# Automatically Deriving JavaScript Static Analyzers from Specifications using Meta-Level Static Analysis

Jihyeok Park
Oracle Labs
Brisbane, Australia
jihyeok.park@oracle.com

Seungmin An
KAIST
Daejeon, South Korea
h2oche@kaist.ac.kr

Sukyoung Ryu
KAIST
Daejeon, South Korea
sryu.cs@kaist.ac.kr

## ABSTRACT

JavaScript is one of the most dominant programming languages. However, despite its popularity, it is a challenging task to correctly understand the behaviors of JavaScript programs because of their highly dynamic nature. Researchers have developed various static analyzers that strive to conform to ECMA-262, the standard specification of JavaScript. Unfortunately, all the existing JavaScript static analyzers require *manual updates* for new language features. This problem has become more critical since 2015 because the JavaScript language itself rapidly evolves with a yearly release cadence and open development process.

In this paper, we present JSAVER, the first tool that automatically derives JavaScript static analyzers from language specifications. The main idea of our approach is to extract a *definitional interpreter* from ECMA-262 and perform a *meta-level static analysis* with the extracted interpreter. A meta-level static analysis is a novel technique that indirectly analyzes programs by analyzing a definitional interpreter with the programs. We also describe how to indirectly configure abstract domains and analysis sensitivities in a meta-level static analysis. For evaluation, we derived a static analyzer from the latest ECMA-262 (ES12, 2021) using JSAVER. The derived analyzer soundly analyzed all applicable 18,556 official conformance tests with 99.0% of precision in 590 ms on average. In addition, we demonstrate the configurability and adaptability of JSAVER with several case studies.

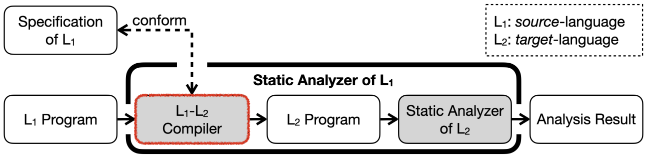
## CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools*; **Software verification and validation**.

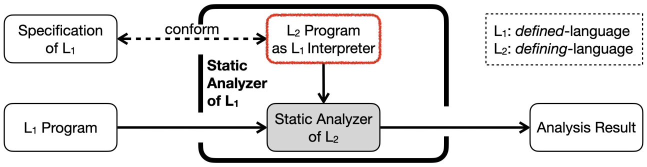
## KEYWORDS

JavaScript, definitional interpreter, meta-level static analysis

**ACM Reference Format:**
Jihyeok Park, Seungmin An, and Sukyoung Ryu. 2022. Automatically Deriving JavaScript Static Analyzers from Specifications using Meta-Level Static Analysis. In *Proceedings of The 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn



**(a) A *compiler*-based approach (existing)**



**(b) An *interpreter*-based approach (ours)**

**Figure 1: Two approaches of static analysis for a language $L_1$ using a static analyzer of another language $L_2$**

## 1 INTRODUCTION

Researchers have presented JavaScript static analyzers to reason about the complex behaviors of JavaScript programs. Existing JavaScript static analyzers, such as JSAI [22], SAFE [27, 44], TAJS [21], and WALA [51], over-approximate the semantics described in ECMA-262, the standard specification of ECMAScript (the official name of JavaScript) written in English. Moreover, various JavaScript static analysis techniques have been presented and implemented on these tools: loop sensitivity [33], advanced string domains [4, 32], analysis based on property relations [25, 31, 51], on-demand backward analysis [52], and combined analysis with dynamic analysis [41, 45, 47, 55].

Existing JavaScript static analyzers take a *compiler*-based approach with intermediate representations (IRs). To reduce the burden of handling numerous language features, most analyzer developers design an IR with a compiler that translates a programming language to its IR to indirectly represent the language semantics [15, 53, 54]. For example, Figure 1(a) depicts a compiler-based approach for static analysis of a *source*-language $L_1$ using a static analyzer of a *target*-language $L_2$. It first compiles an $L_1$ program to an $L_2$ program using an $L_1$-$L_2$ compiler that conforms to the semantics described in the specification of $L_1$. Then, it analyzes the compiled $L_2$ program using a static analyzer of $L_2$. For a JavaScript static analyzer, JavaScript and its own IR are $L_1$ and $L_2$, respectively.

---

*ESEC/FSE 2022, 14 - 18 November, 2022, Singapore*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

### 13.15.2 Runtime Semantics: Evaluation

```
AssignmentExpression : LeftHandSideExpression ||= AssignmentExpression
  1. Let lref be the result of evaluating LeftHandSideExpression.
  2. Let lval be ? GetValue(lref).
  3. Let lbool be ! ToBoolean(lval).
  4. If lbool is true, return lval.
  5. If IsAnonymousFunctionDefinition(AssignmentExpression) is true
     and IsIdentifierRef of LeftHandSideExpression is true, then
       a. Let rval be NamedEvaluation of AssignmentExpression
          with argument lref.[[ReferencedName]].
  6. Else,
       a. Let rref be the result of evaluating AssignmentExpression.
       b. Let rval be ? GetValue(rref).
  7. Perform ? PutValue(lref, rval).
  8. Return rval.
```

name assignments for anonymous functions

(a) **Evaluation algorithm for the logical OR assignment**

```
syntax def AssignmentExpression[8].Evaluation(
  this, LeftHandSideExpression, AssignmentExpression
) { /* entry */
  let lref = (LeftHandSideExpression.Evaluation)
  let lval = [? (GetValue lref)]
  let lbool = [! (ToBoolean lval)] /* #1 */
  if (= lbool true) { /* #2 */ return lval } else {} /* #3 */
  if (&& (IsAnonymousFunctionDefinition AssignmentExpression)
    (LeftHandSideExpression.IsIdentifierRef)) { /* #4 */
    let rval = (AssignmentExpression.NamedEvaluation
               lref.ReferencedName)
  } else { /* #5 */
    let rref = (AssignmentExpression.Evaluation)
    let rval = [? (GetValue rref)]
  } /* #6 */
  [? (PutValue lref rval)]
  return rval
} /* exit */
```

**(b) Extracted $\text{IR}_{\text{ES}}$ function for the logical OR assignment**

**Figure 2: Evaluation algorithm for the eighth alternative of *AssignmentExpression* in ES12 and its extracted $\text{IR}_{\text{ES}}$ function**

However, static analyzers with the compiler-based approach are unable to keep up with fast-evolving JavaScript because they require *manual updates* for new language semantics. The JavaScript language itself is rapidly evolving nowadays. Since 2015, the Ecma Technical Committee 39 (TC39) has maintained the specification as an open-source GitHub project and released its official versions annually. The specification size has been getting bigger as well, and the latest version of ECMA-262 (ES12, 2021) [20] is 879 pages. Since existing JavaScript static analyzers cannot update JavaScript-IR compilers automatically, they still focus on ES5.1 and only support a few ES6 features manually. Because recent JavaScript programs often use new features like arrow functions, and promises, the lack of their support becomes increasingly problematic.

To alleviate this problem, we present JSAVER, a JavaScript Static Analyzer via ECMAScript Representations. It is the first tool that automatically derives JavaScript static analyzers from language specifications. The main idea of JSAVER is to shift the paradigm from *compiler*-based approaches to *interpreter*-based ones to utilize "the interpreter-based nature" of JavaScript. The history of JavaScript [56] testifies that the working group designing JavaScript in the 1990s defined the semantics using reference interpreters:

> Guy Steele would ask a question about some edge-case feature behavior. [...] they would each turn to their respective implementation and try a test case. If they got the same answer, that became the specified behavior.

The interpreter-based nature also affects the writing style of the specifications. ECMA-262 describes the language semantics with pseudocode algorithms consisting of sequentially numbered steps to represent program executions. To fully utilize this interpreter-based nature of JavaScript, JSAVER derives a static analyzer by 1) extracting a *definitional interpreter* [46] from ECMA-262 and 2) performing a *meta-level static analysis* with the extracted interpreter.

First, JSAVER extracts definitional interpreters from ECMAScript language specifications. A definitional interpreter provides a way to represent the language semantics of a *defined*-language using its interpreter written in a *defining*-language. We extract a JavaScript definitional interpreter from ECMA-262 using JISET [42], which automatically extracts a definitional interpreter from ECMA-262 taking advantage of its writing style. In the extracted definitional interpreter, the defined-language is JavaScript, and the defining-language is $\text{IR}_{\text{ES}}$, which is an intermediate representation for ECMAScript language specifications. JISET shows its adaptability by extracting definitional interpreters from future versions of ECMA-262 without extending $\text{IR}_{\text{ES}}$.

Then, we present a meta-level static analysis to analyze JavaScript programs indirectly using the extracted interpreters. A meta-level static analysis is an interpreter-based approach for static analysis of a *defined*-language $L_1$ using a static analyzer of a *defining*-language $L_2$ as depicted in Figure 1(b). Since an $L_1$ interpreter is an $L_2$ program, it indirectly analyzes an $L_1$ program by analyzing the interpreter using a static analyzer of $L_2$ with the $L_1$ program as the input. Thus, we develop a static analyzer of $\text{IR}_{\text{ES}}$ for a meta-level static analysis for JavaScript and experimentally show that it can indirectly analyze JavaScript programs effectively. Moreover, for its expressivenss, we present ways to indirectly configure *abstract domains* and *analysis sensitivities* for JavaScript in the static analysis of $\text{IR}_{\text{ES}}$. First, we provide a method to configure abstract domains for JavaScript values and structures. Second, we present the *AST sensitivity* to express analysis sensitivities for JavaScript such as flow-sensitivity and $k$-callsite-sensitivity.

The contributions of this paper are as follows:

- We propose a novel *meta-level static analysis* technique. It indirectly analyzes a *defined*-language program by analyzing its *definitional interpreter* using a static analyzer of the *defining*-language with the program as the input.
- We present JSAVER, the first tool that derives JavaScript static analyzers from language specifications by 1) extracting a definitional interpreter from ECMA-262 and 2) performing a meta-level static analysis with the extracted interpreter.
- We derive a static analyzer $\text{JSA}_{\text{ES12}}$ from the latest ECMA-262, ES12, to evaluate JSAVER. The derived analyzer $\text{JSA}_{\text{ES12}}$ soundly analyzes all applicable 18,556 official conformance tests with 99.0% of precision in 590 ms on average. Moreover, we demonstrate the configurability and adaptability of JSAVER with several case studies.

```
1 let f = /* a random integer from 0 to 99 */;
2 f ||= x => x;    // f: {name: "f", ...} or [1, 99]
3 let y = f.name; // x: "f" or undefined
```

**Figure 3: JavaScript code using the logical OR assignment**

## 2 BACKGROUND

In this section, we briefly explain ECMA-262 and introduce JISET, which extracts a definitional interpreter from ECMA-262. Since we perform meta-level static analysis for JavaScript using extracted definitional interpreters, it is essential to understand how ECMA-262 describes the JavaScript semantics and how JISET extracts a definitional interpreter from it.

