Accelerating JavaScript Static Analysis via Dynamic Shortcuts

Joonyoung Park*
Korea Advanced Institute of Science and Technology
Daejeon, South Korea
gmb55@kaist.ac.kr

Jihyeok Park*
Korea Advanced Institute of Science and Technology
Daejeon, South Korea
jhpark0223@kaist.ac.kr

Dongjun Youn
Korea Advanced Institute of Science and Technology
Daejeon, South Korea
f52985@kaist.ac.kr

Sukyoung Ryu
Korea Advanced Institute of Science and Technology
Daejeon, South Korea
sryu.cs@kaist.ac.kr

ABSTRACT

JavaScript has become one of the most widely used programming languages for web development, server-side programming, and even micro-controllers for IoT. However, its extremely functional and dynamic features degrade the performance and precision of static analysis. Moreover, the variety of built-in functions and host environments requires excessive manual modeling of their behaviors. To alleviate these problems, researchers have proposed various ways to leverage dynamic analysis during JavaScript static analysis. However, they do not fully utilize the high performance of dynamic analysis and often sacrifice the soundness of static analysis.

In this paper, we present *dynamic shortcuts*, a new technique to flexibly switch between abstract and concrete execution during JavaScript static analysis in a sound way. It can significantly improve the analysis performance and precision by using highly-optimized commercial JavaScript engines and lessen the modeling efforts for opaque code. We actualize the technique via SAFE_{DS}, an extended combination of SAFE and Jalangi, a static analyzer and a dynamic analyzer, respectively. We evaluated SAFE_{DS} using 269 official tests of Lodash 4 library. Our experiment shows that SAFE_{DS} is 7.81x faster than the baseline static analyzer, and it improves the precision to reduce failed assertions by 12.31% on average for 22 opaque functions.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

JavaScript, static analysis, dynamic analysis, dynamic shortcut, sealed execution

*Both authors contributed equally to the paper.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468556

ACM Reference Format:
Joonyoung Park, Jihyeok Park, Dongjun Youn, and Sukyoung Ryu. 2021. Accelerating JavaScript Static Analysis via Dynamic Shortcuts. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468556

1 INTRODUCTION

Over the past decades, the rise of JavaScript as the de facto language for web development has expanded its reach to diverse fields. Node.js [5] supports server-side programming, React Native [6] and Electron [1] produce cross-platform applications, and Moddable [4] and Espruino [2] provide JavaScript environments in micro-controllers for IoT. Such wide prevalent uses place JavaScript at #7 programming language in the TIOBE Programming Community index[1]. Thus, researchers have developed static analyzers such as JSAI [20], TAJS [19], WALA [41], and SAFE [25, 37] to understand behaviors of JavaScript programs and to detect their bugs in a sound manner.

However, static analysis of real-world JavaScript programs suffers from immensely functional and dynamic features of JavaScript such as callback functions, first-class property names, and dynamic code generation. While they provide flexibility in software development, it is challenging to statically analyze such features. To overcome these problems, researchers have proposed several analysis techniques: advanced string domains [10, 26, 29], loop sensitivity [30, 31], analysis based on property relations [23, 24, 28, 41], and on-demand backward analysis [42].

At the same time, JavaScript host environments require excessive manual modeling of their behaviors for static analysis. Because built-in functions and host-dependent functions are implemented in native languages like C and C++ instead of JavaScript, their code is *opaque* during static analysis. Thus, static analyzers often model their behaviors manually, which is error-prone, tedious, and labor-intensive. While researchers have proposed automatic modeling techniques [11, 32], since they utilize only type information, they generate imprecise models compared with the manual approach.

To alleviate these problems, researchers have leveraged dynamic analysis during static analysis. Unlike static analyzers that run on their own interpreters, dynamic analyzers such as Jalangi [40] and DLint [17] run on highly-optimized commercial JavaScript engines, which makes them much faster than static analyzers. Figure 1 shows

[1] https://www.tiobe.com/tiobe-index/

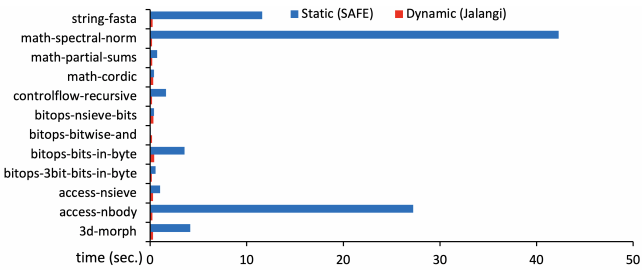


Figure 1: Performance of a dynamic analyzer and a static analyzer for a subset of the SunSpider benchmark

that the dynamic analyzer Jalangi is 34.8x faster than the static analyzer SAFE for a subset of the SunSpider [8] benchmark that is input-independent and deterministic. Using high performance dynamic analysis, researchers have reduced the scope of static analysis [39, 44] and constructed initial abstract states [34, 38] and automatic modeling of opaque code [33].

Unfortunately, existing techniques using dynamic analysis for static analysis have two limitations: 1) they do not fully utilize the high performance of dynamic analysis, and 2) they sacrifice the soundness of static analysis. Most of them are *staged analyses*, which first extract specific information via dynamic analysis and utilize it in static analysis. Schäfer et al. [39] identify determinate expressions that always have the same values at given program points, Wei and Ryder [44] extract dynamic values to change expressions to certain literals, and Park et al. [34, 38] dump the initial states of a certain host environment or the entry of an event handler. However, because they do not utilize dynamic analysis as soon as static analysis begins, they do not get performance benefits since then. Moreover, they sacrifice the soundness of static analysis by performing dynamic analysis. For example, the SRA model [33] uses dynamic analysis for opaque code with abstract arguments during static analysis. When the abstract arguments represent an infinite number of values, it randomly samples finite concrete values for the abstract arguments, which makes the analysis result unsound due to missing concrete values.

In this paper, we present *dynamic shortcuts*, a new technique to flexibly switch between abstract and concrete execution during JavaScript static analysis in a sound way. During static analysis, one can take a dynamic shortcut, which consists of three parts: 1) converting the current abstract state to its corresponding *sealed state*, 2) performing *sealed execution* on the sealed state, and 3) converting the result of the sealed execution to its corresponding abstract state. Our key observation is that we can use the fast concrete execution for specific program parts while preserving the soundness if they do not use abstract values. For example, consider static analysis of the following JavaScript code:

```
var v = ... // an abstract value
var obj = { p1: v }, y = "p";
x = obj[y + 1];
```

Because `y` stores a string `"p"`, the expression `y + 1` evaluates to a string `"p1"` and `x = obj[y + 1]` assigns the abstract value of `v` stored in `obj.p1` to the variable `x`. Note that even though `obj` contains an abstract value `v`, because the third line does not "use" the value of `v` but only "passes" it to the variable `x`, we can concretely execute the code. Based on this observation, we introduce sealed execution, which is concrete execution using *sealed values*. A sealed value is a symbol that represents an abstract value in sealed execution; it signals the end of the current dynamic shortcut when the sealed execution tries to access its value. To evaluate our technique, we implemented SAFE_{DS} using SAFE and Jalangi and analyzed 269 official tests of Lodash 4 library.

The contributions of this paper include the following:

- We present a novel technique for JavaScript static analysis to leverage the high performance of dynamic analysis using dynamic shortcuts. We formally define the technique and prove its soundness and termination.
- We actualize the proposed technique in SAFE_{DS}, an extended combination of SAFE and Jalangi.
- For empirical evaluation, we analyzed 269 official tests of Lodash 4 library. The experiment shows that SAFE_{DS} outperforms SAFE 7.81x on average. Moreover, by using dynamic shortcuts instead of manual modeling for 22 opaque functions, SAFE_{DS} improves the analysis precision to reduce failed assertions by 12.31% on average.

In the remainder of this paper, Section 2 explains the motivation of this work with a simple example. Section 3 formalizes the language-agnostic part of the technique in the abstract interpretation framework. Then, we extend the formalization with JavaScript specific features in Section 4. Section 5 describes important details of the SAFE_{DS} implementation. We explain the evaluation results of SAFE_{DS} with real-world benchmarks in Section 6. Section 7 discusses related work and Section 8 concludes.

2 MOTIVATION

This section explains the motivation of dynamic shortcuts using real-world examples in Figure 2. We describe their behaviors and explain how we can utilize dynamic shortcuts during static analysis.

Figure 2(a) shows the `concat` function defined in Lodash library [3] (v4.17.20); it is the most popular npm package[2] and 131,517 npm packages have a dependency on it. The `concat` function creates a new array concatenating given arrays or values. It first checks the length of `arguments` on lines 1–3. Then, it stores the first argument to `array` on line 4 and copies the remaining arguments to `args` on lines 5–8. On line 9, it checks whether `array` is an array object using the built-in function `isArray`. If so, it creates a new array by copying the given array via `copyArray`; otherwise, it creates a singleton array `[array]`. Finally, it flattens `args` via `baseFlatten` and pushes the result to the new array on line 11.

Figure 2(b) and Figure 2(c) show use cases of the `concat` function in the `zoom.us` [9] site. It is the homepage of Zoom, a videotelephony software by Zoom Video Communications and it is ranked as the 15th popular web site according to Alexa[3] in February 2021.

