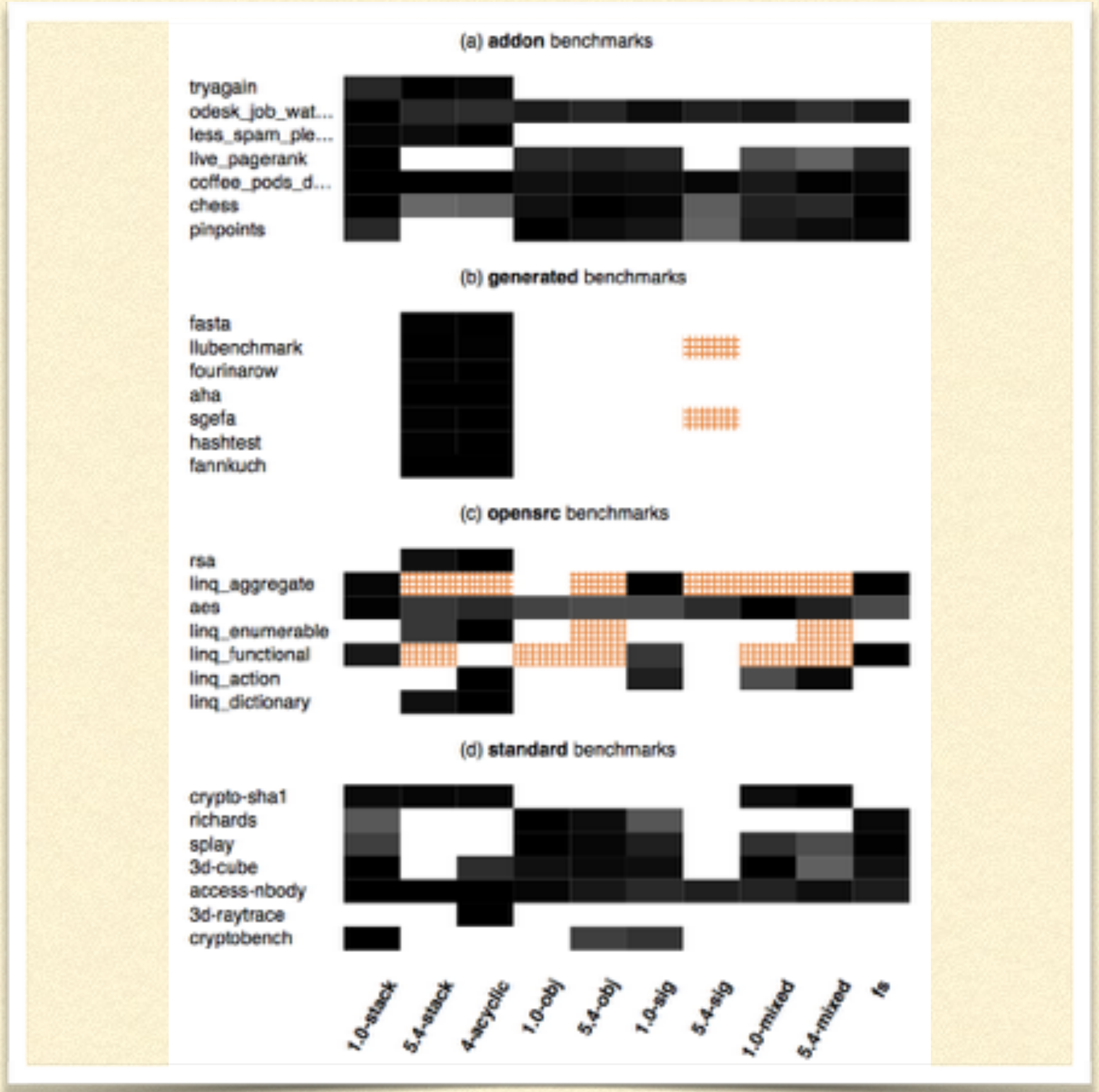
EVALUATION SET-UP

- Independent AWS instances.
  - 15GB memory.
  - 8 ECUs (1.0-1.2 GHz per ECU).
- Tested k.h-stack, h-acyclic, k.h-obj, k.h-sig, and k.h-mixed.
  - k = k-limiting context depth.
  - h = heap sensitivity.
- Benchmark Suites (28 total):
  - standard - Largest and most complex programs from SunSpider and V8.
  - addon - Firefox browser addons from Mozilla repository.
  - generated - Emscripten LLVM test suite generated programs translated to JavaScript
  - opensrc - Open source JavaScript framework programs
- Measured execution time and precision.
  - Performance measured by running each analysis 11 times, discarding the first, and taking the median of the remaining 10.
  - Measure precision by the number of static program locations that might throw exceptions based on type tracking.