As a running example, we use the "logical OR assignment" introduced in ES12. Figure 2(a) shows its semantics described as an algorithm in English, Figure 2(b) shows an $\text{IR}_{\text{ES}}$ function extracted from the algorithm, and Figure 3 presents an example JavaScript program using a logical OR assignment.

### 2.1 JavaScript Semantics in ECMA-262

ECMA-262 is the official specification of JavaScript, which describes its syntax in a variant of the extended Backus–Naur form (EBNF) and its semantics as algorithms in English. For example, consider the example code in Figure 3. It uses the logical OR assignment newly introduced in ES12. Its syntax is defined by the eighth of nine alternatives of the syntactic production of *AssignmentExpression*, and their semantics is defined by the algorithm in Figure 2(a). The algorithm first evaluates *LeftHandSideExpression* to get a reference *lref* and its value *lval* in steps 1 and 2, respectively. Then, it checks its boolean value *lbool* for short-circuiting in steps 3-4. In step 5, if the left- and right-hand-side is an identifier and an anonymous function, it defines the name of the function as the identifier name in step 5-a. In step 6, otherwise, the algorithm evaluates the right-hand-side expression to the value *rval*. It then puts *rval* to the reference *lref* and returns *rval*.

While the operator seems to be the same as combining the logical OR operator (`||`) with the assignment operator (`=`), they have different semantics. Consider the example code. It first defines a variable `f` with a random integer from `0` to `99`. Then, it uses a logical OR assignment to update `f` with an arrow function whose name becomes `"f"` only if `f`'s value is `0` because `0` represents **`false`**, but the other integers represent **`true`**. Finally, it defines a variable `y` with `f.name`, whose value is `"f"` if `f`'s value is the arrow function, but `undefined`, otherwise. If the statement on line 2 is `f = f || (x => x);`, the value of `y` is `undefined` or `""` instead of `"f"`. Thus, to construct a sound static analyzer, one should consider such detailed semantics by referring to all the algorithms in ECMA-262.

ECMA-262 uses two kinds of algorithms: *syntax-directed algorithms* and *normal algorithms*. A syntax-directed algorithm consists of 1) its corresponding alternative of a syntactic production, 2) its name, 3) parameters, and 4) body steps. For example, the algorithm in Figure 2(a) is a syntax-directed algorithm consisting of the eighth alternative of *AssignmentExpression*, Evaluation as its name, no parameters, and the body consisting of eight steps. Unlike syntax-directed algorithms, a normal algorithm is defined with only its name, parameters, and body steps. Their invocations are like function calls with parentheses: GetValue(*lref*) in step 2. Finally, each algorithm always returns a *completion record* to handle different kinds of JavaScript control flows. The prefixes "?" or "!" converts them to their containing values with or without checking for abrupt completions, respectively.

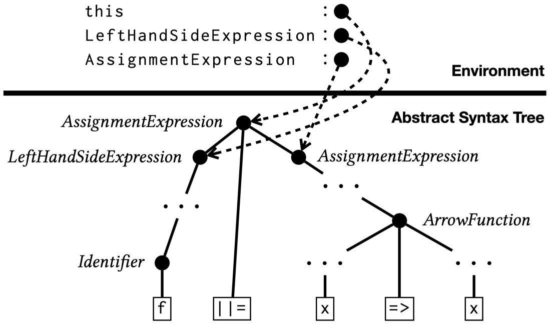


**Figure 4: Result of `f ||= x => x` in a definitional interpreter**

### 2.2 JavaScript Definitional Interpreter

Several researchers have presented JavaScript *definitional interpreters* [2, 3, 6, 7, 17, 42] instead of the compiler-based approaches [14, 16, 21, 27, 35]. A definitional interpreter is written in a *defining*-language to describe the language semantics of a *defined*-language. Among them, we utilize JISET [42] to automatically extract a definitional interpreter from a given version of ECMA-262. The tool JISET 1) generates a parser for syntax and 2) transforms algorithms to corresponding $\text{IR}_{\text{ES}}$ functions for semantics. For example, when JISET takes ES12 as an input, it generates a parser that supports logical OR assignments according to the syntactic production of *AssignmentExpression*. It then transforms the syntax-directed algorithm in Figure 2(a) into the $\text{IR}_{\text{ES}}$ function in Figure 2(b).

The defining-language of a definitional interpreter often treats *abstract syntax trees* (ASTs) of the defined-language as values. The defining-language $\text{IR}_{\text{ES}}$ also treats ASTs of the defined-language JavaScript as its values. For example, the parser generated from ES12 parses the second statement in Figure 3 and produces an AST shown at the bottom of Figure 4. Then, the extracted $\text{IR}_{\text{ES}}$ function in Figure 2(b) takes the AST and its left and right subtrees as its arguments and defines three local variables as shown at the top of Figure 4.

## 3 OVERVIEW

In this section, we explain the overall structure of JSAVER as depicted in Figure 5. It performs a *meta-level static analysis* with JavaScript as its *defined*-language and $\text{IR}_{\text{ES}}$ as its *defining*-language. Thus, JSAVER indirectly analyzes a JavaScript program by analyzing $\text{IR}_{\text{ES}}$ functions with the AST of the program as an argument. For a more detailed explanation, we describe how it performs a meta-level static analysis for the code in Figure 3 with ES12.

JSAVER first utilizes JISET to extract a definitional interpreter from ES12. As explained in Section 2, it generates a JavaScript parser supporting new language features, including the logical OR assignment, and extracts $\text{IR}_{\text{ES}}$ functions, including the function in Figure 2(b), by compiling algorithms. The generated parser parses the example code to produce an AST, which contains the AST shown

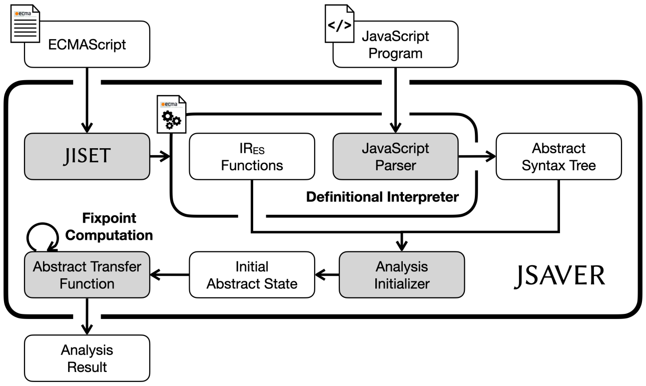


**Figure 5: Overall structure of JSAVER**

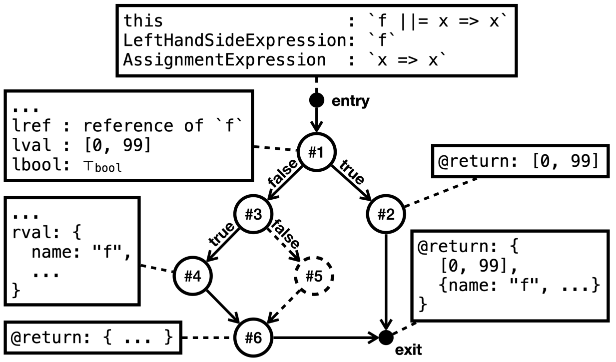


**Figure 6: Control flow graph of the IR$_{\mathrm{ES}}$ function in Figure 2(b) with its flow-sensitive analysis results**

at the bottom of Figure 4 as a subtree. Then, Analysis Initializer constructs an initial abstract state with the extracted IR$_{\mathrm{ES}}$ functions and the produced AST. Finally, JSAVER computes the fixpoint of Abstract Transfer Function with the initial abstract state, and the fixpoint is the analysis result of the example code.

Now, let us explain how we analyze the IR$_{\mathrm{ES}}$ function in Figure 2(b). We support *view-based analysis sensitivities* [24, 40] and utilize a worklist algorithm to perform view-wise updates of analysis results. In this example, we perform flow-sensitive analysis by splitting views based on program points annotated in comments of the IR$_{\mathrm{ES}}$ function: `entry`, `exit`, and from `#1` to `#6`.

Figure 6 shows a control flow graph of the IR$_{\mathrm{ES}}$ function with its flow-sensitive analysis results. In the graph, each node and arrow denotes a program point and a control flow, respectively. If nodes or arrows are dotted, they are unreachable. In this example, we use the interval domain [10] for integers. At the entry point, three parameters point to three ASTs, respectively, as shown at the top of Figure 4. At point `#1`, new local variables are defined: `lref`, `lval`, and `lbool`. Since the variable `LeftHandSideExpression` points to the AST of the JavaScript variable `f`, `lref` points to its reference and `lval` points to the interval `[0, 99]`. Moreover, `lbool` points to the top boolean value $\top_{\mathtt{bool}}$ because `lval` contains `0` representing **`false`** and `[1, 99]` representing **`true`**. Therefore, both points `#2` and `#3` are reachable. At point `#2`, it returns `lval`; thus, the return value `@return` at the exit point becomes `[0, 99]`. At point `#3`, the condition is always true; thus, only point `#4` is reachable, and it assigns a new variable `rval` with a JavaScript function object whose `name` property is a string `"f"`. At point `#6`, it updates the reference of the JavaScript variable `f` with `rval` and returns it. Thus, the return value `@return` at the exit point is merged with the function object stored in `rval`. Finally, the IR$_{\mathrm{ES}}$ function returns the abstract value representing both `[0, 99]` and the JavaScript function object.

Finally, we can automatically derive a JavaScript static analyzer for a specific version of ECMA-262 using JSAVER. For example, if we want to derive a JavaScript static analyzer for ES12, it is sufficient to fix the first argument of JSAVER as ES12 and passes a given JavaScript program as the second argument.

In the remainder of this paper, we formally define the meta-level static analysis for JavaScript with abstract domains and analysis sensitivities (Section 4). Then, we explain how to implement JSAVER with several optimization and analysis techniques (Section 5). After evaluating JSAVER (Section 6), we discuss related work (Section 7) and conclude (Section 8).