Dynamic shortcuts with concrete values. When a function is called with concrete values, we can perform dynamic analysis instead of static analysis. For example, `changeCountry` in Figure 2(b) is invoked

[2]https://www.npmjs.com/browse/depended
[3]https://www.alexa.com/siteinfo/zoom.us

```
1  function concat() {
2    var length = arguments.length;
3    if (!length) return [];
4    var array = arguments[0],
5        args  = Array(length - 1),
6        index = length;
7    while (index--)
8      args[index-1] = arguments[index];
9    return arrayPush(isArray(array) ?
10     copyArray(array) : [array],
11     baseFlatten(args, 1));
12 }
```

(a) Lodash's concat function

```
13 function changeCountry(G) { ...
14   if (G.selectedVal === "US" && state) {
15     // deterministic arguments of `concat`
16     state.items = _.concat([["Other", "Other"]],
17       WebinarBase.questions.state.items);
18     state.selectedVal = _.head(_.head(C.items));
19   }
20 }
```

(b) Call of concat with concrete values

```
22 function getData(e) {
23   var option = ... // option for server connection
24   post(option).then(function(e) {
25     if (e.total_records && e.total_records > 0) {
26       // non-deterministic arguments of `concat`
27       this.pastEvents =
28         _.concat(this.pastEvents, e.events);
29       this.total = e.total_records;
30     } else this.noPastData = !0
31   })
32 }
```

(c) Call of concat with abstract values

Figure 2: Lodash library function and its uses in `zoom.us`

when a user selects a country from a drop-down list in the registration page. It calls the `concat` function to update the drop-down list of states or provinces on lines 16–17. However, when the user selects "United States of America," which is `"US"`, two arguments are pre-defined with deterministic values; the first one is an array literal `[["Other", "Other"]]` and the second one is an array of pairs of abbreviations and names of the states defined as follows:

```
WebinarBase.questions.state.items =
  [["AL","Alabama"], ..., ["WY", "Wyoming"]]
```

Moreover, `this` also has a concrete value, the Lodash top-level object _. Thus, we can perform dynamic analysis by invoking `concat` with _ as its `this` value and the above concrete values as arguments. By skipping the analysis of the function call on lines 17–18 and utilizing the result of dynamic analysis, it improved the analysis performance.

Dynamic shortcuts with abstract values. Even when a function is called with abstract values, we can still perform dynamic analysis using sealed execution. For example, `getData` in Figure 2(c) is invoked when a user clicks the "Load More" button to load more Zoom events in the "Webinars & Events" page. It sends a POST request to a server and receives additional events `e` on line 24. Then, eight events in

Property	Value
⊤	$\omega_{\texttt{evt}}$
`"length"`	$\omega_{\texttt{int}}$

(a) `this.pastEvents`

Property	Value
`0`	$\omega_{\texttt{evt}}$
...	...
7	$\omega_{\texttt{evt}}$
`"length"`	8

(b) `e.events`

Figure 3: Concrete objects with sealed values

`e.events` are appended to `this.pastEvents` using `concat` on lines 27–28. However, the arguments of `concat` are not deterministic because 1) the event list stored in `this.pastEvents` is continuously grown for each load and 2) the events stored in `e.events` are dependent on the data given from the server.

To perform dynamic analysis with abstract values, we seal abstract values with sealed values as in Figure 3. Two sealed values $\omega_{\texttt{evt}}$ and $\omega_{\texttt{int}}$ represent an event object and an integer, respectively. Then, we can perform dynamic analysis successfully until line 9. On line 2, `length` is `2`; on line 4, `array` points to `this.pastEvents`; on lines 5–8, `args` stores an array with a single object stored in `e.events`; and on line 9, `isArray(array)` is `true`. However, dynamic analysis fails for `copyArray(array)` on line 10 because the value of the `length` property of `array` is the sealed value $\omega_{\texttt{int}}$. Then, we stop the sealed execution, convert the current sealed state to its corresponding abstract state, and resume the static analysis from line 10. Because sealed execution leverages fast dynamic analysis as long as possible, the overall analysis becomes more scalable.

Dynamic shortcuts for opaque functions. As the previous two examples additionally show, using dynamic shortcuts lessens the burden of modeling opaque functions from static analysis, and it can even improve the analysis precision. On line 9, since the `isArray` function is a JavaScript built-in library function, it is implemented in a native language of the host environment, which often requires manual modeling of its behaviors for JavaScript static analysis. Assuming that a static analyzer models `isArray` to return the boolean top value $\top_b$ that encompasses both `true` and `false`, static analysis of the ternary conditional expression on lines 9–10 analyzes both branches `copyArray(array)` and `[array]`, even though `[array]` is never reachable in the example code. On the contrary, using dynamic shortcuts, static analysis does not need to model `isArray`. It can perform sealed execution for `isArray`, which returns a more precise result `true` than $\top_b$.

3 DYNAMIC SHORTCUTS

In this section, we formally define static analysis using dynamic shortcuts by introducing sealed execution in the abstract interpretation framework. We extend the formalization of abstract interpretation of Cousot and Cousot [13, 14] and views-based analysis sensitivity of Kim et al. [21]. For dynamic shortcuts, we define sealed execution with a sealed domain and abstract instantiation maps. To combine sensitive abstract interpretation and sealed execution, we define a combined domain of sensitive abstract domain and sealed domain and explain it with a simple example. Finally, we prove the soundness and termination property of abstract interpretation using the combined domain.

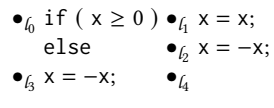


Figure 4: Negation of the absolute value of x

3.1 Concrete Semantics

We define a program P as a state transition system $(\mathbb{S}, \rightsquigarrow, \mathbb{S}_\iota)$. A program starts with an initial state in $\mathbb{S}_\iota$ and the transition relation $\rightsquigarrow \subseteq \mathbb{S} \times \mathbb{S}$ describes how states are transformed to other states. A *collecting semantics* $[\![P]\!] = \{\sigma \in \mathbb{S} \mid \sigma_\iota \in \mathbb{S}_\iota \wedge \sigma_\iota \rightsquigarrow^* \sigma\}$ consists of reachable states from initial states of the program P. We can compute it using a *transfer function* $F : \mathbb{D} \to \mathbb{D}$ as follows:

$$[\![P]\!] = \lim_{n\to\infty} F^n(d_\iota) \qquad F(d) = d \sqcup \mathrm{step}(d)$$

where the *concrete domain* $\mathbb{D} = \mathcal{P}(\mathbb{S})$ is a complete lattice with $\cup$, $\cap$, and $\subseteq$ as its join($\sqcup$), meet($\sqcap$), and partial order($\sqsubseteq$) operators. The set of states d_ι denotes the initial states $\mathbb{S}_\iota$. The *one-step execution* $\mathrm{step} : \mathbb{D} \to \mathbb{D}$ transforms states using the transition relation $\rightsquigarrow$: $\mathrm{step}(d) = \{\sigma' \mid \sigma \in d \wedge \sigma \rightsquigarrow \sigma'\}$.

For example, the code in Figure 4 is a simple program that calculates the negation of the absolute value of the variable x. States are pairs of labels and integers stored in x: $\mathbb{S} = \mathcal{L} \times \mathbb{N}$. Assume that the initial states are $\mathbb{S}_\iota = \{(\ell_0, -42)\}$, which denotes that the program starts at ℓ_0 with the variable x of value -42. Then, it executes with the following trace:

$$(\ell_0, -42) \rightsquigarrow (\ell_2, -42) \rightsquigarrow (\ell_3, 42) \rightsquigarrow (\ell_4, -42)$$

3.2 Abstract Interpretation

Abstract interpretation [13, 14] over-approximates the transfer function F as an *abstract transfer function* $F^\# : \mathbb{D}^\# \to \mathbb{D}^\#$ to get an *abstract semantics* $[\![P]\!]^\#$ in finite iterations as follows:

$$[\![P]\!]^\# = \lim_{n\to\infty} (F^\#)^n(d_\iota^\#)$$

We define a *state abstraction* $\mathbb{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{D}^\#$ as a Galois connection between the concrete domain $\mathbb{D}$ and an abstract domain $\mathbb{D}^\#$ with a *concretization function* γ and an *abstraction function* α. The initial abstract state $d_\iota^\# \in \mathbb{D}^\#$ represents an abstraction of the initial state set: $d_\iota \subseteq \gamma(d_\iota^\#)$. The abstract transfer function $F^\# : \mathbb{D}^\# \to \mathbb{D}^\#$ is defined as $F^\#(d^\#) = d^\# \sqcup \mathrm{step}^\#(d^\#)$ with an *abstract one-step execution* $\mathrm{step}^\# : \mathbb{D}^\# \to \mathbb{D}^\#$. For a sound state abstraction, the join operator and the abstract one-step execution should satisfy the following conditions:

$$\forall d_0^\#, d_1^\# \in \mathbb{D}^\#.\ \gamma(d_0^\#) \cup \gamma(d_1^\#) \subseteq \gamma(d_0^\# \sqcup d_1^\#) \tag{1}$$

$$\forall d^\# \in \mathbb{D}^\#.\ \mathrm{step} \circ \gamma(d^\#) \subseteq \gamma \circ \mathrm{step}^\#(d^\#) \tag{2}$$

A simple example abstract domain is $\mathbb{D}_\pm^\# = \mathcal{P}(\{-, +, 0\})$ with set operators as domain operators; $-$ denotes negative integers, $+$ positive integers, and 0 zero. Assume that we analyze the code in Figure 4 with the abstract domain and the initial abstract state $d_\iota^\# = \{-\}$. Then, the analysis result is $\{-, +\}$ because x can have a positive value by executing x = −x but there is no way for x to have 0 in this program.

