Adding Dynamic Shortcuts on JavaScript Static Analysis

Anonymous Author(s)

Abstract

JavaScript becomes one of the most widely used programming languages for web development, server-side programming, and even micro-controllers for IoT. However, its dynamic language features degrade the scalability and precision of the static analysis. Moreover, the variety of built-in functions and host environments requires excessive manual modeling of their behaviors. To alleviate this problem, several researchers utilize dynamic analysis to take its advantage during JavaScript static analysis. However, most of them cannot fully utilize high performance of dynamic analysis and sacrifice the soundness of static analysis.

In this paper, we propose a novel *dynamic shortcut* technique to leverage high performance of dynamic analysis for static analysis by freely switching between them in a sound way. It could significantly accelerate the JavaScript static analysis by using the highly-optimized commercial JavaScript engines for the concretely executable program parts. Moreover, it might improve the precision and lessen the modeling effort by performing dynamic analysis for opaque codes. To maximize the usage of dynamic analysis, we propose *sealed symbolic execution*, which is an augmented concrete execution that represents abstract values as sealed symbolic values. We formally define the static analysis with dynamic shortcut using sealed symbolic execution. We actualize our method via SAFE_{DS}, which is an extension of the existing JavaScript static analyzer SAFE with dynamic shortcut. For evaluation of SAFE_{DS}, we analyzed abstracted versions of 77 official tests of Lodash 4 library. Our tool is X.Xx faster than the original static analysis on average. Moreover, it improves XX.XX% of analysis precision by using dynamic analysis instead of static analysis for XX.X opaque functions on average.

Keywords TODO

Woodstock '18, June 03–05, 2018, Woodstock, NY
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/10.1145/1122445.1122456

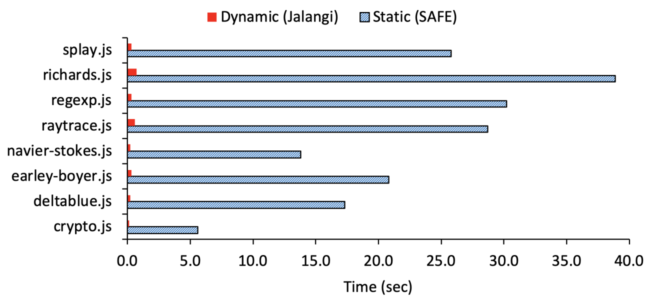


Figure 1. The performances of dynamic analysis and static analysis for V8 benchmarks (version 7)

1 Introduction

Over the past decade, the rise of JavaScript as the de facto programming language in web development has expanded its reach to diverse fields. Node.js [4] supports server-side programming, React Native [5] and Electron [6] produce cross-platform applications. Moreover, Moddable [3] and Espruino [1] provide JavaScript environments in micro-controllers for IoT. Such wide prevalent usage places JavaScript at #7 in TIOBE Programming Community index[1]. Thus, researchers have developed static analyzers such as JSAI [18], TAJS [17], WALA [42], and SAFE [29, 37] to understand behaviors of JavaScript programs and to detect their bugs in a sound way.

However, static analysis of real-world JavaScript programs suffers from its dynamic language features. JavaScript supports various *dynamic language features* such as high-order functions, first-class property names, and dynamic code generations. While they provide great flexibility in development but it is challenging to statically analyze such features. To overcome this problem, researchers have proposed several analysis techniques: advanced string domains [9, 24, 27], loop sensitivity [28, 30], property relation based analysis [21, 22, 26, 43], and on-demand backward analysis [44].

At the same time, various JavaScript host environments require excessively manual modeling of their behaviors for static analysis. Built-in functions and host-dependent functions are implemented in another native language like C++ instead of JavaScript thus their codes are *opaque* during static analysis. Thus, we need to manually model their behaviors but it is error-prone, tedious, and labor-intensive. While several automatic modeling techniques [10, 31] have been proposed, they utilize only type information thus they generate imprecise modeling compared to the manual approach.