## 4 META-LEVEL STATIC ANALYSIS

In this section, we formalize a *meta-level static analysis* for JavaScript as a *defined*-language with IR$_{\mathrm{ES}}$ as a *defining*-language. We first define a JavaScript *definitional interpreter* as an IR$_{\mathrm{ES}}$ program. Then, we define a meta-level static analysis for JavaScript with the abstract semantics of IR$_{\mathrm{ES}}$ in the abstract interpretation framework [9, 11]. In addition, we explain how to indirectly express abstract domains and analysis sensitivities for JavaScript.

### 4.1 JavaScript Definitional Interpreter

We first define IR$_{\mathrm{ES}}$, an Intermediate Representation for ECMA-262, with its collecting and restricted semantics.

$$\begin{array}{l}
\mathfrak{P} \ni P ::= f^* \qquad \mathcal{X} \ni \mathtt{x} \qquad \mathcal{L} \ni \ell \\
\mathcal{F} \ni f ::= \mathtt{syntax}^?\ \mathtt{def}\ \mathtt{x}(\mathtt{x}^*)\ \{[\ell : i]^*\} \\
\mathcal{I} \ni i ::= r := e \mid \mathtt{x} := \{\} \mid \mathtt{x} := e(e^*) \mid \mathtt{if}\ e\ \ell\ \ell \mid \mathtt{return}\ e \\
\mathcal{E} \ni e ::= v^{\mathrm{p}} \mid \mathtt{op}(e^*) \mid r \qquad \mathcal{R} \ni r ::= \mathtt{x} \mid e[e] \mid e[e]_{\mathrm{js}}
\end{array}$$

*4.1.1 Syntax and Notations.* An IR$_{\mathrm{ES}}$ program $P$ is a sequence of functions. A function $f$ is defined with its name, parameters, and body instructions with labels. If it is defined with the prefix `syntax`, it is a syntax-directed function, otherwise, a normal function. An instruction $i$ is a reference update, an object allocation, a function call, a branch, or a return instruction. An expression $e$ is a primitive value, a primitive operation, or a reference expression. A reference is a variable, an internal field access, or an external field access. For a given program $P$, three helper functions $\mathrm{func} : \mathcal{L} \rightarrow \mathcal{F}$, $\mathrm{inst} : \mathcal{L} \rightarrow \mathcal{I}$, and $\mathrm{next} : \mathcal{L} \rightarrow \mathcal{L}$ return the function, instruction, and next label, respectively, of a given label.

$$\begin{array}{lll}
\mathbb{S} = \mathcal{L} \times \mathbb{E} \times \mathbb{C}^* \times \mathbb{H} & & \\
\mathbb{E} = \mathcal{X} \xrightarrow{\text{fin}} \mathbb{V} & \mathbb{C} = \mathcal{L} \times \mathbb{E} & \mathbb{H} = \mathbb{A} \xrightarrow{\text{fin}} \mathcal{L} \times \mathbb{M} \times \mathbb{M}_{\mathrm{js}} \\
\mathbb{M} = \mathbb{V}_{\mathrm{str}} \xrightarrow{\text{fin}} \mathbb{V} & \mathbb{M}_{\mathrm{js}} = \mathbb{V}_{\mathrm{str}} \xrightarrow{\text{fin}} \mathbb{V} & \mathbb{V} = \mathbb{A} \uplus \mathbb{V}^{\mathrm{p}} \uplus \mathbb{T} \uplus \mathcal{F}
\end{array}$$

*4.1.2 Concrete States.* An IR$_{\mathrm{ES}}$ state $\sigma \in \mathbb{S}$ consists of a label, an environment, a stack of calling contexts, and a heap. An environment $\rho \in \mathbb{E}$ is a finite mapping from variables to values. A calling context $c \in \mathbb{C}$ consists of a label and an environment of the caller.

A heap $h \in \mathbb{H}$ is a finite mapping from addresses to labels for allocation sites and two finite mappings from strings to values. The former mapping represents internal fields accessible by $e[e]$, and the latter represents external fields accessible by $e[e]_{\text{js}}$. A value $v \in \mathbb{V}$ is an address, a primitive value (e.g., a boolean $b$, an integer $k$, and a string $s$), a JavaScript AST $t \in \mathbb{T}$, or a function $f \in \mathcal{F}$.

*4.1.3 Restricted Semantics.* We first define a *collecting semantics* $\llbracket P \rrbracket = \lim_{n\to\infty} F^n(\mathbb{S}^\iota)$ using a *transfer function* $F : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$. While we formally define the transfer function $F$ in a companion report [1], we omit it in this paper for brevity. Then, we define a *restricted semantics* $\llbracket P \rrbracket^{\mathbf{R}} : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$ as a set of reachable states from the initial states restricted by a given set of states $S$:

$$\llbracket P \rrbracket^{\mathbf{R}}(S) = \lim_{n\to\infty} F^n(\mathbb{S}^\iota \cap S).$$

*4.1.4 Definitional Interpreter.* We define a *definitional interpreter* for JavaScript as an $\text{IR}_{\text{ES}}$ program to indirectly represent the collecting semantics $\llbracket P_{\text{js}} \rrbracket_{\text{js}}$ of the JavaScript program $P_{\text{js}}$ using the restricted semantics $\llbracket P \rrbracket^{\mathbf{R}}$:

*Definition 4.1 (JavaScript Definitional Interpreter).* An $\text{IR}_{\text{ES}}$ program $P$ is a JavaScript *definitional interpreter* if and only if the following condition holds for each JavaScript program $P_{\text{js}} \in \mathfrak{P}_{\text{js}}$:

$$\llbracket P_{\text{js}} \rrbracket_{\text{js}} = \text{decode} \circ \llbracket P \rrbracket^{\mathbf{R}} \circ \text{encode}(P_{\text{js}})$$

where $\text{encode} : \mathfrak{P}_{\text{js}} \to \mathcal{P}(\mathbb{S})$ encodes a JavaScript program to $\text{IR}_{\text{ES}}$ states and $\text{decode} : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S}_{\text{js}})$ decodes $\text{IR}_{\text{ES}}$ states to JavaScript states.

## 4.2 JavaScript Meta-Level Static Analysis

For a JavaScript *meta-level static analysis*, we define an abstract semantics of $\text{IR}_{\text{ES}}$ in the abstract interpretation framework with *view-based analysis sensitivities* [24, 40].

*4.2.1 Abstract Domains.* We first define the abstract domain for each structure. We define an analysis sensitivity as a *view abstraction* $\delta : \Pi \to \mathcal{P}(\mathbb{S})$, a function from finite *views* to sets of states. Thus, a sensitive abstract state is defined as a function from pairs of labels and views to abstract states:

$$\begin{array}{lll}
\widehat{\mathbb{D}}_\delta = \mathcal{L} \times \Pi \to \widehat{\mathbb{S}} & \widehat{\mathbb{S}} = \widehat{\mathbb{E}} \times \widehat{\mathbb{C}} \times \widehat{\mathbb{H}} & \widehat{\mathbb{A}} = \mathcal{L} \\
\widehat{\mathbb{E}} = \mathcal{X} \to \widehat{\mathbb{V}} & \widehat{\mathbb{C}} = \mathcal{P}(\mathcal{L} \times \Pi) & \widehat{\mathbb{H}} = \widehat{\mathbb{A}} \to \widehat{\mathbb{M}} \times \widehat{\mathbb{M}_{\text{js}}} \\
\widehat{\mathbb{M}} = \mathbb{V}_{\text{str}} \to \widehat{\mathbb{V}} & \widehat{\mathbb{M}_{\text{js}}} = \mathbb{V}_{\text{str}} \to \widehat{\mathbb{V}} & \widehat{\mathbb{V}} = \mathcal{P}(\widehat{\mathbb{A}} \uplus \mathbb{V}^{\text{p}} \uplus \mathbb{T} \uplus \mathcal{F})
\end{array}$$

While we use concrete strings in abstract field maps and sets of primitive values in abstract values in this formalization for brevity, we abstract them to bound the height of their lattices as finite in the implementation. We use *allocation-site abstraction* [8] to define abstract addresses $\widehat{\mathbb{A}}$ as partitions of concrete addresses $\mathbb{A}$ based on their allocation sites $\mathcal{L}$. We define a partial order $\sqsubseteq$, a join operator $\sqcup$, a meet operator $\sqcap$, and a concretization function $\gamma$ for each abstract domain using a *valuation* [12] $\eta : \mathbb{A} \to \widehat{\mathbb{A}}$ to correctly concretize abstract addresses.

*4.2.2 Restricted Abstract Semantics.* We first define the *abstract semantics* $\widehat{\llbracket P \rrbracket} = \lim_{n\to\infty} \widehat{F}^n(\widehat{d^\iota_\delta})$ of an $\text{IR}_{\text{ES}}$ program $P$ with an *initial sensitive abstract state* $\widehat{d^\iota_\delta}$ (i.e., $\mathbb{S}^\iota \subseteq \gamma(\widehat{d^\iota_\delta})$) and an *abstract transfer function* $\widehat{F} : \widehat{\mathbb{D}}_\delta \to \widehat{\mathbb{D}}_\delta$. While we formally define the abstract transfer function $\widehat{F}$ in a companion report [1], we omit them in this paper for brevity. Then, we also define the *restricted abstract semantics* $\widehat{\llbracket P \rrbracket^{\mathbf{R}}} : \widehat{\mathbb{D}}_\delta \to \widehat{\mathbb{D}}_\delta$ of an $\text{IR}_{\text{ES}}$ program $P$ with a given sensitive abstract state $\widehat{d}_\delta$:

$$\widehat{\llbracket P \rrbracket^{\mathbf{R}}}(\widehat{d}_\delta) = \lim_{n\to\infty} \widehat{F}^n(\widehat{d^\iota_\delta} \sqcap \widehat{d}_\delta)$$

*4.2.3 Meta-Level Static Analysis.* Finally, we define a JavaScript meta-level static analysis using the restricted abstract semantics $\widehat{\llbracket P \rrbracket^{\mathbf{R}}}$ of a JavaScript definitional interpreter $P$:

*Definition 4.2 (JavaScript Meta-Level Static Analysis).* A JavaScript *meta-level static analysis* is a way to indirectly analyze a JavaScript program $P_{\text{js}}$ using a restricted abstract semantics $\widehat{\llbracket P \rrbracket^{\mathbf{R}}}$ of a JavaScript definitional interpreter $P$:

$$\llbracket P_{\text{js}} \rrbracket_{\text{js}} \subseteq \widehat{\text{decode}} \circ \widehat{\llbracket P \rrbracket^{\mathbf{R}}} \circ \widehat{\text{encode}}(P_{\text{js}})$$

where $\widehat{\text{encode}} : \mathfrak{P}_{\text{js}} \to \widehat{\mathbb{D}}_\delta$ encodes a JavaScript program to a sensitive abstract state and $\widehat{\text{decode}} : \widehat{\mathbb{D}}_\delta \to \mathcal{P}(\mathbb{S}_{\text{js}})$ decodes a sensitive abstract state to JavaScript states.