3.3 Analysis Sensitivity

Abstract interpretation is often defined with *analysis sensitivity* to increase the precision of static analysis. A sensitive abstract domain $\mathbb{D}_\delta^\# : \Pi \to \mathbb{D}^\#$ is defined with a *view abstraction* $\delta : \Pi \to \mathbb{D}$ that provides multiple points of views for reachable states during static analysis. It maps a finite number of views Π to sets of states $\mathbb{D}$. Each view $\pi \in \Pi$ represents a set of states $\delta(\pi)$ and each state is included in a unique view: $\forall \sigma \in \mathbb{S}.\ \sigma \in \delta(\pi) \Rightarrow \forall \pi' \in \Pi. \sigma \in \delta(\pi') \Rightarrow \pi = \pi'$. A *sensitive state abstraction* $\mathbb{D} \underset{\alpha_\delta}{\overset{\gamma_\delta}{\leftrightarrows}} \mathbb{D}_\delta^\#$ is a Galois connection between the concrete domain $\mathbb{D}$ and the sensitive abstract domain $\mathbb{D}_\delta^\#$ with the following concretization function:

$$\gamma_\delta(d_\delta^\#) = \bigcup_{\pi\in\Pi} \delta(\pi) \cap \gamma \circ d_\delta^\#(\pi)$$

With analysis sensitivities, the abstract one-step execution $\mathrm{step}_\delta^\# : \mathbb{D}_\delta^\# \to \mathbb{D}_\delta^\#$ is defined as follows:

$$\mathrm{step}_\delta^\#(d_\delta^\#) = \lambda\pi \in \Pi.\ \bigsqcup_{\pi'\in\Pi} [\![\pi' \to \pi]\!]^\# \circ d_\delta^\#(\pi')$$

where $[\![\pi' \to \pi]\!]^\# : \mathbb{D}^\# \to \mathbb{D}^\#$ is an abstract semantics of a *view transition* from a view π' to another view π. It should satisfy the following condition for the soundness of the analysis:

$$\forall d^\# \in \mathbb{D}^\#.\ \mathrm{step}(\gamma(d^\#) \cap \delta(\pi')) \cap \delta(\pi) \subseteq \gamma \circ [\![\pi' \to \pi]\!]^\#(d^\#)$$

One of the most widely-used analysis sensitivity is *flow sensitivity* defined with a flow-sensitive view abstraction $\delta^{\mathsf{FS}} : \mathcal{L} \to \mathbb{D}$ where:

$$\forall \ell \in \mathcal{L}.\ \delta^{\mathsf{FS}}(\ell) = \{\sigma \mid \sigma = (\ell, \_)\}$$

If we apply the flow sensitivity for the above example with the initial abstract state $[\ell_0 \mapsto \{-, 0, +\}]$, the analysis result is as follows:

$\mathcal{L}$	ℓ_0	ℓ_1	ℓ_2	ℓ_3	ℓ_4
$\mathbb{D}_\pm^\#$	$-, 0, +$	$0, +$	$-$	$0, +$	$-, 0$

3.4 Sealed Execution

We define *sealed execution* by extending the transition relation $\rightsquigarrow$ as a sealed transition relation $\rightsquigarrow_\omega$ on sealed states. First, we extend concrete states $\mathbb{S}$ to sealed states $\mathbb{S}_\omega$ by extending values $\mathbb{V}$ with *sealed values* Ω. We also define the sealed transition relation $\rightsquigarrow_\omega \subseteq \mathbb{S}_\omega \times \mathbb{S}_\omega$. We use the notation $\rightsquigarrow_\omega^k$ for k repetition of $\rightsquigarrow_\omega$, and write $\sigma_\omega \rightsquigarrow_\omega \bot$ when σ_ω does not have any sealed transitions to other sealed states. We define the validity of sealed execution as follows:

Definition 3.1 (Validity). The sealed transition relation is *valid* when the following condition is satisfied for any sealed states σ_ω and σ'_ω:

$$\sigma_\omega \rightsquigarrow_\omega \sigma'_\omega \Leftrightarrow \forall m \in \mathbb{M}.\ \{\sigma' \mid \sigma_\omega|_m \rightsquigarrow \sigma'\} = \{\sigma'_\omega|_m\}$$

where $\mathbb{M} : \Omega \to \mathbb{V}$ represent *instantiation maps* from sealed values to concrete values, and $\sigma_\omega|_m$ denotes a state produced by replacing each sealed value ω in σ_ω with its corresponding value $m(\omega)$ using the instantiation map $m \in \mathbb{M}$.

Sealed execution is different from traditional symbolic execution [22] in that it supports only sealed values instead of symbolic expressions and path constraints. For example, the following trace

represents traditional symbolic execution of the running example in Figure 4:

$$(\ell_0, \omega)\,[\varnothing] \begin{cases} \nearrow (\ell_1, \omega)\,[\omega \geq 0] \rightsquigarrow (\ell_3, \ \omega)\,[\omega \geq 0] \rightsquigarrow (\ell_4, -\omega)\,[\omega \geq 0] \\ \searrow (\ell_2, \omega)\,[\omega < 0] \rightsquigarrow (\ell_3, -\omega)\,[\omega < 0] \rightsquigarrow (\ell_4, \ \omega)\,[\omega < 0] \end{cases}$$

It first assigns a symbolic value ω to the variable x at ℓ_0. For the conditional branch, it creates two symbolic states with different path conditions $\omega \geq 0$ and $\omega < 0$ for true and false branches, respectively. After executing statements x = x and x = −x, the variable x stores symbolic expressions ω and $-\omega$ at ℓ_3, respectively. Similarly, x stores $-\omega$ and ω at ℓ_4. However, sealed execution stops at ℓ_0 as follows:

$$(\ell_0, \omega) \rightsquigarrow_\omega \bot$$

because the branch requires the actual value of the sealed value ω.

To define an abstract domain that contains sealed states, we define *abstract instantiation maps* $\mathbb{M}^\# : \Omega \to \mathbb{V}^\#$ from sealed values to abstract values. Its concretization function $\gamma_m : \mathbb{M}^\# \to \mathcal{P}(\mathbb{M})$ is defined with the concretization function $\gamma_v : \mathbb{V}^\# \to \mathcal{P}(\mathbb{V})$ for values as follows:

$$\gamma_m(m^\#) = \{m \mid \forall \omega \in \Omega.\ m(\omega) \in \gamma \circ m^\#(\omega)\}$$

The instantiation of a given sealed state $\sigma_\omega \in \mathbb{S}_\omega$ with an abstract instantiation map $m^\# \in \mathbb{M}^\#$ is defined as follows:

$$\sigma_\omega|_{m^\#} = \{\sigma_\omega|_m \mid m \in \gamma_m(m^\#)\}$$

Now, we define a *sealed domain* as follows:

Definition 3.2 (Sealed Domain). A *sealed domain* $\mathbb{D}_\omega : \mathcal{P}(\mathbb{M}^\# \times \mathbb{S}_\omega)$ is defined with the concretization function $\gamma_\omega : \mathbb{D}_\omega \to \mathbb{D}$ and the sealed one-step execution $\mathsf{step}_\omega : \mathbb{D}_\omega \to \mathbb{D}_\omega$ such that

$$\gamma_\omega(d_\omega) = \bigsqcup \{\sigma_\omega|_{m^\#} \mid (m^\#, \sigma_\omega) \in d_\omega\} \tag{3}$$

$$\mathsf{step}_\omega(d_\omega) = \{(m^\#, \sigma'_\omega) \mid (m^\#, \sigma_\omega) \in d_\omega \wedge \sigma_\omega \rightsquigarrow_\omega \sigma'_\omega\} \tag{4}$$

3.5 Combined Domain

We now define a *combined domain* of a given sensitive abstract domain with the sealed domain and its one-step execution.

Definition 3.3 (Combined Domain). A *combined domain* is $\widetilde{\mathbb{D}} = \mathbb{D}^\#_\delta \times \mathbb{D}_\omega$ and its concretization function $\widetilde{\gamma} : \widetilde{\mathbb{D}} \to \mathbb{D}$ and join operator are defined as follows:

$$\widetilde{\gamma}((d^\#_\delta, d_\omega)) = \gamma_\delta(d^\#_\delta) \cup \gamma_\omega(d_\omega) \tag{5}$$

$$(d^\#_\delta, d_\omega) \sqcup (d^{\#\prime}_\delta, d_\omega{}') = (d^\#_\delta \sqcup d^{\#\prime}_\delta, d_\omega \cup d_\omega{}') \tag{6}$$

Before defining the one-step execution for the combined domain, we introduce *analysis elements* to easily configure different types of abstract states in the sensitive abstract domain and the sealed domain.