PERFORMANCE BAR GRAPH

PERFORMANCE RESULTS

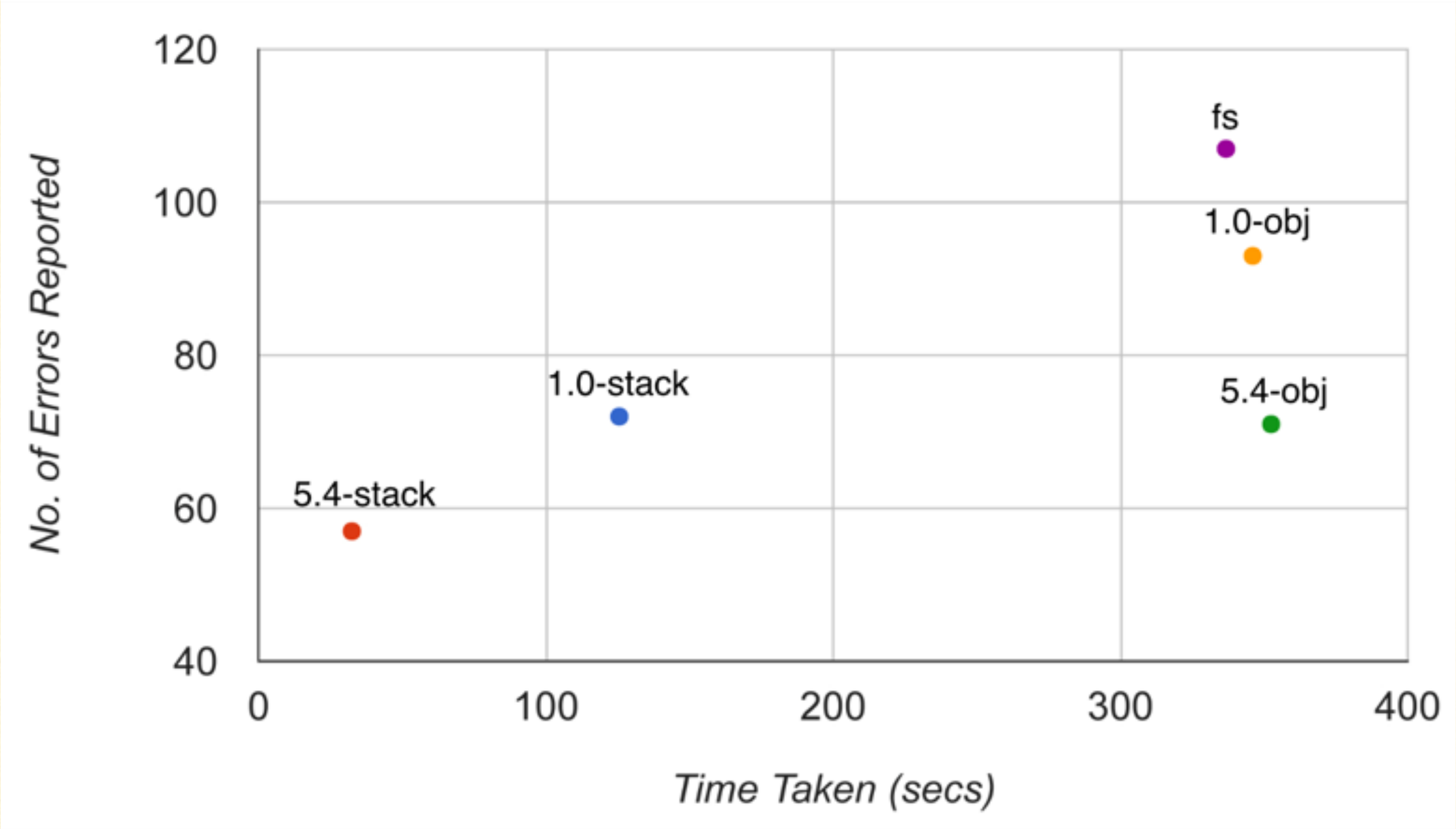


- Higher sensitivities can be more performant than their lower counterparts (5.4-stack on linq_dictionary).
- Common knowledge was that k/h > 2 are unreasonably expensive.
- Trend is not universal.

PRECISION RESULTS

	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
(a) addon benchmarks										
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										
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										
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										
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

- Callstring-based sensitivities (k.h-stack and h-acyclic) were more precise than object sensitivities.
- Most precise and efficient were stack-based k-CFA.
- Partly due to the 1/4 of the benchmark suite being machine-generated.
- Increased sensitivity does not always increase precision.



PRECISION VS. PERFORMANCE

JSAI VS TAJS

- TAJS (Type Analysis for JavaScript) - Only static analysis for JavaScript which can soundly analyze the whole language.
- Features that differentiate JSAI from TAJS:
  - Configurable sensitivity.
  - Formalized abstract semantics.
  - Novel abstract domains.
  - No (discovered) bugs which decrease soundness.
  - Can analyze more benchmark suites.
- Advantages of TAJS:
  - More precise implementation of core JavaScript APIs.
  - Possesses performance and precision optimizations (heap abstraction and lazy propagation).
- Comparison
  - JSAI requires 0.3x to 1.8x more time to run.
  - JSAI reports 9 fewer to 104 more type errors.
  - JSAI reports at most 20 more type errors not including the bug.

CONCLUSIONS

- Main contribution: JSAI, a configurable, sound and efficient abstract interpreter for JS.
- Uses concrete and abstract semantics to model program execution and produce static analyses.
- More precise context sensitive analyses are sometimes the most performant.
- Only similar platform, TAJS, is unsound and difficult to accurately compare the two.