## 4.3 Abstract Domains for JavaScript

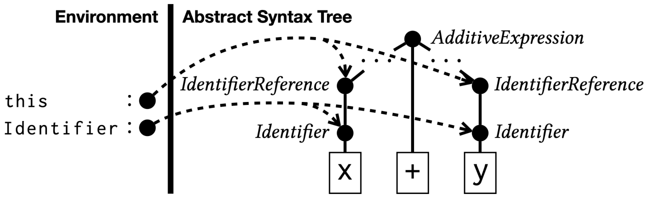
Since the configuration of abstract domains in static analyzers allows fine-tuning the quality of analysis results, we provide a way to indirectly configure abstract domains for JavaScript *values* and *data structures* in a JavaScript meta-level static analysis.

*4.3.1 Values.* Since a JavaScript value is also an $\text{IR}_{\text{ES}}$ value $v \in \mathbb{V}$, we can configure $\widehat{\mathbb{V}}$ for JavaScript values. For example, recall that Figure 6 shows the flow-sensitive analysis results of the code in Figure 3 using the interval domain. Assume that we desire to use the flat domain whose elements are concrete integer values, the bottom value $\perp_{\text{int}}$ for nothing, and the top value $\top_{\text{int}}$ for JavaScript integers. Then, it is sufficient to use the flat domain for integers in the $\text{IR}_{\text{ES}}$ abstract values $\widehat{\mathbb{V}}$. In this setting, the $\text{IR}_{\text{ES}}$ local variable `lval` points to $\top_{\text{int}}$ at point `#1`. At the exit point, the $\text{IR}_{\text{ES}}$ function returns $\top_{\text{int}}$ and the function object whose `name` property is `"f"`.

*4.3.2 Data Structures.* In JavaScript, data structures including environment records and objects have *external* fields directly accessible by JavaScript syntax. For example, an environment record has variables as external fields, accessible by identifier references. Similarly, an object has properties as external fields accessible by property read expressions. However, they also have *internal* fields, which are not directly accessible by JavaScript syntax, and one should update them only indirectly. For example, `[[HasBinding]]` in environment records or `[[Prototype]]` in objects. While such internal fields are pre-defined and the number of possible internal fields is finite, the number of external fields could be infinite. Since internal and external fields are quite different in this regard, we provide a way to configure them differently. In Section 4.2, we define an abstract heap $h \in \mathbb{H}$ as a finite mapping from abstract addresses $\widehat{\mathbb{A}}$ to pairs of abstract internal field maps $\widehat{\mathbb{M}}$ for internal fields and abstract external field maps $\widehat{\mathbb{M}_{\text{js}}}$ for external fields.

```
syntax def IdentifierReference[0].Evaluation(
  this, Identifier
) { return [? (ResolveBinding (Identifier.StringValue))]
    }
```

**(a) Extracted IR$_{\text{ES}}$ function for identifier references**



**(b) Result of `x + y` via a definitional interpreter**

**Figure 7: A JavaScript meta-level static analysis with the flow-sensitivity for IR$_{\text{ES}}$**

## 4.4 Analysis Sensitivities for JavaScript

In a JavaScript meta-level static analysis, analysis sensitivities for JavaScript are different from those for IR$_{\text{ES}}$. Consider the analysis of the following JavaScript code with the flow-sensitivity for IR$_{\text{ES}}$:

```
let x = 1, y = 2;      x + y; // 3
```

Figure 7 shows (a) its extracted IR$_{\text{ES}}$ function and (b) the parsing result of `x + y` and the initial local environment of the IR$_{\text{ES}}$ function. Since the flow-sensitivity merges states on the same labels, contexts for the evaluation of both identifier references `x` and `y` are merged. Thus, the IR$_{\text{ES}}$ variable `Identifier` points to their ASTs as illustrated at the right of Figure 7(b). Due to the imprecise merge of contexts, `StringValue` of `Identifier` returns `"x"` and `"y"`, and `ResolveBinding` with them returns both `1` and `2`. Finally, the analysis result of `x + y` becomes `{ 2, 3, 4 }`.

### 4.4.1 Flow-Sensitivity.

To resolve this problem, we present an *AST sensitivity* for IR$_{\text{ES}}$ as a variant of *object sensitivity* [30, 50] to represent flow-sensitivity for JavaScript. It utilizes JavaScript ASTs $\mathbb{T}$ stored in `this` parameter for syntax-directed functions as views with a view abstraction $\delta^{\text{js-flow}} : \mathbb{T} \uplus \{\bot\} \to \mathcal{P}(\mathbb{S})$:

$$\delta^{\text{js-flow}}(t_\bot) = \{\sigma = (\_, \_, \bar{c}, \_) \in \mathbb{S} \mid \mathsf{ast}(\bar{c}) = t_\bot\}$$

where $\mathsf{ast} : \mathbb{C}^* \to \mathbb{T} \uplus \{\bot\}$ denotes the JavaScript AST stored in `this` parameter of the top-most syntax-directed function for a given calling context stack:

$$\mathsf{ast}(\bar{c}) = \begin{cases} t \;\; \text{if } \exists c.\ \bar{c} = c_1 :: \cdots :: c_n :: c :: \cdots \wedge c = (\ell, \rho) \wedge \\ \qquad \mathsf{func}(\ell) = \mathsf{syntax\ def} \cdots \wedge \rho(\mathtt{this}) = t \wedge \\ \qquad \forall 1 \leq j \leq n.\ c_j = (\ell_j, \_) \wedge \mathsf{func}(\ell_j) = \mathsf{def} \cdots \\ \bot \text{ otherwise} \end{cases}$$

Note that the number of views for the AST sensitivity is finite as well because JavaScript ASTs are finite in a JavaScript program. We define the flow-sensitivity for JavaScript using the AST sensitivity for IR$_{\text{ES}}$. It successfully divides contexts for the evaluation of JavaScript identifiers `x` and `y` in the example even though their labels in IR$_{\text{ES}}$ are the same.

### 4.4.2 Callsite-Sensitivity.

We define the *callsite-sensitivity* [48, 49] for JavaScript by extending the AST sensitivity for specific normal IR$_{\text{ES}}$ functions. In ECMA-262, all explicit and even implicit JavaScript function calls invoke normal IR$_{\text{ES}}$ functions Call and Construct. Thus, we define the callsite-sensitivity for JavaScript by extending the AST sensitivity with two normal IR$_{\text{ES}}$ functions with a view abstraction $\delta^{\text{js-}k\text{-cfa}} : \mathbb{T}^{\leq k} \to \mathcal{P}(\mathbb{S})$:

$$\begin{aligned} \delta^{\text{js-}k\text{-cfa}}([t_1, \cdots, t_n]) = \{&\sigma = (\_, \_, \bar{c}, \_) \in \mathbb{S} \mid \\ &n \leq k \wedge (n = k \vee \mathsf{js\text{-}ctxt}^{n+1}(\bar{c}) = \bot) \wedge \\ &\forall 1 \leq i \leq n.\ \mathsf{ast} \circ \mathsf{js\text{-}ctxt}^{i}(\bar{c}) = t_i\} \end{aligned}$$

where $\mathsf{js\text{-}ctxt} : \mathbb{C}^* \to \mathbb{C}^* \uplus \{\bot\}$ pops out calling contexts until the function of the top-most context is Call or Construct:

$$\mathsf{js\text{-}ctxt}(\bar{c}) = \begin{cases} \bar{c} & \text{if } \bar{c} = (\ell, \rho) :: \_ \wedge \\ & \quad (\mathsf{func}(\ell) = \mathsf{def\ Call} \cdots \vee \\ & \quad\; \mathsf{func}(\ell) = \mathsf{def\ Construct} \cdots) \\ \mathsf{js\text{-}ctxt}(\bar{c}') & \text{if } \bar{c} = \_ :: \bar{c}' \\ \bot & \text{otherwise} \end{cases}$$

Using this callsite-sensitivity for JavaScript, the meta-level static analyzer can discriminate implicit JavaScript function calls, including getters/setters, user-defined implicit conversions, and implicit function calls in built-in libraries.

We also formally define their abstract semantics $\delta^{\text{js-flow}}\widehat{[\![i]\!]}$ and $\delta^{\text{js-}k\text{-cfa}}\widehat{[\![i]\!]}$ in the companion report [1].

## 5 IMPLEMENTATION

In this section, we describe the challenges in implementing a meta-level static analyzer and present our solutions for them. The source code of JSAVER and the dataset of our study are publicly available at https://doi.org/10.5281/zenodo.6785678, and the latest version is maintained as a GitHub repository.[1]

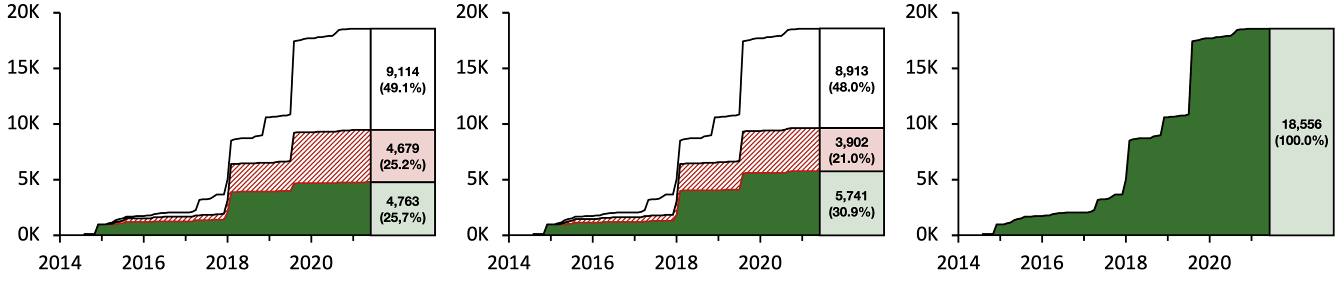
*Layered Abstract States.* Unlike traditional JavaScript static analyses, a meta-level static analysis for JavaScript should track analysis results not only for JavaScript but also for IR$_{\text{ES}}$. Thus, the sizes of abstract states are much larger than those of traditional analyzers. We implement *layered abstract states* to maintain only updated analysis results compared to the initial abstract state. It can reduce the time to perform the join $\sqcup$, meet $\sqcap$, and partial order $\sqsubseteq$ operations by considering only the updated parts in abstract states.