Definition 3.4 (Analysis Elements). An *analysis element* $\epsilon \in \mathbb{E} = (\Pi \times \mathbb{D}^\#) \uplus (\mathbb{M}^\# \times \mathbb{S}_\omega)$ is either 1) a pair of a view and an abstract state in a sensitive abstract domain $\mathbb{D}^\#_\delta$, or 2) a pair of an abstract instantiation map and a sealed state in a sealed domain $\mathbb{D}_\omega$. Its concretization function $\gamma_\epsilon : \mathbb{E} \to \mathbb{D}$ is defined as follows:

$$\gamma_\epsilon(\epsilon) = \begin{cases} \delta(\pi) \cap \gamma(d^\#) & \text{if } (\pi, d^\#) = \epsilon \\ \sigma_\omega|_{m^\#} & \text{if } (m^\#, \sigma_\omega) = \epsilon \end{cases}$$

Moreover, to freely convert between different kinds of analysis elements, we define two converters:

$$\tau_\omega : (\Pi \times \mathbb{D}^\#) \hookrightarrow (\mathbb{M}^\# \times \mathbb{S}_\omega) \tag{7}$$

$$\tau^\# : (\Pi \times \mathbb{D}^\#) \hookleftarrow (\mathbb{M}^\# \times \mathbb{S}_\omega) \tag{8}$$

While the converter $\tau^\#$ is total, the other one τ_ω is *partial*. Thus, it is possible to convert an analysis element $(\pi, d^\#)$ in a sensitive abstract domain to another analysis element in a sealed domain only if the convert τ_ω is defined: $(\pi, d^\#) \in \mathsf{Dom}(\tau_\omega)$. In addition, they should convert given analysis elements without loss of information for all $\epsilon \in \mathbb{E}$:

$$\tau_\omega(\epsilon) = \epsilon' \Rightarrow \begin{cases} \epsilon = \tau^\#(\epsilon') \\ \gamma_\epsilon(\epsilon) = \gamma_\epsilon(\epsilon') \end{cases}$$

Now, we define the *combined one-step execution* $\widetilde{\mathsf{step}} : \widetilde{\mathbb{D}} \to \widetilde{\mathbb{D}}$ with two converters τ_ω and $\tau^\#$. It consists of two steps: 1) the *reform* step converts analysis elements if a new sealed execution starts or an existing one stops, and 2) the *execution* step performs execution of each analysis element using the abstract one-step execution $\mathsf{step}^\#_\delta$ in the sensitive abstract domain and the sealed one-step execution step_ω in the sealed domain.

Definition 3.5 (Combined One-Step Execution). A *combined one-step execution* $\widetilde{\mathsf{step}} : \widetilde{\mathbb{D}} \to \widetilde{\mathbb{D}}$ is define as follows:

$$\widetilde{\mathsf{step}}(\widetilde{d}) = (\mathsf{step}^\#_\delta(d^\#_\delta), \mathsf{step}_\omega(d_\omega))$$

where $(d^\#_\delta, d_\omega) = \mathsf{reform}(\widetilde{d})$.

From a given combined state $\widetilde{d}$, the reform function makes analysis elements and converts them if a new sealed execution begins or an existing sealed execution terminates. Specifically, for an analysis element $(\pi, d^\#)$ in the sensitive abstract domain, if the converter τ_ω is defined for it, reform introduces a new sealed execution by converting the analysis element to its corresponding one $(m^\#, \sigma_\omega) = \tau_\omega((\pi, d^\#))$ in the sealed domain. On the other hand, for an analysis element $(m^\#, \sigma_\omega)$ in the sealed domain, if it does not have any sealed states to transit to, $\sigma_\omega \rightsquigarrow_\omega \bot$, the sealed execution for $(m^\#, \sigma_\omega)$ terminates. It converts the analysis element to its corresponding one $(\pi, d^\#) = \tau^\#((m^\#, \sigma_\omega))$ in the sensitive abstract domain and merges the current abstract state stored in π with $d^\#$.

To formally define the reform function, we first define a reform_ϵ function for analysis elements using two converters.

Definition 3.6 (reform$_\epsilon$). The function $\mathsf{reform}_\epsilon : \mathbb{E} \to \mathbb{E}$ for analysis elements is defined as follows:

$$\mathsf{reform}_\epsilon(\epsilon) = \begin{cases} \tau_\omega(\epsilon) & \text{if } \epsilon = (\pi, d^\#) \wedge \epsilon \in \mathsf{Dom}(\tau_\omega) \\ \tau^\#(\epsilon) & \text{if } \epsilon = (m^\#, \sigma_\omega) \wedge \sigma_\omega \rightsquigarrow_\omega \bot \\ \epsilon & \text{Otherwise} \end{cases}$$

Definition 3.7 (reform). The reform function $\mathsf{reform} : \widetilde{\mathbb{D}} \to \widetilde{\mathbb{D}}$ for combined states is defined as follows:

$$\mathsf{reform}((d^\#_\delta, d_\omega)) = \left(\lambda\pi. \bigsqcup \{d^\# \mid (\pi, d^\#) \in E\}, E \cap (\mathbb{M}^\# \times \mathbb{S}_\omega)\right)$$

where

$$E = \dot{\mathsf{reform}}_\epsilon(\{(\pi, d^\#_\delta(\pi)) \mid \pi \in \Pi\} \cup d_\omega)$$

and the dot notation $\dot{f}$ denotes the element-wise extended function of a function f.

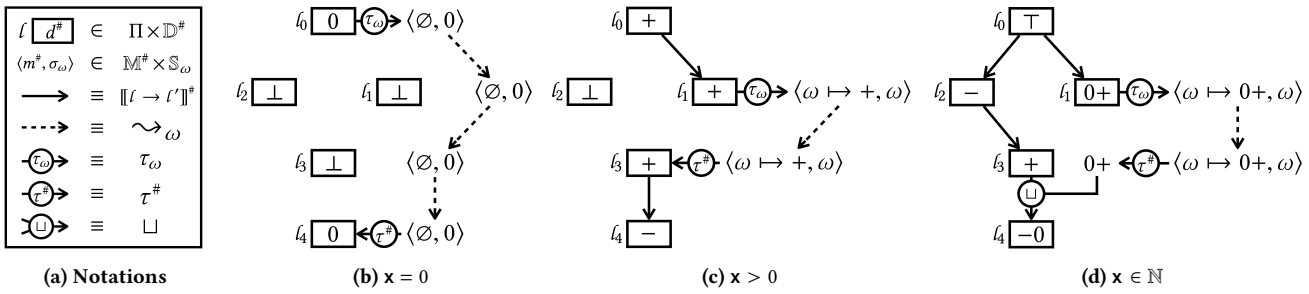


Figure 5: Abstract interpretation using a combined domain for the running example with different initial values for x.

3.6 Examples

Now, we show examples of abstract interpretation with a combined domain. Figure 5 depicts the flow of analysis for the running example in Figure 4 with three different initial sets of values for the variable x. In this example, we use the abstract domain $\{-, 0, +\}$ for integers stored in x as introduced in Section 3.2, and the *flow sensitivity* that utilizes the labels of states as their views as introduced in Section 3.3. For brevity, we use concatenation of abstract values so that -0 denotes the set $\{-, 0\}$.

Figure 5(a) presents notations used in each graph. A solid box denotes an analysis element that is a pair of a label ℓ and an abstract state $d^\#$. A pair enclosed by angle brackets denotes an analysis element that is a pair of an abstract instantiation map $m^\#$ and a sealed state σ_ω. In fact, the sealed state part (right) of each pair in graphs contains only the value of the variable of x without its label. For brevity, we represent its label by locating it next to a node with its label. A solid line is a view transition $[\![\ell \to \ell']\!]^\#$ from a label ℓ to another one ℓ'. A dotted line is a sealed transition $\rightsquigarrow_\omega$. Three solid lines with circled labels denote two converters $\tau^\#$, τ_ω and the join operator $\sqcup$.

Figure 5(b) shows the analysis with the combined domain when the initial value of x is 0. First, in the reform step, the converter τ_ω converts the analysis element $(\ell_0, 0)$ to another analysis element $\langle \varnothing, 0\rangle$ with the label ℓ_0. It does not introduce any sealed values because the value represents only a single value. Until the end of the program, the sealed execution from $\langle \varnothing, 0\rangle$ successfully continues. Because there is no more possible sealed transition for the sealed state $\langle \varnothing, 0\rangle$ with ℓ_4, it is converted to $(\ell_4, 0)$ via the converter $\tau^\#$.

Instead of a single value, assume that the initial value of x is one of any positive integers. Figure 5(c) describes the analysis flow for the case. The initial abstract value at the label ℓ_0 is + and it is impossible to convert it to any sealed values because the next program statement requires the actual value stored in the variable x for the branch condition $x \geq 0$. Thus, it performs view transition $[\![\ell_0 \to \ell_1]\!]^\#$ from the label ℓ_0 to another one ℓ_1 for the abstract value + and the result is also +. Now, the analysis element $(\ell_1, +)$ can be converted to $\langle \omega \mapsto +, \omega\rangle$ with the label ℓ_1. This sealed execution step terminates in the label ℓ_3 because the next statement is $x = -x$ and the negation operator requires the actual value of x. It is converted to $(\ell_3, +)$ via $\tau^\#$, performs the view transition, and results in $(\ell_4, -)$.

For the last case, we assume that all integers are possible for the initial value of the variable x as described in Figure 5(d). While it reaches the false branch in the label ℓ_2 unlike previous cases, it cannot perform dynamic shortcuts because the statement in the false branch is $x = -x$, which requires the actual value of x. At the label ℓ_3, there are two analysis elements: 1) $(\ell_3, +)$ introduced by the view transition from the label ℓ_2 with $-$, and 2) $\langle \omega \mapsto 0+, \omega\rangle$ with ℓ_3 introduced by sealed execution started at ℓ_1. Since it is not possible to perform sealed execution for both elements, the second one is converted to $(\ell_3, 0+)$ and merged with + at ℓ_3 via the join operator $\sqcup$. Finally, the view transition $[\![\ell_3 \to \ell_4]\!]^\#$ from ℓ_3 to ℓ_4 is performed to the merged abstract state $0+$ and the result is -0.