Instead of advancing static analysis techniques, several researchers leverage dynamic analysis to resolve such obstacles

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

in static analysis. Dynamic analyzers such as Jalangi [41] and DLint [15] run on a highly-optimized commercial JavaScript engine thus they are much faster than static analysis. Figure 1 shows that the dynamic analysis (Jalangi) is X.Xx faster than static analysis (SAFE) for V8 benchmarks (version 7). Using high performance dynamic analysis, several researchers reduce the scope of static analysis [40, 46], construct initial abstract state [35, 39], and automatic modeling of opaque codes [33].

Unfortunately, existing techniques using dynamic analysis for static analysis have two limitations: 1) it cannot fully utilize the high performance of dynamic analysis, and 2) it sacrifices the soundness of static analysis. Most of existing techniques are *staged analyses* that first extract specific information via dynamic analysis and utilize it in static analysis; Schäfer et al. [40] identifies determinate expressions that always have the same value at given program point, Wei and Ryder [46] extract the dynamic values to change expressions as certain literals, and Park et al. [35, 39] dump initial states in a certain host environment or the entry of an event handler. However, they cannot utilize dynamic analysis any more after starting static analysis thus they disturb to maximize the performance advantage. Moreover, they sacrifice the soundness of static analysis to perform dynamic analysis. For example, Park et al. [33] utilize dynamic analysis for opaque codes with given abstract arguments during static analysis. However, they randomly sample finite concrete values for given abstract arguments when they represent infinite number of values. Thus, the analysis result becomes unsound because of the missing concrete values.

In this paper, we propose a novel *dynamic shortcut* technique to leverage high performance of dynamic analysis for static analysis by freely switching between them in a sound way. Our key observation is that we can use concrete semantics for specific program parts while preserving the soundness if they are not dependent with meaning of abstract values. For example, the third line of the following program passes the abstract value v stored in obj.p1 to the variable x:

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

Even though obj contains an abstract value v, the semantics of the third line does not dependent with the meaning of v. To utilize this observation in dynamic shortcut, we introduce *sealed symbolic execution*, which is an augmented concrete execution with sealed symbolic values. We represent each abstract value as a sealed symbolic value, which is a symbolic value that detects when trying to access its actual meaning. To evaluate our approach, we implemented SAFE_{DS} by extending the existing JavaScript static analyzer SAFE with dynamic shortcut. For evaluation of SAFE_{DS}, we analyzed abstracted versions of 77 official tests of Lodash 4 library.

Our contributions in this paper include the following:

- We present a novel *dynamic shortcut* for JavaScript static analysis to leverage the high performance of dynamic analysis. We also formally define it with sealed symbolic execution and prove the soundness of static analysis using dynamic shortcut.
- We actualize dynamic shortcut technique in SAFE_{DS}, which is an extension of an existing JavaScript static analyzer SAFE with a dynamic analyzer Jalangi.
- For empirical evaluation, we analyzed 77 official tests of Lodash 4 library with randomly abstracted values. Our tool accelerates X.Xx the static analysis on average. Moreover, it improves XX.XX% of analysis precision by using dynamic shortcut instead of manual modeling for XX.X opaque functions on average.

In the remainder of this paper, Section 2 explains the motivation of this work with a simple example. Section 3 formalizes adding dynamic shortcuts on abstract interpretation. We extended the formalization with JavaScript specific language features with a core language in Section 4. Section 5 describes key techniques to implement dynamic shortcuts for JavaScript static analysis. We describes evaluation result of our techniques with real-world benchmarks in Section 6. Section 7 explains related works and Section 8 concludes.

2 Motivation

In this section, we will explain the motivation of the dynamic shortcut for JavaScript static analysis with real-world examples described in Figure 2. We first describe their program behaviors and then explain how to utilize dynamic analysis during static analysis of them.

For motivating examples, we excerpt the concat function in Figure 2a from Lodash library [2] (v4.17.20), which is the most popular npm package[2] and 124,562 npm packages have dependency with it. The concat function creates a new array concatenating given arrays or values. It first checks the length of arguments in line 1-3. Then, it stores the first argument to array in line 4 and copies the remaining arguments to args in line 5-8. In line 9, it checks whether array is an array object with the built-in function isArray. If so, it creates a new array by copying the given array via or initializing with a single value in line 10. Finally, it flattens args via baseFlatten and pushes the result to the new array in line 11.