*Heap Cloning and Abstract Counting.* JavaScript Object properties could be dynamically added, modified, or deleted and even accessible by first-class property names. Thus, in JavaScript static analysis, performing *strong updates* rather than *weak updates* for object properties as many as possible is critical for precise analysis results. It becomes more important in our approach because it should track even internal fields for IR$_{\text{ES}}$. Therefore, we implement *heap cloning* [26] and *abstract counting* [29] to increase the chances of performing strong updates for internal and external fields.

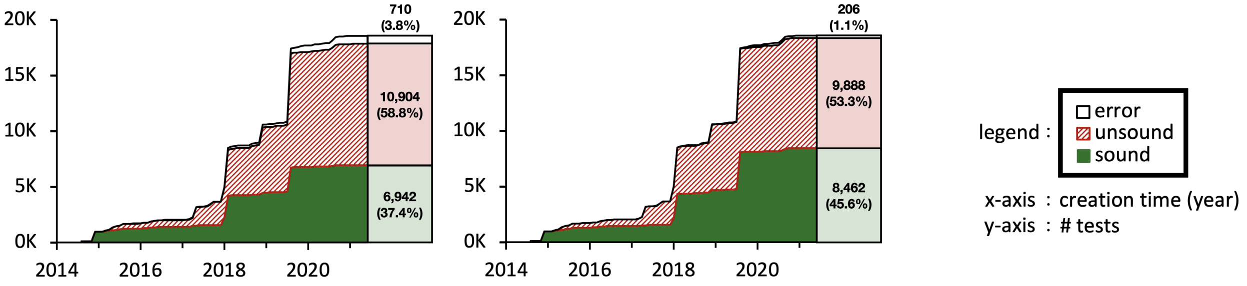
*Loop Sensitivity.* Since merged loop contexts often cause imprecise relations between JavaScript object properties, researchers presented diverse techniques to resolve this problem [25, 31, 51, 52]. Among them, we implement the *loop sensitivity* [33, 34] to increase the analysis precision by discriminating loop contexts. Therefore, derived analyzers via JSAVER can discriminate contexts for explicit loops such as `for-in` and `for-of` and even implicit loops such as the assignment of arguments or the `length` property of arrays.

---

[1] https://github.com/kaist-plrg/jsaver



**(a) Analysis results of TAJS** **(b) Analysis results of SAFE** **(c) Analysis results of JSA$_{\text{ES12}}$**



**(d) Analysis results of TAJS with Babel** **(e) Analysis results of SAFE with Babel**

**Figure 8: Analysis results of TAJS and SAFE without and with Babel and JSA$_{\text{ES12}}$ for 18,556 applicable tests**

**Table 1: Applicable conformance tests in Test262**

| **All Test262 Conformance Tests** | **41,415** |
|---|---|
| **Inapplicable Tests** | **22,859** |
| Web Browsers / Internationalization | 2,036 |
| In-Progress Features | 5,719 |
| Non-Strict / Module | 2,625 |
| Early Errors | 2,949 |
| Inessential Built-in Objects (e.g. `JSON`, `Atomics`) | 9,530 |
| **Applicable Tests** | **18,556** |

## 6 EVALUATION

We evaluate JSAVER using JSA$_{\text{ES12}}$, the JavaScript static analyzer derived from ES12 via JSAVER, with the following research questions:

- **RQ1: Soundness.** Can JSA$_{\text{ES12}}$ analyze JavaScript programs using new language features in a sound way?
- **RQ2: Precision.** Can JSA$_{\text{ES12}}$ precisely analyze JavaScript programs compared to the existing static analyzers?
- **RQ3: Configurability.** Can we configure abstract domains and analysis sensitivities for JavaScript in JSA$_{\text{ES12}}$?
- **RQ4: Adaptability.** Can JSAVER adapt to new language features not yet introduced in ES12?

We performed experiments on an Ubuntu machine equipped with 4.2GHz Quad-Core Intel Core i7 and 32GB of RAM.

### 6.1 Soundness

To evaluate the soundness of JSA$_{\text{ES12}}$, we used Test262, the official conformance test suite. Since ES12 was officially released in June 2021, we used Test262 as of June 2021[2]. While it consists of 41,415 tests, it even contains tests using additional features for web browsers, in-progress features, modules, or early errors for the parsing process. To focus on the core language semantics of JavaScript in ES12, we excluded 22,859 tests for such features, as summarized in Table 1 using JISET. Therefore, we analyzed 18,556 applicable Test262 tests, each of which is 235.5 lines on average. Furthermore, we compared the soundness of JSA$_{\text{ES12}}$ with that of the existing JavaScript static analyzers, TAJS and SAFE. We used their default context sensitivities: the object sensitivity for TAJS and 20-callsite-sensitivity for SAFE. For a fair comparison, we used 20-callsite-sensitivity for JSA$_{\text{ES12}}$ as well.

In addition, we compared the soundness of JSA$_{\text{ES12}}$ with that of the existing analyzers after transpiling Test262 tests via Babel[3], a *hand-written* transpiler from ES6+ to ES5.1. We used the latest Babel v7.17.6 (February 21, 2022). While Babel is often used with core-js[4], a third-party polyfill library implementing ES6+ built-in functions in ES5.1, we did not use core-js in the evaluation because it significantly increases code size. For example, the latest core-js v3.21.1 (February 17, 2022) increases the number of code lines in `harness/sta.js`, which is executed before each Test262 test, from 28 to 3,364. Even before analyzing any Test262 test, TAJS and SAFE failed to analyze `harness/sta.js` in 60 seconds due to the bloated code size. As a result, we used Babel without any polyfill libraries; on average, each transpiled Test262 test is 361.6 lines.

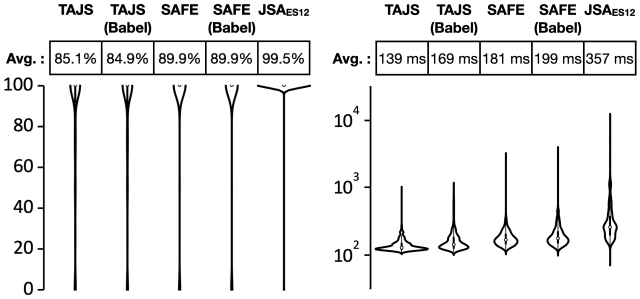
For each test program, we evaluated the soundness of an analyzer by comparing its analysis result with the final state of the program in concrete execution. The *comparison targets* are 1) the reachability of the exit and the exceptional exit points and 2) primitive values stored in variables and object properties at the exit point. We checked whether the analyzer over-approximates the expected values of comparison targets. For example, in the JavaScript

---

[2] https://github.com/tc39/test262/tree/aaf4402b4ca9923012e6

[3] https://babeljs.io/

[4] https://github.com/zloirock/core-js



**(a) The analysis precision** **(b) The analysis performance**

**Figure 9: The analysis precision and performance for 3,878 tests soundly analyzable by all of five analyzers**

program, `let x = 42; x++;`, only the exit point is reachable, and the variable `x` points to `43`. Thus, the analysis result should cover the reachability of the exit point and `43` in `x` for a sound result.

Figure 8 shows the analysis results of existing static analyzers (TAJS and SAFE) without or with Babel and the derived analyzer JSA$_{\text{ES12}}$ for 18,556 applicable tests. In each chart, the $x$-axis denotes when tests are created, and the $y$-axis denotes the number of tests created before the time. The mark sound (green, filled) denotes a sound analysis, unsound (red, stripe) an unsound analysis, and error (white, blank) an unexpected error. Figures 8(a) and 8(b) show that TAJS and SAFE analyzed most tests created before 2015 in a sound way. However, the number of tests that they cannot soundly analyze has consistently increased from 2015. TAJS and SAFE can soundly analyze only 4,763 (25.7%) and 5,741 (30.9%) programs, respectively. As depicted in Figures 8(d) and 8(e), Babel mitigates this problem by transpiling ES6+ features to ES5.1, and it increases the number of programs soundly analyzed by TAJS and SAFE to 6,942 (37.4%) and 8,462 (45.6%), respectively. However, TAJS and SAFE still cannot soundly analyze more than half of Test262 test programs. On the other hand, JSA$_{\text{ES12}}$ successfully analyzes all 18,556 applicable test programs in a sound way, even without Babel.

## 6.2 Precision

We measured the analysis precision by counting how many *comparison targets* were precisely analyzed. For all applicable 18,556 Test262 test programs, JSA$_{\text{ES12}}$ analyzed them with a high analysis precision of 99.0% in 590 ms on average. Then, we compared its analysis precision with that of TAJS and SAFE. For a fair comparison, we measured the analysis precision for 3,878 test programs soundly analyzable by all of five analyzers: TAJS, TAJS with Babel, SAFE, SAFE with Babel, and JSA$_{\text{ES12}}$. Figure 9(a) depicts the average and distribution of the analysis precision in violin plots [19]. TAJS and SAFE analyzed 3,878 test programs with 85.1% and 89.9% precision on average, respectively. While Babel increased the number of test programs soundly analyzed by existing analyzers, it decreased the average analysis precision of TAJS to 84.9% and had no effect on SAFE. It is due to that Babel transpiles simple ES6+ features into a more complex combination of ES5 features even though TAJS directly supports a small part of the ES6 features like arrow functions or `Symbol`. However, JSA$_{\text{ES12}}$ has the highest analysis precision of 99.5% on average.