3.7 Soundness and Termination

The converter τ_ω and the sealed transition $\rightsquigarrow_\omega$ are keys to configure the introduction and termination of sealed execution. To ensure the *soundness* and *termination* of an abstract interpretation defined with a combined domain of a sensitive abstract domain and a sealed domain, the following conditions should hold.

Theorem 3.8 (Soundness and Termination). *An abstract interpretation with dynamic shortcuts is **sound** and **terminates** in a finite time if:*

- *the abstract transfer function $F^\#$ is sound,*
- *the sensitive abstract domain $\mathbb{D}^\#_\delta$ has a finite height,*
- *the sealed transition $\rightsquigarrow_\omega$ is valid, and*
- *there exists $N < \infty$ such that*

$$\forall \epsilon \in \mathbb{E}.\ \tau_\omega(\epsilon) = (m^\#, \sigma_\omega) \Rightarrow \sigma_\omega \rightsquigarrow_\omega^k \bot \wedge 1 < k \leq N$$

For soundness proof, we should prove two conditions presented in Section 3.2: (1) for the join operator $\sqcup$ and (2) for the combined one-step execution. The core idea of the proof is to use Lemma 3.9 and Lemma 3.10 for the sealed one-step execution step_ω and the reform function, respectively.

Lemma 3.9. *Assume that the following condition holds:*

$$\forall (m^\#, \sigma_\omega) \in d_\omega.\ \exists \sigma'_\omega \in \mathbb{S}_\omega.\ \sigma_\omega \rightsquigarrow_\omega \sigma'_\omega$$

then the following property holds:

$$step \circ \gamma_\omega(d_\omega) \subseteq \gamma_\omega \circ step_\omega(d_\omega)$$

Lemma 3.10. *For a given combined state $\widetilde{d} \in \widetilde{\mathbb{D}}$, the reform function satisfies the following two properties:*

- $\widetilde{\gamma}(\widetilde{d}) \subseteq \widetilde{\gamma} \circ reform(\widetilde{d})$
- $\forall (m^\#, \sigma_\omega) \in d_\omega.\ \exists \sigma'_\omega \in \mathbb{S}_\omega.\ \sigma_\omega \rightsquigarrow_\omega \sigma'_\omega$

where $(d^\#_\delta, d_\omega) = reform(\widetilde{d})$

The core idea of the termination proof is to use the property that the second and the fourth conditions provide upper bounds of the number of sensitive abstract states and the number of sealed states, respectively. We formally define and prove the property using *time to live (TTL)* functions of sealed states, TTL_i for each iteration $i \geq 0$, and prove the termination using them. Due to the page limitation, we omit the proof in this paper and present it in a companion report [35].

4 DYNAMIC SHORTCUTS FOR JAVASCRIPT

In this section, we introduce the core language of JavaScript that supports first-class functions, open objects, and first-class property names, and define sealed execution of the core language for dynamic shortcuts. Due to the space limitation, we present the main design of the language in this paper and refer the interested readers to a companion report [35].

4.1 Core Language of JavaScript

Programs	$P ::= (\ell : i)^*$
Labels	$\ell \in \mathcal{L}$
Instructions	$i ::= r = e \mid r = \{\} \mid r = e(e) \mid \texttt{ret}\ e \mid \texttt{if}\ e\ \ell$
References	$r ::= x \mid e[e]$
Expressions	$e ::= v_\mathsf{p} \mid \lambda x.\ \ell \mid r \mid \mathsf{op}(e^*)$

A program P is a sequence of labeled instructions. An instruction i is an expression assignment, an object creation, a function call, a return instruction, or a branch. A reference r is a variable or a property access of an object. An expression e is a primitive, a lambda function, a reference, or an operation between other expressions.

States	$\sigma \in \mathbb{S} = \mathcal{L} \times \mathcal{M} \times \mathbb{C} \times \mathbb{A}_\mathsf{env}$
Memories	$M \in \mathcal{M} = \mathbb{L} \xrightarrow{\text{fin}} \mathbb{V}$
Contexts	$c \in \mathbb{C} = \mathbb{A}_\mathsf{env} \xrightarrow{\text{fin}} (\mathbb{A}_\mathsf{env} \times \mathcal{L} \times \mathbb{L})$
Locations	$l \in \mathbb{L} = (\mathbb{A}_\mathsf{env} \times \mathbb{X}) \uplus (\mathbb{A}_\mathsf{obj} \times \mathbb{V}_\mathsf{str})$
Values	$v \in \mathbb{V} = \mathbb{V}_\mathsf{p} \uplus \mathbb{A}_\mathsf{obj} \uplus \mathbb{F}$
Primitives	$v_\mathsf{p} \in \mathbb{V}_\mathsf{p} = \mathbb{V}_\mathsf{str} \uplus \cdots$
Addresses	$a \in \mathbb{A} = \mathbb{A}_\mathsf{env} \uplus \mathbb{A}_\mathsf{obj}$
Functions	$\lambda x.\ell \in \mathbb{F} = \mathbb{X} \times \mathcal{L}$

States $\mathbb{S}$ consist of labels $\mathcal{L}$, memories $\mathcal{M}$, contexts $\mathbb{C}$, and environment addresses $\mathbb{A}_\mathsf{env}$. A memory $M \in \mathcal{M}$ is a finite mapping from locations to values. A context $c \in \mathbb{C}$ is a finite mapping from environment addresses to tuple of environment addresses, return labels, and left-hand side locations. A location $l \in \mathbb{L}$ is a variable or an object property; a variable location consists of an environment address and its name, and an object property location consists of an object address and a string value. A value $v \in \mathbb{V}$ is a primitive, an address, or a function value. An address $a \in \mathbb{A}$ is an environment address or an object address. A function value $\lambda x.\ell \in \mathbb{F}$ consists of a parameter name and a body label. In the core language, the closed scoping is used for functions for brevity, thus only parameters and local variables are accessible in a function body.

We formulate the concrete semantics of the core language. The transition relation between concrete states is defined with the semantics of references and expressions using two different forms $\boxed{\sigma \vdash_r r \Rightarrow l}$ and $\boxed{\sigma \vdash_e e \Rightarrow v}$, respectively. The initial states are $\mathbb{S}_\iota = \{(\ell_\iota, \varnothing, \epsilon, a_\mathsf{top})\}$ where ℓ_ι denotes the initial label, ϵ empty map, and a_top the top-level environment address.

4.2 Abstract Semantics

In the abstract semantics of the core language, we use the flow sensitivity with a flow sensitive view abstraction $\delta^\mathsf{FS} : \mathcal{L} \to \mathbb{D}$ that discriminates states using their labels: $\forall \ell \in \mathcal{L}.\ \delta^\mathsf{FS}(\ell) = \{\sigma \in \mathbb{S} \mid \sigma = (\ell, \_, \_, \_)\}$. Thus, the sensitive abstract domain is defined as $\mathbb{D}^\#_\delta = \mathcal{L} \to \mathbb{D}^\#$. We define an abstract state $d^\# \in \mathbb{D}^\#$ as a tuple of an abstract memory, an abstract context, an abstract address, and an abstract counter as follows:

Abstract states	$d^\# \in \mathbb{D}^\# = \mathcal{M}^\# \times \mathbb{C}^\# \times \mathbb{A}^\# \times \mathbb{N}^\#$
Abstract memories	$M^\# \in \mathcal{M}^\# = \mathbb{L}^\# \xrightarrow{\text{fin}} \mathbb{V}^\#$
Abstract locations	$l^\# \in \mathbb{L}^\# = (\mathbb{A}^\# \times \mathbb{X}) \uplus (\mathbb{A}^\# \times \mathbb{V}_\mathsf{str})$
Abstract addresses	$a^\# \in \mathbb{A}^\# = \mathcal{L}$
Abstract contexts	$c^\# \in \mathbb{C}^\# = \mathbb{A}^\# \xrightarrow{\text{fin}} \mathcal{P}(\mathbb{A}^\# \times \Pi \times \mathcal{P}(\mathbb{L}^\#))$
Abstract counters	$n^\# \in \mathbb{N}^\# = \mathbb{A}^\# \to \{0^\#, 1^\#, \geq 2^\#\}$
Abstract values	$v^\# \in \mathbb{V}^\# = \mathcal{P}(\mathbb{V}_\mathsf{p} \uplus \mathbb{A}^\# \uplus \mathbb{F})$

An abstract memory $M^\# \in \mathcal{M}^\#$ is a finite mapping from abstract locations $\mathbb{L}^\#$ to abstract values $\mathbb{V}^\#$. Abstract locations $\mathbb{L}^\#$ are pairs of abstract addresses with variable names or string values. Abstract addresses $\mathbb{A}^\#$ are defined with the *allocation-site abstraction* that partitions concrete addresses $\mathbb{A}$ based on their allocation sites $\mathcal{L}$. Abstract contexts $\mathbb{C}^\#$ are finite maps from abstract addresses to powersets of triples of abstract addresses, views, and powerset of abstract locations. For abstract counting [27, 36] in static analysis, we define abstract counters $\mathbb{N}^\#$ that are mappings from abstract addresses to their abstract counts representing how many times each abstract address has been allocated; $0^\#$ denotes that it has never been allocated, $1^\#$ once, and $\geq 2^\#$ more than or equal to twice.