For applications of Lodash, we excerpt two functions changeCountry in Figure 2b and getData in Figure 2c from the zoom.us [8] site. The website zoom.us is homepage of Zoom, which is a videotelephony software program developed by Zooom Video Communications and it is ranked as 16th popular web site according to Alexa[3] in November 2020.

Dynamic shortcut with concrete values. When the given arguments of a function are concrete values, we can perform dynamic analysis instead of static analysis. For example, the

[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   ...
15   if (G.selectedVal === "US" && state) {
16     // deterministic arguments of `concat`
17     state.items = _.concat([["Other", "Other"]],
18       WebinarBase.questions.state.items);
19     state.selectedVal = _.head(_.head(C.items));
20   }
21 }
```

(b) Load the list of states of the United States.

```
22 function getData(e) {
23   var option = ... // option of 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) Load more Zoom events from the server.

Figure 2. Motivating Example: Excerpts from Lodash library and JavaScript codes in zoom.us site.

changeCountry function is invoked when a user selects another country on the drop-down list in the registration page. It calls the concat function to update the drop-down list of states or provinces in 17-18. However, when the user selects "United States of America" (USA), 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 value is also a concrete value, the Lodash top-level object _. Thus, we could perform dynamic analysis by invoking the concat function with _ as this value and above two concrete values as arguments. It increases performance of static analysis by skipping the analysis of function call in line 17-18 and utilizing the result of dynamic analysis.

Property	Value
$\top$	$\omega_{\texttt{evt}}$
"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 for arguments with sealed symbolic values.

Dynamic shortcut with abstract values. Dynamic analysis is still applicable using *sealed symbolic execution* even if the arguments are not concrete values. The getData function is invoked when clicking the "Load More" button to load more Zoom events in "Webinars & Events" page. For each click, the getData sends a POST request to the server and receives additional event information e in line 24. Then, eight events in e.events are appended to this.pastEvents using the concat function in line 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) also each event stored in e.events are dependent on the data given from the server.

To perform dynamic analysis with abstract values, we sealed the abstract values of arguments with symbolic values as described in Figure 3. It contains two symbolic values $\omega_{\texttt{evt}}$ and $\omega_{\texttt{int}}$ that represent any event objects and integer values, respectively. Then, dynamic analysis is successfully performed before copying array via copyArray in line 10. First, length stores 2 and passes the length check in line 1-2. Then, array points to the same object of this.pastEvents in line 4, args stores an array with a single object stored in e.events in 5-8, and the isArray function returns true for array in line 9. However, it fails to perform dynamic analysis for copyArray because the length property of array is the symbolic value $\omega_{\texttt{int}}$. Thus, the dynamic shortcut returns the analysis result and the static analyzer continues to analyze the program from line 10. Then, only copying via copyArray, flattening via baseFlatten, and pushing via arrayPush utilize the abstract semantics. This is how to utilize sealed symbolic execution to maximize the part of dynamic analysis during static analysis.

Dynamic shortcut for opaque functions. The previous two examples also show that dynamic shortcut can improve the analysis precision and lessen the effort of modeling the opaque functions. In line 9, the isArray function is a JavaScript built-in library thus it is written in a native language of the host environment. Thus, we need to manually model its behavior to statically analyze it. Assume that we model the isArray function to return the top boolean value that denotes both of true and false. If we perform static analysis with this modeling, both of true branch copyArray(array) and false branch [array] in line 10 are always analyzed while [array] is never reachable in the motivating examples. However, with the dynamic shortcut, the analyzer utilizes the concrete semantics of `isArray`. It returns more precise result `true` instead of the top boolean value and it is not necessary to model the `isArray` function for static analysis of motivating examples.

3 Dynamic Shortcut

In this section, we formally define the dynamic shortcut with sealed symbolic execution. We extend the formalization of abstract interpretation of Cousot and Cousot [13, 14] and views-based analysis sensitivity of Kim et al. [19].