**Table 2: Definitions of three string abstract domains *String Set* (SS$_k$), *Character Inclusion* (CI), and *Prefix-Suffix* (PS)**

| Domain | Definition | | |
|---|---|---|---|
| SS$_k$ | $\text{SS}_k$ | = | $\{\top\} \cup \{S \subseteq \Sigma^* \mid \lvert S\rvert \leq k\}$ |
| | $\gamma(S)$ | = | $S$ |
| | $S \cdot S'$ | = | $\{s \cdot s' \mid s \in S \land s' \in S'\}$ |
| CI | $\text{CI}$ | = | $\{\bot\} \cup \{[L, U] \mid L, U \subseteq \Sigma \land L \subseteq U\}$ |
| | $\gamma([L, U])$ | = | $\{w \in \Sigma^* \mid L \subseteq \text{chars}(w) \subseteq U\}$ |
| | $[L, U] \cdot [L', U']$ | = | $[L \cup L', U \cup U']$ |
| PS | $\text{PS}$ | = | $\{\bot\} \cup (\Sigma^* \times \Sigma^*)$ |
| | $\gamma(\langle p, s\rangle)$ | = | $\{p \cdot w \mid w \in \Sigma^*\} \cap \{w \cdot s \mid w \in \Sigma^*\}$ |
| | $\langle p, s\rangle \cdot \langle p', s'\rangle$ | = | $\langle p, s'\rangle$ |

```
1 let x = /* "a" or "b" */;
2 let y = `c${x}d`;     // "cad" or "cbd"
3 let z = `${x}e${x}`; // "aea" or "beb"
```

**Figure 10: A JavaScript program using template literals**

On the other hand, the analysis speed of JSA$_{\text{ES12}}$ is slower than that of TAJS and SAFE, and Figure 9(b) depicts them in violin plots on a logarithmic scale. TAJS and SAFE took 139 ms and 181 ms, respectively, to analyze 3,878 test programs on average. Babel increases their average analysis time to 169 ms and 199 ms, respectively, because it transpiles all ES6+ features in test programs to verbose ES5.1 features. However, JSA$_{\text{ES12}}$ took 357 ms on average to analyze them because JSAVER derives precise abstract semantics for all language features. On the contrary, TAJS and SAFE developers often *imprecisely* or even *unsoundly* model the abstract semantics of specific language features to increase the analysis speed. For example, TAJS does not discriminate positive/negative infinity values or positive/negative zeros to reduce the number of possible cases in abstract values. Similarly, SAFE ignores the semantics of getters and setters to analyze object property reads quickly.

## 6.3 Configurability

We demonstrate the configurability of JSAVER with several case studies for abstract domains and analysis sensitivities. We discuss how different abstract domains or analysis sensitivities affect analysis results of JSA$_{\text{ES12}}$ with examples.

### 6.3.1 *Abstract Domains.*

As explained in Section 4.3, we can configure abstract domains for JavaScript values by configuring those for IR$_{\text{ES}}$ values. In JavaScript static analysis, researchers have presented diverse string domains to precisely analyze object property names. Among them, we implemented three representative string abstract domains [4]: the *String Set* (SS$_k$) domain, the *Character Inclusion* (CI) domain, and the *Prefix-Suffix* (PS) domain. Table 2 summarizes formal definitions of their elements, concretization functions, and concatenation operations. In the table, $\Sigma$ denotes a set of characters, and the set of strings is $\mathbb{V}_{\text{str}} = \Sigma^*$. We analyzed a JavaScript program in Figure 10 using JSA$_{\text{ES12}}$ with different string abstract domains. The program uses a new language feature introduced in ES6 called a *template literal*, which is a literal delimited with backticks (`), allowing embedded expressions called *substitutions*. For example, the template literal `` `c${x}d` `` on line 2 concatenates a string `"c"`, the value in the variable `x`, and a string `"d"`. Since `x` points to `"a"` or `"b"` on line 1, the variable `y` points to `"cad"` or `"cbd"`. Similarly, `z` points to `"aea"` or `"beb"` by concatenating `x`, `"e"`, and `x`.

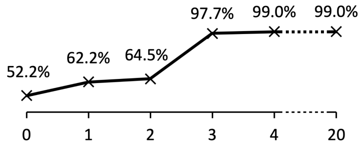


**Figure 11: The Analysis precision of $\text{JSA}_{\text{ES12}}$ with different $k$-callsite-sensitivities for all 18,556 applicable test programs**

First, the *String Set* ($\text{SS}_k$) domain represents a set of strings whose size is bounded by $k$ as an abstract string. Therefore, $\text{JSA}_{\text{ES12}}$ with $\text{SS}_5$ produced the following analysis results:

$$
\begin{aligned}
\mathtt{x} &\mapsto \{\texttt{"a"}, \texttt{"b"}\} \\
\mathtt{y} &\mapsto \{\texttt{"c"}\} \cdot \{\texttt{"a"}, \texttt{"b"}\} \cdot \{\texttt{"d"}\} = \{\texttt{"cad"}, \texttt{"cbd"}\} \\
\mathtt{z} &\mapsto \{\texttt{"a"}, \texttt{"b"}\} \cdot \{\texttt{"e"}\} \cdot \{\texttt{"a"}, \texttt{"b"}\} = \{\texttt{"aea"}, \texttt{"aeb"}, \texttt{"bea"}, \texttt{"beb"}\}
\end{aligned}
$$

It produced precise analysis results for `x` and `y`. However, the result for `z` has spurious values `"aeb"` and `"bea"` because it does not keep the information that the left and right strings of `"e"` are the same.

The *Character Inclusion* (CI) domain tracks the lower and upper bounds of characters occurring in strings. The analysis with this domain produced the following analysis results:

$$
\begin{aligned}
\mathtt{x} &\mapsto [\varnothing, \{\mathsf{a}, \mathsf{b}\}] \\
\mathtt{y} &\mapsto [\{\mathsf{c}\}, \{\mathsf{c}\}] \cdot [\varnothing, \{\mathsf{a}, \mathsf{b}\}] \cdot [\{\mathsf{d}\}, \{\mathsf{d}\}] = [\{\mathsf{c}, \mathsf{d}\}, \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}] \\
\mathtt{z} &\mapsto [\varnothing, \{\mathsf{a}, \mathsf{b}\}] \cdot [\{\mathsf{e}\}, \{\mathsf{e}\}] \cdot [\varnothing, \{\mathsf{a}, \mathsf{b}\}] = [\{\mathsf{e}\}, \{\mathsf{a}, \mathsf{b}, \mathsf{e}\}]
\end{aligned}
$$

This domain ignores structures of strings, but it is a cheap abstract domain to check only the inclusion of characters in strings. For example, it can say that the string in `y` always includes c and d, and the string in `z` always includes e.

The last domain, *Prefix-Suffix* (PS) keeps prefixes and suffixes of strings. $\text{JSA}_{\text{ES12}}$ produced the following analysis results with PS:

$$
\begin{aligned}
\mathtt{x} &\mapsto \langle \texttt{""}, \texttt{""} \rangle \\
\mathtt{y} &\mapsto \langle \texttt{"c"}, \texttt{"c"} \rangle \cdot \langle \texttt{""}, \texttt{""} \rangle \cdot \langle \texttt{"d"}, \texttt{"d"} \rangle = \langle \texttt{"c"}, \texttt{"d"} \rangle \\
\mathtt{z} &\mapsto \langle \texttt{""}, \texttt{""} \rangle \cdot \langle \texttt{"e"}, \texttt{"e"} \rangle \cdot \langle \texttt{""}, \texttt{""} \rangle = \langle \texttt{""}, \texttt{""} \rangle
\end{aligned}
$$

This domain is also cheap but focuses on prefixes and suffixes. Thus, the analysis results cannot say anything about the strings in `x` or `z`, but it describes that the string in `y` starts with `"c"` and ends with `"d"`.

Therefore, we showed that one can freely configure string abstract domains for JavaScript in the derived analyzer $\text{JSA}_{\text{ES12}}$.

### 6.3.2 *Analysis Sensitivities.*

As explained in Section 4.4, we formally define the flow- and $k$-callsite-sensitivity for JavaScript using the AST-sensitivity for $\text{IR}_{\text{ES}}$. In $\text{JSA}_{\text{ES12}}$, we can freely configure the value $k$ of the $k$-callsite-sensitivity. In Section 6.2, we showed that $\text{JSA}_{\text{ES12}}$ with the 20-callsite-sensitivity can precisely analyze 18,556 applicable tests in Test262 with a high analysis precision of 99.0%. Now, we analyze them with different $k$-callsite-sensitivities to understand how different $k$ values affect the analysis results. We started from the context-insensitive analysis ($k = 0$) and increased $k$ of the $k$-callsite-sensitivity until their analysis precision is similar to that of the 20-callsite-sensitivity as depicted in Figure 11. As expected, the context-insensitive analysis has the lowest analysis precision of 52.2%. Then, the analysis precision consistently increases with a higher $k$ value, and it reaches 99.0% when $k = 4$.

Therefore, we showed that one can configure the analysis precision of $\text{JSA}_{\text{ES12}}$ by using different $k$-callsite-sensitivities for JavaScript.

*PipelineExpression* **:** *PipelineExpression* **|>** *LogicalORExpression*

**(a) Syntactic production for the pipeline operator**

```
1 let add    = y => x => x + y;
2 let double = z => z * 2;
3 let n = /* any integer from 0 to 99 */;
4 let a = n |> add(1)   // [1, 100]
5           |> double;  // [2, 200]
6 let b = n |> add(1n)  // TypeError for `+`
7           |> unknown; // unreachable
```

**(b) A JavaScript program using the pipeline operator**

**Figure 12: Syntax and use of the pipeline operator `|>`**

## 6.4 Adaptability

We evaluated the adaptability of JSAVER using two case studies with new language features. TC39 maintains proposals for future language features in GitHub repositories. In the order of the most GitHub stars, the top three features are the pipeline operator `|>`[5] with 6.1K stars, the pattern matching[6] with 4.1K stars, and the `Observable` library[7] with 2.8K stars. Because the pattern matching proposal is in an early stage with only basic concepts without any detailed semantics, we evaluated the adaptability of JSAVER with two proposals for the pipeline operator `|>` and the `Observable` library.

### 6.4.1 *Pipeline Operator (`|>`).*

The *pipeline operator* is typically supported in functional programming languages, such as F# and OCaml. Its behavior is almost the same with a syntactic sugar of a function call with a single argument. To support this operator, we first applied its proposal, which contains the syntactic production in Figure 12(a) and algorithms, to ES12. Then, we derived a JavaScript static analyzer from the updated ES12 via JSAVER. Finally, we analyzed the example JavaScript program in Figure 12(b) with the interval domain for integers using the derived analyzer.