We define the semantics of the view transition for the core language. For abstract memories, we use the notation $M^\#[L \mapsto\!\!\!\!\cdot\ v^\#]$ to represent the update of multiple abstract locations in L with the abstract value $v^\#$. It performs the strong update if the abstract address for an abstract location $(a^\#, \_) \in L$ is singleton: $n^\#(a^\#) = 1^\#$. Otherwise, it performs the weak update for the analysis soundness. We use the increment function $\mathsf{inc} : \mathbb{N}^\# \times \mathbb{A}^\# \to \mathbb{N}^\#$ of the abstract counter defined as follows:

$$\mathsf{inc}(n^\#)(a_0^\#) = \lambda a^\# \in \mathbb{A}^\#.\ \begin{cases} 1^\# & \text{if } a^\# = a_0^\# \wedge n^\#(a_0^\#) = 0^\# \\ \geq 2^\# & \text{if } a^\# = a_0^\# \wedge n^\#(a_0^\#) = 1^\# \\ n^\#(a^\#) & \text{otherwise} \end{cases}$$

4.3 Sealed Execution

We define sealed states by not only extending the concrete values $\mathbb{V}$ with sealed values Ω but also adding the abstract counters $\mathbb{N}^\#$:

$$\begin{aligned} \mathbb{S}_\omega &= \mathcal{L} \times \mathcal{M} \times \mathbb{C} \times \mathbb{A}_\mathsf{env} \times \mathbb{N}^\# \\ \mathbb{C} &= \mathbb{A}_\mathsf{env} \xrightarrow{\text{fin}} ((\mathbb{A}_\mathsf{env} \times \mathcal{L} \times \mathbb{L}) \uplus \Omega) \\ \mathbb{V} &= \mathbb{V}_\mathsf{p} \uplus \mathbb{A}_\mathsf{obj} \uplus \mathbb{F} \uplus \Omega \\ \mathbb{N}^\# &= \mathbb{A}_\mathsf{obj} \to \{0^\#, 1^\#, \geq 2^\#\} \end{aligned}$$

Because JavaScript provides open objects, the properties of objects can be dynamically added or deleted. Moreover, since object properties are string values that can be constructed at run time, it is difficult to perform sound strong updates in static analysis. To

check the possibility of strong updates during sealed execution, we augment its states with the abstract counters $\mathbb{N}^{\#}$.

For each abstract value in a given abstract state, if the abstract value denotes a single concrete value, the converter $\tau_\omega : (\Pi \times \mathbb{D}^{\#}) \to (\mathbb{M}^{\#} \times \mathbb{S}_\omega)$ keeps it; otherwise, τ_ω replaces the abstract value with its unique identifier and maintains the mapping from the unique identifier to the abstract value to construct an abstract instantiation map. The opposite converter $\tau^{\#} : (\mathbb{M}^{\#} \times \mathbb{S}_\omega) \to (\Pi \times \mathbb{D}^{\#})$ recovers abstract values from their unique identifiers using the abstract instantiation map. We define the sealed transition relation $\leadsto_\omega$ only if the next step does not require actual values of any sealed values. Otherwise, a given sealed state does not have any sealed transitions to apply. For example, we add the following rule:

$$\frac{P(\ell) = \texttt{ret}\ e \qquad \sigma_\omega \vdash_e e \Rightarrow v \qquad c(a) \in \Omega}{\sigma_\omega = (\ell, M, c, a, n^{\#}) \leadsto_\omega \bot}$$

for the `ret` statement. We extend each rule of the concrete semantics to support such behaviors of sealed values.

5 IMPLEMENTATION

We implemented JavaScript static analysis using dynamic shortcuts presented in Section 4 in a prototype implementation dubbed SAFE_{DS}. The tool is an extension of an existing state-of-the-art JavaScript static analyzer SAFE [25, 37] with a dynamic analyzer Jalangi [40], and it is an open-source project and available online [4]. In this section, we introduce challenges and solutions in implementing dynamic shortcuts on existing JavaScript analyzers.

Sealed Values. The main challenge of implementing dynamic shortcuts is to support sealed execution on an existing JavaScript engine. To represent an abstract value, we use the `Proxy` object introduced in ECMAScript 6 (2015, ES6) [7], which allows developers to handle internal behaviors of specific objects such as property reads and writes and implicit conversions. We are inspired by Mimic [18], which used `Proxy` to capture accesses from internals of opaque functions. When the dynamic analyzer constructs an execution environment at the start of a dynamic shortcut, it creates `Proxy` objects to represent abstract values via the following `getSealedValue` function:

```
1  function getSealedValue() {
2    function detect() { /* access detection */ }
3    return new Proxy(function() {}, {
4      getPrototypeOf: detect,  ...
5      construct     : detect
6    }); }
7  var x = getSealedValue();
8  var y = x;
9  var z = x + 1;
```

The function creates a sealed value as a proxy object with a dummy function object and a handler for all 13 traps using an access detection function `detect`. A sealed value invokes the function `detect` when any of 13 pre-defined traps are operated on the object, which enables us to determine whether an object is sealed or not. For example, the variable `y` successfully points to the same sealed value stored in `x`, but the program invokes the function `detect` on line 9 because `x + 1` requires the actual value of the sealed value. In addition, we instrument unary and binary operations in Jalangi so that we can detect all the accesses on the sealed value beyond the 13 traps provided by `Proxy`. Using this idea, we successfully extended the JavaScript engine to support sealed execution.

Synchronization of Control Points. For seamless interaction between static analysis and sealed execution, synchronization of control points in both sides is necessary. The SAFE static analyzer and the Jalangi dynamic analyzer have their own notations for control points that are not directly compatible. We use the source-code location of a target program as a key to synchronize. Even though they use different parsers and we faced numerous location mismatches for corner cases, we could synchronize control points of two analyzers by using the closest match of their source-code locations rather than using their exact match.

Function-Level Dynamic Shortcut. A dynamic shortcut is activated when the current abstract state passes the filter checker. Because SAFE and Jalangi are implemented in different languages, Scala and JavaScript, respectively, we represent abstract states as JSON objects and communicate between analyzers by passing JSON objects through a localhost server. If the filter admits dynamic shortcuts generously, the analysis may suffer from frequent communications between static and dynamic analyzers. To adjust such a burden, SAFE_{DS} supports only *function-level* dynamic shortcuts by activating dynamic shortcuts in function entries and deactivating them in their corresponding function exits.

Termination. To guarantee the termination of static analysis using dynamic shortcuts, the converter τ_ω should pass an analysis element $(\pi, d^{\#})$ only when it terminates in a time bound N. Since statically checking the termination property is difficult, we simply perform sealed execution with a pre-determined time limit of 5 seconds. When it times out, we treat it as a failure in conversion; otherwise, we use the result of sealed execution.

6 EVALUATION

We evaluate SAFE_{DS} using the following research questions:

- **RQ1) Analysis Speed-up:** How much analysis time is reduced by using dynamic shortcuts?
- **RQ2) Precision Improvement:** How much analysis precision is improved by using dynamic shortcuts?
- **RQ3) Opaque Function Coverage:** How many opaque functions are covered only by dynamic shortcuts?

We selected the official 306 tests of Lodash 4 (v.4.17.20)[5] used in the examples in Section 2 as our evaluation target. Recent work [28, 42] also used the tests to evaluate their techniques. Among them, we filtered out 37 tests that use JavaScript language features SAFE does not support such as dynamic code generation using `Function`, getters and setters, and browser-specific features like `__proto__`. Thus, we used 269 out of 306 tests for the evaluation of SAFE_{DS} and compared its evaluation results with those of the baseline analyzer, SAFE. For both SAFE and SAFE_{DS}, we used 400-depth, 10-length loop strings and 30-length call strings for precise analysis, and added some incomplete models for opaque functions to soundly analyze Lodash tests. We performed our experiments on a Ubuntu machine equipped with 4.2GHz Quad-Core Intel Core i7 and 32GB of RAM.

[4] https://github.com/kaist-plrg/safe-ds

[5] https://github.com/lodash/lodash/blob/4.17.20/test/test.js

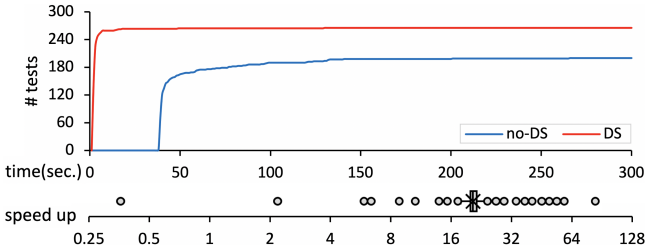


Figure 6: Analysis time for Lodash 4 *original* tests without (no-DS) and with (DS) dynamic shortcuts within 5 minutes

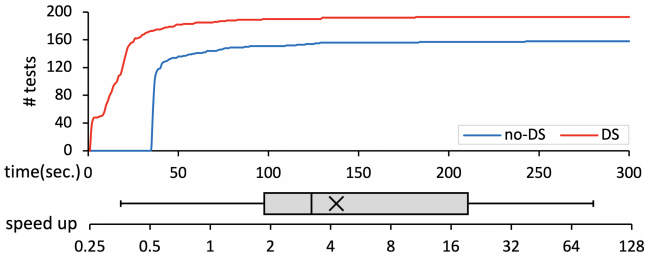


Figure 7: Analysis time for Lodash 4 *abstracted* tests without (no-DS) and with (DS) dynamic shortcuts within 5 minutes

6.1 Analysis Speed-up

We evaluated the effectiveness of dynamic shortcuts by static analysis of 269 Lodash 4 tests with and without dynamic shortcuts. Figure 6 depicts cumulative distribution charts for their analysis time and a box plot in a logarithmic scale for speed up after applying dynamic shortcuts. In the upper chart, the x-axis is time and the y-axis shows the number of tests within the time. While the baseline analysis (no-DS) finished analysis of 200 out of 269 tests within 5 minutes, our tool (DS) finished analysis of 265 tests using dynamic shortcuts. For finished tests, the average analysis time is 49.46 seconds for no-DS and 3.21 seconds for DS. Among 200 tests analyzed by no-DS, one test is timeout in DS, thus 199 tests are analyzable by both analyzers. For them, we depict the box plot for analysis speed up by dynamic shortcuts. It shows that DS outperforms no-DS up to 83.71x and 22.30x on average. Only for one test using `_.sample`, which randomly samples a value from a given array, DS showed 0.36x speed of no-DS due to 24 times uses of dynamic shortcuts.

Note that since most tests use concrete values instead of non-deterministic inputs, they can be analyzed by a few number of dynamic shortcuts. In fact, among 269 tests, 259 tests are analyzed by a single dynamic shortcut without using abstract semantics. However, in real-world JavaScript programs, arguments of library functions may include non-deterministic inputs. To evaluate SAFE_{DS} in a real-world setting, we modified the tests to use abstract values. We made abstract values by randomly selecting literals and replacing one of them with its corresponding abstract value. For example, if we select a numeric literal `42`, we modified it to the abstract numeric value $\top_{\text{num}}$, which represents all the numeric values. In the remaining section, we evaluated SAFE_{DS} using the *original* tests and the *abstracted* tests.

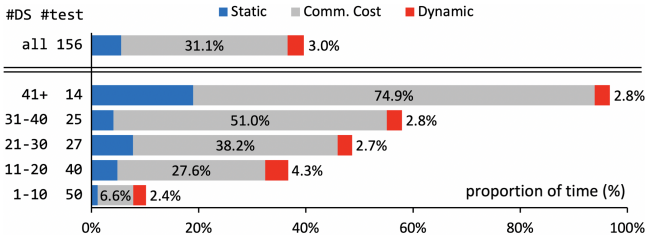


Figure 8: Analysis time ratio for 156 *abstracted* tests

For abstracted tests as well, DS outperformed no-DS. Figure 7 shows the analysis time of the abstracted tests. Among 269 abstracted tests, no-DS finished analysis of 158 tests within 5 minutes, but DS finished analysis of 193 tests. For finished tests, the average analysis time is 44.88 seconds for no-DS and 19.05 seconds for DS. Among 158 tests analyzed by no-DS, DS timed-out for 2 tests. For 156 tests analyzable by both analyzers, DS outperformed no-DS up to 78.07x and 7.81x on average. Except for 9 test cases, using dynamic shortcuts did show speed-ups.

Unlike for the original tests, analysis of 156 abstracted tests invoked 20.35 dynamic shortcuts. Because taking a dynamic shortcut requires conversion between abstract states and sealed values and their exchanges between the static analyzer and the dynamic analyzer, using dynamic shortcuts multiple times may incur more performance overhead than performance benefits by using sealed execution. One conjecture is that the communication cost between the static analyzer and the dynamic analyzer may be proportional to the number of dynamic shortcuts.

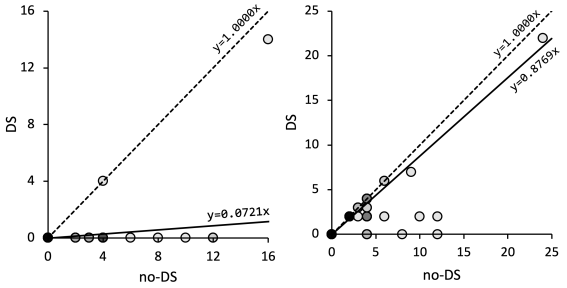
To experimentally evaluate the conjecture, we investigated the relationship between the communication cost (Comm. Cost) between analyzers and the number of dynamic shortcuts. For 199 original tests, Comm. Cost was only 1.58% compared to the analysis time of no-DS. However, for 156 abstracted tests, Comm. Cost was 31.06% compared to the analysis time of no-DS. Figure 8 presents the analysis time ratio for 156 abstracted tests. The x-axis represents the time ratio normalized by the total analysis time of no-DS and the y-axis denotes the number of dynamic shortcuts and the number of corresponding tests. For all 156 tests, Comm. Cost is larger than both the static analysis time (Static) and the dynamic analysis time (Dynamic). When dynamic shortcuts are performed less than 10 times, Comm. Cost is modest compared to the baseline static analysis time. However, the more dynamic shortcuts are performed, the less the performance benefits by using dynamic shortcuts. Specifically, when dynamic shortcuts are performed more than 30 times, Comm. Cost is even larger than half of cost of no-DS. Based on this evaluation result, we believe that we can leverage dynamic shortcuts by optimizing Comm. Cost between the static analyzer and the dynamic analyzer. One possible approach is to reduce the sizes of JSON objects that represent abstract and sealed states by representing only their updated parts. Another approach could be to use a communication system faster than a localhost server for passing JSON objects.

6.2 Precision Improvement

To evaluate the analysis precision improvement of dynamic shortcuts, we measured the number of failed assertions produced by

Table 1: Number of original (orig.) and abstracted (abs.) tests using dynamic shortcuts only for each JavaScript built-in library

Object	Function	# Replaced		Object	Function	# Replaced		Object	Function	# Replaced	
		orig.	abs.			orig.	abs.			orig.	abs.
Array	Array	204 / 205	119 / 141	String	String	20 / 20	13 / 14	Object	Object	265 / 265	181 / 193
	new Array	0 / 0	0 / 7		toString	0 / 0	0 / 14		getPrototypeOf	56 / 56	34 / 35
	isArray	264 / 265	181 / 193		valueOf	0 / 0	0 / 20		create	265 / 265	193 / 193
	concat	265 / 265	189 / 193		charAt	8 / 8	6 / 6		defineProperty	265 / 265	190 / 193
	join	265 / 265	193 / 193		charCodeAt	15 / 15	8 / 8		freeze	1 / 1	1 / 1
	pop	25 / 25	14 / 14		indexOf	2 / 2	1 / 1		keys	265 / 265	191 / 193
	push	265 / 265	186 / 193		match	26 / 26	16 / 18		toString	264 / 265	138 / 193
	reverse	10 / 10	6 / 6		replace	56 / 56	31 / 37		hasOwnProperty	265 / 265	190 / 193
	shift	3 / 3	2 / 2		slice	265 / 265	191 / 193	JSON	stringify	1 / 1	1 / 1
	slice	265 / 265	193 / 193		split	5 / 5	2 / 2	Global	parseInt	2 / 2	1 / 2
	sort	69 / 69	38 / 39		substring	214 / 214	136 / 145		isNaN	15 / 15	11 / 40
	splice	25 / 25	9 / 12		toLowerCase	215 / 215	135 / 146		isFinite	3 / 3	1 / 1
	unshift	2 / 2	2 / 2		toUpperCase	11 / 11	6 / 7	RegExp	RegExp	265 / 265	193 / 193
	indexOf	94 / 94	61 / 66		fromCharCode	1 / 1	1 / 1		new RegExp	0 / 0	0 / 1
	every	92 / 92	43 / 47	Date	new Date	0 / 1	0 / 1		exec	265 / 265	193 / 193
Math	ceil	37 / 38	20 / 21	Number	Number	2 / 2	2 / 2		test	264 / 265	185 / 193
	floor	16 / 18	8 / 10		toFixed	1 / 1	0 / 0	Error	Error	1 / 1	0 / 1
	max	264 / 265	179 / 193		valueOf	0 / 0	0 / 28		new RangeError	0 / 0	0 / 2
	min	64 / 64	31 / 44		toString	265 / 265	193 / 193		new TypeError	0 / 0	0 / 7
	pow	11 / 11	6 / 6	Function	apply	263 / 265	133 / 193	Boolean	Boolean	3 / 3	2 / 2
	round	2 / 2	1 / 1		call	259 / 265	50 / 193		valueof	0 / 0	0 / 7



(a) 199 *original* tests **(b) 156 *abstracted* tests**

Figure 9: Failed assertions of analysis without (no-DS) and with (DS) dynamic shortcuts

no-DS and DS. Because both no-DS and DS are sound, high (low) number of failed assertions denotes low (high) analysis precision.

Figure 9 depicts the comparison of the analysis precision between no-DS and DS. The x-axis and the y-axis denote the number of failed assertions produced by no-DS and DS, respectively. For example, if both DS and no-DS failed 4 assertions in an original test, the figure shows a circle at the point (4, 4). Since multiple circles can be at the same point if both DS and no-DS failed the same number of assertions, we use darker gray to denote a larger number of tests in a heat-map form. The darker the circle is, the more tests it indicates. The dotted line denotes the $y = x$ line and all the circles are below or on the line, which means DS produces less or equal numbers of assertions compared to no-DS for all tests. On the other hand, the solid line denotes the average improvement, which is the ratio of the total number of failed assertions produced by no-DS to that produced by DS. For 199 original tests that are analyzable by both analyzers, Figure 9(a) shows that dynamic shortcuts reduced the number of failed assertions by 92.79% on average. For 156 abstracted tests that are analyzable by both analyzers, Figure 9(b) shows that dynamic shortcuts successfully cut down the number of failed assertions by 12.31% on average. Thus, on average, dynamic shortcuts removed analysis of 92.79% and 12.31% failed assertions for original and abstracted tests, respectively.