First, the derived analyzer successfully analyzes the stored value in the variable `a`. The program defines two functions: `add` receives a value in `y` and adds it to the second argument in `x`, and `double` multiplies the argument `z` by `2`. The analyzer first analyzes that the variable `n` points to the interval `[0, 99]` on line 3. Then, the abstract value is updated to `[1, 100]` and `[2, 200]` by analyzing `|> add(1)` on line 4 and `|> double` on line 5, respectively. Therefore, the derived analyzer successfully analyzes that the variable `a` stores the interval `[2, 200]`. The derived analyzer also correctly analyzes the execution order of the pipeline operator on lines 6–7. The pipeline operator first executes the argument part rather than the function part. Thus, the original program throws a `TypeError` exception on line 6 because the addition of the BigInt value `1n` with another numeric value is ill-typed. The derived analyzer successfully analyzes that the program terminates on line 6 with a `TypeError` exception by correctly considering the execution order of the pipeline operator.

### 6.4.2 `Observable` *Library.*

JSAVER can support not only a new syntactic feature but also a new built-in library. Using the `Observable` library, we can model push-based data sources, such as DOM events, timer intervals, and sockets. Consider an example program in Figure 13. On lines 1–2, the program first randomly defines variables `x`

---

[5] https://github.com/tc39/proposal-pipeline-operator
[6] https://github.com/tc39/proposal-pattern-matching
[7] https://github.com/tc39/proposal-observable

```
1 let x = /* 1 or 2 */;
2 let y = /* any string */;
3 let o = new Observable(subscriber => {
4   subscriber.next(1);
5   subscriber.next(2);
6   subscriber.next(3);
7 });
8 o.subscribe(k => x *= k); // x: 6 or 12
9 o.subscribe(k => y += k); // y: any string + "123"
```

**Figure 13: An example of the `Observable` built-in library**

with `1` or `2` and `y` with a random string. Then, it *registers* an arrow function `subscriber => { ... }` to a new `Observable` object and assigns it to the variable `o` on lines 3–7. On line 8, it *subscribes* `k => x *= k` via `subscribe` to invoke the registered arrow function. Then, the arrow function `k => x *= k` is synchronously invoked three times with multiple values `1`, `2`, and `3`. Therefore, the variable `x` points to `6` or `12` because the initial value of `x` is `1` or `2`, and it is multiplied by `1`, `2`, and `3`. Similarly, the variable `y` points to any string ending with `"123"` because its initial value is a random string, and it is updated by concatenating string values of `1`, `2`, and `3`, on line 9.

To analyze the example program, we applied the proposal of the `Observable` library to ES12 and derived a JavaScript static analyzer from it. We used the interval domain for integers and the *Prefix-Suffix* (PS) domain explained in Section 6.3 for strings. On lines 1–2, the derived analyzer first assigns `[1, 2]` and ⟨`""`, `""`⟩ to the variables `x` and `y`, respectively. Then, it assigns the new abstract `Observable` object with the arrow function `subscriber => { ... }` to `o` by analyzing the invocation of the constructor of `Observable` on lines 3–7. On line 8, the analyzer analyzes that an arrow function `k => x *= k` is subscribed, and the variable `x` is updated to the interval `[6, 12]`. Similarly, it analyzes that another arrow function `k => y += k` is subscribed on line 9, and the variable `y` is updated to the abstract value ⟨`""`, `"123"`⟩. Thus, the derived analyzer successfully analyzes the example program and precisely represents the possible values of `x` and `y` at the end of the program.

## 6.5 Discussion

In this section, we discuss promising directions for the improvement of JSAVER and limitations of JISET, the tool used in the extraction of definitional interpreters from ECMA-262.

### *6.5.1 Promising Directions of JSAVER.*

The analyzer $\text{JSA}_{\text{ES12}}$ automatically derived from ES12 via JSAVER has two directions for improvement compared to existing hand-written JavaScript static analyzers.

First, because our approach considers only the semantics described in ECMA-262, $\text{JSA}_{\text{ES12}}$ does not support host environments such as DOM and Node.js used in modern JavaScript applications. However, just like existing analyzers, $\text{JSA}_{\text{ES12}}$ can utilize manual modeling of host environments to analyze real-world applications.

Second, as described in Section 6.2, $\text{JSA}_{\text{ES12}}$ is slower than existing analyzers. While JSAVER derives precise abstract semantics for all language features, developers of existing analyzers often model the abstract semantics of specific language features *imprecisely* or even *unsoundly* to enhance the analysis performance. A promising direction is to support host environments efficiently, possibly semi-automatically, and optimize derived analyzers for better performance and memory use.

### *6.5.2 Limitations of JISET.*

In this work, we utilized another tool JISET to extract a JavaScript definitional interpreter from ECMA-262. It has two limitations; it 1) covers only about 95% of the algorithm steps and 2) generates a JavaScript parser slower than hand-written parsers. Thus, a manual effort is still required for about 5% of the steps, and JSAVER slows down because of the longer parsing time. Nevertheless, we believe that JISET significantly reduces the burden of manual approaches and could generate a faster parser using more advanced parsing techniques.

## 7 RELATED WORK

*JavaScript Static Analysis.* Researchers have proposed JavaScript static analyzers, such as JSAI [22], SAFE [27, 44], TAJS [21], and WALA [51], to detect program bugs without concrete execution and to understand program behaviors. They also presented and implemented various JavaScript static analysis techniques on these tools. Since string values of arbitrary expressions can be used in property accesses, a precise string analysis is more critical for JavaScript than static analysis for other programming languages. Thus, several advanced string abstract domains [4, 23, 28, 32] have been presented for JavaScript. Several researchers presented analysis techniques [25, 31, 33, 51, 52] to increase imprecise relations between object properties. Moreover, due to the highly dynamic nature of JavaScript, static analyzers suffer from heavy computations as well as imprecise analysis results. Hence, combined analyses [41, 43, 45, 47, 55] with dynamic analyses have been proposed to enhance analysis performance by leveraging highly optimized commercial JavaScript engines.

However, all of the existing JavaScript static analyzers cannot support language features of ES6 or later versions, including `let` bindings, arrow functions, generators, and promises. JSAVER resolves this problem by automatically deriving JavaScript static analyzers from language specifications. Xu et al. [57] recently presented a technique to synthesize data-flow analyzers, but they focused on only Java-like languages, and the technique does not guarantee the soundness of synthesized analyzers. Note that the soundness of the meta-level static analysis for JavaScript comes from the soundness of the static analysis for $\text{IR}_{\text{ES}}$.

*Definitional Interpreter.* Reynolds [46] first introduced the concept of definitional interpreters to describe the semantics of defined-languages using their interpreters written in defining-languages. Darais et al. [13] extended them to a definitional abstract interpreter, representing the abstract semantics of a defined-language using its abstract interpreter written in a defining-language. However, unlike a meta-level static analysis, it directly describes the abstract semantics of the defined-language without using a static analyzer of the defining-language. Therefore, it still requires manual updates when the defined-language evolves. For the JavaScript programming language, Herman and Flanagan [17] proposed the first definitional interpreter written in ML to represent the JavaScript semantics. Then, Bodin et al. [6] manually defined the JavaScript semantics in JSCert using the Coq proof assistant and extracted a definitional interpreter from Coq to OCaml. However, they require manual updates when JavaScript evolves. On the other hand, Park et al. [42] presented JISET, which automatically extracts a JavaScript definitional interpreter from ECMA-262. Because JISET provides a way to deal with the JavaScript semantics mechanically, researchers have developed several tools on top of it, such as a test synthesizer [38] and a specification type analyzer [37]. Similarly, we developed JSAVER by extending JISET to automatically derive a static analyzer via a meta-level static analysis for JavaScript.

*Automatic Modeling.* For JavaScript static analysis, modeling the behaviors of built-in library functions is essential because their implementations are usually in other programming languages, such as C++, rather than in JavaScript. Because it is labor-intensive to manually model them, several researchers [5, 36] utilized type information to automatically model them. However, they oversimplify complex behaviors and miss side-effects. On the other hand, several researchers utilized concrete executions to model them using program synthesis [18] or input/output abstractions [39]. However, they do not guarantee the soundness of the generated abstract semantics. Unlike existing approaches, JSAVER automatically translates the abstract algorithms in ECMA-262 to $\text{IR}_{\text{ES}}$ functions and utilizes them in JavaScript static analysis. Therefore, the analyzer derived by JSAVER can soundly analyze the built-in library functions without any manual modeling.

## 8 CONCLUSION

The fast evolution and massive size of ECMA-262 make it difficult to develop and update JavaScript static analyzers manually. To resolve this problem, we present JSAVER, the first tool that automatically derives JavaScript static analyzers from the language specifications. The main idea of JSAVER is to shift the paradigm from *compiler*-based approaches to *interpreter*-based approaches to fully utilize "the interpreter-based nature" of JavaScript. It extracts a *definitional interpreter* from ECMA-262 and performs a *meta-level static analysis* to indirectly analyze JavaScript programs using the extracted interpreter. We also present how to configure *abstract domains* and *analysis sensitivities* for JavaScript indirectly in the meta-level static analysis. We evaluated JSAVER by using a derived static analyzer $\text{JSA}_{\text{ES12}}$ from the latest ECMA-262, ES12. It soundly analyzes all applicable 18,556 official conformance tests with 99.0% of precision in 590 ms on average. We also demonstrated the configurability and adaptability of JSAVER with several case studies. We believe that JSAVER can reduce the burden of defining the abstract semantics of numerous language features for static analysis of fast-evolving JavaScript.

## ACKNOWLEDGMENTS

This work was supported by National Research Foundation of Korea (NRF) (Grants 2022R1A2C200366011 and 2021R1A5A1021944) and the Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government (MSIT) (2022-0-00460).