6.3 Opaque Function Coverage

To evaluate how much manual modeling efforts of opaque functions are reduced by dynamic shortcuts, we measured the number of tests for which opaque functions are analyzed only by dynamic analysis not by static analysis. Table 1 summarizes the result. For 265 original tests and 193 abstracted tests that DS finished analysis, we measured the number of tests that use only dynamic shortcuts instead of manual modeling for each JavaScript built-in library function. For each row, **Object** column denotes a built-in object, **Function** a function name, and **# Replaced** the number of tests successfully replacing manual modeling via dynamic shortcuts over the total number of tests using the target function. For example, the first row in the leftmost side describes that `Array` is used in 205 original tests and 141 abstracted tests. Among them, 204 original tests and 119 abstracted tests are successfully analyzed by using dynamic shortcuts instead of manual modeling of `Array`. Each filled cell describes a fully replaceable case. Therefore, dynamic shortcuts effectively lessen the burden of manual modeling for JavaScript built-in functions. For the original tests, 45 out of 63 built-in functions are replaceable for them. For the abstracted tests, 22 built-in functions are analyzed by only dynamic shortcuts.

7 RELATED WORK

Combined Analysis. The most related previous work is combined analysis that utilizes dynamic analysis during Java static analysis introduced by Toman and Grossman [43]. They proved that their combined analysis is sound and showed that it could significantly improve the precision and performance of Java static analysis by evaluating their tool, Concerto. However, their approach has several limitations compared with dynamic shortcuts. First, it syntactically divides a given program to *applications* parts for static analysis and *frameworks* parts for dynamic analysis. Thus, it cannot freely switch between static analysis and dynamic analysis. It is even impossible to perform both static and dynamic analysis of the same program part in different contexts. In addition, while they introduced *mostly-concrete interpretation* similar to our sealed execution, it supports only a special *unknown* value that represents any

possible value. Thus, it cannot preserve the precision of complex abstract domains [23, 24, 29, 36] frequently used in JavaScript static analysis. On the contrary, sealed execution automatically detects when to switch to static analysis to use abstract semantics for abstract values. Finally, Concerto preserves the soundness when a program satisfies the *state separation hypothesis*. It assumes that the states of application parts and framework parts are not interrogated or manipulated by each other. While the assumption may be reasonable for static analysis of Java applications using external libraries, it is not satisfied for JavaScript programs in general. Unlike their approach, our approach does not have any assumptions between static and dynamic analysis parts.

Concolic Execution. Concolic execution [15] is closely related to dynamic shortcuts because it also leverages concrete execution for symbolic execution. Symbolic execution [22] is an execution of a program with symbolic values, and it can be treated as an abstract interpretation with symbolic expressions and path constraints. To resolve path constraints with symbolic expressions, symbolic execution engines such as KLEE [12] and SAGE [16] utilize Satisfiability Modulo Theory (SMT) solvers as back-end modules. On the contrary, we formalized dynamic shortcuts as a technique to combine concrete execution with a general abstract interpretation, not only with symbolic execution. Thus, dynamic shortcuts are theoretically applicable to any kind of abstract interpretation, including symbolic execution, and it is a more general definition of concolic execution.

Automatic Modeling. For static analysis of JavaScript programs, modeling behaviors of built-in libraries or host-dependent functions is necessary because they are opaque code. Since manual modeling is error-prone and labor-intensive, researchers [11, 32] have utilized type information to automatically model their behaviors. However, type is not enough to reflect complex semantics and side-effects. To alleviate the problem, Heule et al. [18] introduced a technique to infer JavaScript code for opaque code using concrete execution. They leveraged ES6 `Proxy` objects to collect partial execution traces from opaque code and synthesized JavaScript code using the extracted behaviors. Instead of synthesizing JavaScript code, Park et al. [33] presented a *Sample-Run-Abstract (SRA)* approach for on-demand modeling focusing on the current abstract states during static analysis by sampling well-distributed concrete states. However, all the previous work sacrifice the soundness of static analysis. On the contrary, while dynamic shortcuts is not always applicable to opaque functions, it is sound if it is applicable.

Pruning Analysis Scope. Another approach to utilize dynamic analysis for JavaScript static analysis is to prune the scope of analysis. Schäfer et al. [39] proposed dynamic determinacy analysis. They specialized target source code with determinacy facts so that static analysis can get benefits from elimination of `eval` and constant property names. Wei and Ryder [44] introduced *blended taint analysis*, which specializes JavaScript dynamic language features such as dynamic code generation or variadic function calls. It first performs dynamic analysis to collect traces with concrete values used in dynamic language features and restricts the semantics of features based on the collected traces during static analysis. Park et al. [34, 38] utilize three points to reduce analysis scope: initial states, dynamically loaded files, and event handlers. Unfortunately, all the above approaches except [39] do not preserve soundness of static analysis unlike our approach using dynamic shortcuts.

8 CONCLUSION

We presented a novel technique for JavaScript static analysis using *dynamic shortcuts*. It can significantly accelerate static analysis and lessen the modeling efforts for opaque code by freely leveraging high performance of dynamic analysis for concretely executable program parts. To maximize such benefits, we proposed *sealed execution*, which performs concrete execution using sealed values for abstract values. We formally defined static analysis using dynamic shortcuts in the abstract interpretation framework and proved its soundness and termination. We developed SAFE_{DS} as a prototype implementation of the proposed approach by extending a combination of the state-of-the-art static and dynamic analyzers SAFE and Jalangi. Our tool accelerates the speed of static analysis 22.30x for original tests and 7.81x for abstracted tests of Lodash 4 library. Moreover, it reduces the number of failed assertions by 12.31% by using sealed execution instead of manual modeling for 22 opaque functions on average.

ACKNOWLEDGEMENTS

This work was supported by National Research Foundation of Korea (NRF) (Grants NRF-2017R1A2B3012020 and 2017M3C4A7068177).

REFERENCES

[1] 2020. *Electron - A framework for cross-platform desktop apps with JavaScript, HTML, and CSS.* Retrieved May 25, 2021 from https://www.electronjs.org/
[2] 2020. *Espruino - An open-source JavaScript interpreter for microcontrollers.* Retrieved May 25, 2021 from https://www.espruino.com/
[3] 2020. *Lodash - A modern JavaScript library delivering modularity, performance, and extras.* Retrieved May 25, 2021 from https://lodash.com/
[4] 2020. *Moddable - Tools to create open IoT products using standard JavaScript on low cast microcontrollers.* Retrieved May 25, 2021 from https://www.moddable.com/
[5] 2020. *Node.js - A JavaScript runtime built on Chrome's V8 JavaScript engine.* Retrieved May 25, 2021 from https://nodejs.org/
[6] 2020. *React Native - A framework for building native apps using React.* Retrieved May 25, 2021 from https://reactnative.dev/
[7] 2020. *Standard ECMA-262 6th Edition, ECMAScript 2015 Language Specification.* Retrieved May 25, 2021 from https://262.ecma-international.org/6.0/
[8] 2020. *SunSpider Javascript Benchmark.* Retrieved May 25, 2021 from https://webkit.org/perf/sunspider/sunspider.html
[9] 2020. *Zoom - A videotelephony software program developed by Zoom Video Communications.* Retrieved May 25, 2021 from https://zoom.us/
[10] 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
[11] SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. 2014. SAFEWAPI: 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
[12] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vol. 8. 209–224. https://dl.acm.org/doi/10.5555/1855741.1855756
[13] 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
[14] 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

[15] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. https://doi.org/10.1145/1065010.1065036

[16] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Communications of the ACM (CACM)* 55, 3 (2012), 40–44. https://doi.org/10.1145/2093548.2093564

[17] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015. DLint: Dynamically Checking Bad Coding Practices in JavaScript. In *Proceedings of the 24th International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/2771783.2771809

[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] 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

[20] 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

[21] 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 (2018), 1–44. https://doi.org/10.1145/3230624

[22] James C King. 1976. Symbolic execution and program testing. *Communications of the ACM (CACM)* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

[23] Yoonseok Ko, Xavier Rival, and Sukyoung Ryu. 2017. Weakly Sensitive Analysis for Unbounded Iteration over JavaScript Objects. In *Proceedings of the 15th Asian Symposium on Programming Languages and Systems (APLAS)*. https://doi.org/10.1007/978-3-319-71237-6_8

[24] 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

[25] 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)*.

[26] 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

[27] 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)*. https://doi.org/10.1145/1159803.1159807

[28] 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

[29] 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

[30] 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

[31] 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

[32] 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

[33] 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

[34] 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

[35] Joonyoung Park, Jihyeok Park, Dongjun Youn, and Sukyoung Ryu. 2021. *Accelerating JavaScript Static Analysis via Dynamic Shortcuts (Extended Version)*. Technical Report. https://arxiv.org/abs/2105.13699

[36] Jihyeok Park, Xavier Rival, and Sukyoung Ryu. 2017. Revisiting Recency Abstraction for JavaScript: Towards an Intuitive, Compositional, and Efficient Heap Abstraction. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP)*. https://doi.org/10.1145/3088515.3088516

[37] 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

[38] 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

[39] 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

[40] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. 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/10.1145/2491411.2491447

[41] 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

[42] 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

[43] John Toman and Dan Grossman. 2019. Concerto: A Framework for Combined Concrete and Abstract Interpretation. In *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/3290356

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