## REFERENCES

[1] 2021. *A Meta-Level Static Analysis for JavaScript.* Technical Report. https://figshare.com/s/88ef85259c4ef6b6fa49

[2] 2021. engine262: An Implementation of ECMA-262 in JavaScript. Retrieved November 19, 2021 from https://github.com/engine262/engine262

[3] 2021. Narcissus: A JavaScript Interpreter written in pure JavaScript developed by Mozilla. Retrieved November 19, 2021 from https://github.com/mozilla/narcissus

[4] Roberto Amadini, Alexander Jordan, Graeme Gange, François Gauthier, Peter Schachte, Harald Søndergaard, Peter J Stuckey, and Chenyi Zhang. 2017. Combining String Abstract Domains for JavaScript Analysis: An Evaluation. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS).* https://doi.org/10.1007/978-3-662-54577-5_3

[5] SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. 2014. SAFE-WAPI: Web API Misuse Detector for Web Applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE).* https://doi.org/10.1145/2635868.2635916

[6] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming (POPL)* 49, 1 (2014), 87–100. https://doi.org/10.1145/2535838.2535876

[7] Arthur Charguéraud, Alan Schmitt, and Thomas Wood. 2018. JSExplain: A Double Debugger for JavaScript. In *Companion Proceedings of the The Web Conference (WWW).* https://doi.org/10.1145/3184558.3185969

[8] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. 1990. Analysis of Pointers and Structures. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI)* (White Plains, New York, USA) *(PLDI '90)*. Association for Computing Machinery, New York, NY, USA, 296–310. https://doi.org/10.1145/93542.93585

[9] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming languages (POPL).* https://doi.org/10.1145/512950.512973

[10] Patrick Cousot and Radhia Cousot. 1977. Static Determination of Dynamic Properties of Generalized Type Unions. In *Proceedings of an ACM Conference on Language Design for Reliable Software* (Raleigh, North Carolina). Association for Computing Machinery, New York, NY, USA, 77–94. https://doi.org/10.1145/800022.808314

[11] Patrick Cousot and Radhia Cousot. 1992. Abstract Interpretation Frameworks. *Journal of Logic and Computation (JLC)* 2, 4 (1992), 511–547. https://doi.org/10.1093/logcom/2.4.511

[12] Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. 2014. Automatic Analysis of Open Objects in Dynamic Language Programs. In *Proceedings of the 21st International Symposium on Static Analysis (SAS).* https://doi.org/10.1007/978-3-319-10936-7_9

[13] David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. 2017. Abstracting Definitional Interpreters (Functional Pearl). In *Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming (ICFP).* https://doi.org/10.1145/3110256

[14] José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. 2017. JaVerT: JavaScript Verification Toolchain. *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming (POPL)* 2, POPL (2017), 1–33. https://doi.org/10.1145/3158138

[15] Dionna Glaze and David Van Horn. 2014. Abstracting Abstract Control. In *Proceedings of the 10th ACM Symposium on Dynamic Languages (DLS).* https://doi.org/10.1145/2661088.2661098

[16] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The essence of JavaScript. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP).* Springer, 126–150. https://doi.org/10.1007/978-3-642-14107-2_7

[17] David Herman and Cormac Flanagan. 2007. Status Report: Specifying Javascript with ML. In *Proceedings of the 2007 Workshop on Workshop on ML.* https://doi.org/10.1145/1292535.1292543

[18] Stefan Heule, Manu Sridharan, and Satish Chandra. 2015. Mimic: Computing Models for Opaque Code. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE).* https://doi.org/10.1145/2786805.2786875

[19] Jerry L Hintze and Ray D Nelson. 1998. Violin Plots: A Box Plot-Density Trace Synergism. *The American Statistician* 52, 2 (1998), 181–184.

[20] Ecma International. 2021. ECMA-262, 12th edition, June 2021, ECMAScript®2021 Language Specification. https://262.ecma-international.org/12.0/

[21] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis (SAS).* https://doi.org/10.1007/978-3-642-03237-0_17

[22] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A Static Analysis Platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE).* https://doi.org/10.1145/2635868.2635904

[23] Se-Won Kim, Wooyoung Chin, Jimin Park, Jeongmin Kim, and Sukyoung Ryu. 2014. Inferring Grammatical Summaries of String Values. In *Asian Symposium on Programming Languages and Systems (APLAS).* Springer, 372–391. https://doi.org/10.1007/978-3-319-12736-1_20

[24] Se-Won Kim, Xavier Rival, and Sukyoung Ryu. 2018. A Theoretical Foundation of Sensitivity in an Abstract Interpretation Framework. *ACM Transactions on*

*Programming Languages and Systems (TOPLAS)* 40, 3, Article 13 (2018), 44 pages. https://doi.org/10.1145/3230624

[25] Yoonseok Ko, Xavier Rival, and Sukyoung Ryu. 2019. Weakly sensitive analysis for JavaScript object-manipulating programs. *Software: Practice and Experience (SPE)* 49, 5 (2019), 840–884. https://doi.org/10.1002/spe.2676

[26] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical for the Real World. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 278–289. https://doi.org/10.1145/1250734.1250766

[27] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. 2012. SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript. In *Proceedings of 19th International Workshop on Foundations of Object-Oriented Languages (FOOL).*

[28] Magnus Madsen and Esben Andreasen. 2014. String Analysis for Dynamic Field Access. In *Proceedings of the 23rd International Conference on Compiler Construction (CC).* https://doi.org/10.1007/978-3-642-54807-9_12

[29] Matthew Might and Olin Shivers. 2006. Improving Flow Analyses via ΓCFA: Abstract Garbage Collection and Counting. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP).* 13–25. https://doi.org/10.1145/1159803.1159807

[30] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 14, 1 (Jan. 2005), 1–41. https://doi.org/10.1145/1044834.1044835

[31] Benjamin Barslev Nielsen and Anders Møller. 2020. Value Partitioning: A Lightweight Approach to Relational Static Analysis for JavaScript. In *Proceedings of the 34th European Conference on Object-Oriented Programming (ECOOP).* https://doi.org/10.4230/LIPIcs.ECOOP.2020.16

[32] Changhee Park, Hyeonseung Im, and Sukyoung Ryu. 2016. Precise and Scalable Static Analysis of jQuery using a Regular Expression Domain. In *Proceedings of the 12th Symposium on Dynamic Languages (DLS).* https://doi.org/10.1145/2989225.2989228

[33] Changhee Park, Hongki Lee, and Sukyoung Ryu. 2018. Static analysis of JavaScript libraries in a scalable and precise way using loop sensitivity. *Software: Practice and Experience (SPE)* 48, 4 (2018), 911–944. https://doi.org/10.1002/spe.2676

[34] Changhee Park and Sukyoung Ryu. 2015. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. In *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP).* https://doi.org/10.4230/LIPIcs.ECOOP.2015.735

[35] Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 346–356. https://doi.org/10.1145/2737924.2737991

[36] Jihyeok Park. 2014. JavaScript API misuse detection by using TypeScript. In *Proceedings of the companion publication of the 13th international conference on Modularity.* https://doi.org/10.1145/2584469.2584472

[37] Jihyeok Park, Seungmin An, Wonho Shin, Yusung Sim, and Sukyoung Ryu. 2021. JSTAR: JavaScript Specification Type Analyzer using Refinement. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE).*

[38] Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu. 2021. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. In *Proceedings of IEEE/ACM 43rd International Conference on Software Engineering (ICSE).* IEEE, 13–24. https://doi.org/10.1109/ICSE43902.2021.00015

[39] Joonyoung Park, Alexander Jordan, and Sukyoung Ryu. 2019. Automatic Modeling of Opaque Code for JavaScript Static Analysis. In *Proceedings of the 22nd International Conference on Fundamental Approaches to Software Engineering (FASE).* https://doi.org/10.1007/978-3-030-16722-6_3

[40] Jihyeok Park, Hongki Lee, and Sukyoung Ryu. 2021. A Survey of Parametric Static Analysis. *ACM Computing Surveys (CSUR)* 54, 7 (2021), 1–37. https://doi.org/10.1145/3464457

[41] Joonyoung Park, Inho Lim, and Sukyoung Ryu. 2016. Battles with False Positives in Static Analysis of JavaScript Web Applications in the Wild. In *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering Companion (ICSE-C).* https://doi.org/10.1145/2889160.2889227

[42] Jihyeok Park, Jihee Park, Seungmin An, and Sukyoung Ryu. 2020. JISET: JavaScript IR-based Semantics Extraction Toolchain. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 647–658. https://doi.org/10.1145/3324884.3416632

[43] Joonyoung Park, Jihyeok Park, Dongjun Youn, and Sukyoung Ryu. 2021. Accelerating JavaScript Static Analysis via Dynamic Shortcuts. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)* (Athens, Greece) *(ESEC/FSE 2021).* Association for Computing Machinery, New York, NY, USA, 1129–1140. https://doi.org/10.1145/3468264.3468556

[44] Jihyeok Park, Yeonhee Ryou, Joonyoung Park, and Sukyoung Ryu. 2017. Analysis of JavaScript Web Applications Using SAFE 2.0. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering Companion (ICSE-C).* https://doi.org/10.1109/ICSE-C.2017.4

[45] Joonyoung Park, Kwangwon Sun, and Sukyoung Ryu. 2018. EventHandler-Based Analysis Framework for Web Apps Using Dynamically Collected States. In *Proceedings of the 21st International Conference on Fundamental Approaches to Software Engineering (FASE).* https://doi.org/10.1007/978-3-319-89363-1_8

[46] John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2.* https://doi.org/10.1145/800194.805852

[47] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Dynamic Determinacy Analysis. In *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/2499370.2462168

[48] Micha Sharir and Amir Pnueli. 1981. *Two Approaches to Interprocedural Data Flow Analysis.*

[49] Olin Grigsby Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages.* Carnegie Mellon University.

[50] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (Austin, Texas, USA). Association for Computing Machinery, New York, NY, USA, 17–30. https://doi.org/10.1145/1926385.1926390

[51] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation Tracking for Points-To Analysis of JavaScript. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP).* https://doi.org/10.1007/978-3-642-31057-7_20

[52] Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. 2019. Static Analysis with Demand-Driven Value Refinement. In *Proceedings of the 34th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* https://doi.org/10.1145/3360566

[53] David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (POPL).* https://doi.org/10.1145/1863543.1863553

[54] David Van Horn and Matthew Might. 2011. Abstracting Abstract Machines: A Systematic Approach to Higher-Order Program Analysis. *Communications of the ACM (CACM)t* 54, 9 (2011), 101–109. https://doi.org/10.1145/1995376.1995400

[55] Shiyi Wei and Barbara G Ryder. 2013. Practical Blended Taint Analysis for JavaScript. In *Proceedings of the 22nd International Symposium on Software Testing and Analysis (ISSTA).* https://doi.org/10.1145/2483760.2483788

[56] Allen Wirfs-Brock and Brendan Eich. 2020. JavaScript: The First 20 Years. *Proceedings of ACM Programming Languages* 4, HOPL, Article 77 (June 2020), 189 pages. https://doi.org/10.1145/3386327

[57] Xuezheng Xu, Xudong Wang, and Jingling Xue. 2021. Automatic Synthesis of Data-Flow Analyzers. In *Proceedings of the 28th International Symposium on Static Analysis (SAS).* Springer, 453–478.