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 could calculate it using the *transfer function* $F : \mathbb{D} \rightarrow \mathbb{D}$ as follows:

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

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 element d_ι denotes the initial states $\mathbb{S}_\iota$. The *one-step execution* $\mathsf{step} : \mathbb{D} \rightarrow \mathbb{D}$ transforms states using the transition relation $\rightsquigarrow$: $\mathsf{step}(d) = \{\sigma' \mid \sigma \in d \wedge \sigma \rightsquigarrow \sigma'\}$.

$$\begin{array}{l}
\bullet_{\ell_0}\ \texttt{x = ? ;}\ \bullet_{\ell_1} \\
\texttt{if (x} \geq \texttt{0)}\ \bullet_{\ell_2}\ \texttt{x = x;} \\
\texttt{if}\ \bullet_{\ell_3}\ \texttt{x = -x;}\ \bullet_{\ell_4}
\end{array}$$

Figure 4. Conditional branch

For example, the code in Figure 4 is a simple program that applies the mathematical absolute value function to the variable x. The question mark ? denotes the user input that returns an integer. States are pairs of labels and integers x: $\mathbb{S} = \mathcal{L} \times \mathbb{N}$. The initial states are $\mathbb{S}_\iota = \{(\ell_0, 0)\}$ which means the program starts with the variable x that stores 0 at ℓ_0. If the user input is -42, the program is executed with the following trace:

$$(\ell_0, 0) \rightsquigarrow (\ell_1, -42) \rightsquigarrow (\ell_3, -42) \rightsquigarrow (\ell_4, 42)$$

3.2 Abstract Interpretation

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

$$[\![P]\!]^\# = \lim_{n \rightarrow \infty} (F^\#)^n(d^\#_\iota)$$

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 $F^\# : \mathbb{D}^\# \rightarrow \mathbb{D}^\#$ is defined as $F^\#(d^\#) = d^\# \sqcup \mathsf{step}^\#(d^\#)$ with an *abstract one-step execution* $\mathsf{step}^\# : \mathbb{D}^\# \rightarrow \mathbb{D}^\#$. For the sound state abstraction, the join operator and the abstract one-step execution should satisfy the following conditions:

- $\forall d^\#_0, d^\#_1 \in \mathbb{D}^\#.\ \gamma(d^\#_0) \cup \gamma(d^\#_1) \subseteq \gamma(d^\#_0 \sqcup d^\#_1)$
- $\forall d^\# \in \mathbb{D}^\#.\ \mathsf{step}^\# \circ \gamma(d^\#) \subseteq \gamma \circ \mathsf{step}^\#(d^\#)$

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. If we use it for the code in Figure 4, the analysis result becomes $\{-, +, 0\}$ because x can have any integers at ℓ_1.

3.3 Analysis Sensitivity

Abstract interpretation is often defined with *analysis sensitivity* to increase the precision of static analysis. A sensitive abstract domain $\mathbb{D}^\#_\delta : \Pi \rightarrow \mathbb{D}^\#$ is defined with a *view abstraction* $\delta : \Pi \rightarrow \mathbb{D}$ 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 $\delta(\pi)$. 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:

$$\gamma_\delta(d^\#_\delta) = \{\sigma \in \mathbb{S} \mid \forall \pi \in \Pi.\ \sigma \in \delta(\pi) \Rightarrow \sigma \in \gamma \circ d^\#_\delta(\pi)\}$$

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

$$\mathsf{step}^\#_\delta(d^\#_\delta) = \lambda \pi \in \Pi.\ \bigsqcup_{\pi' \in \Pi} [\![\pi' \rightarrow \pi]\!]^\# \circ d^\#_\delta(\pi')$$

where $[\![\pi' \rightarrow \pi]\!]^\# : \mathbb{D}^\# \rightarrow \mathbb{D}^\#$ is the 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:

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

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

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

If we apply the flow sensitivity for the above example, the analysis result becomes as follows:

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

3.4 Sealed Symbolic Execution

To handle abstract values in concrete semantics, we define *sealed symbolic execution* by extending the transition relation $\rightsquigarrow$ as a symbolic transition relation $\rightsquigarrow_\omega$ on symbolic states. First, we extends the concrete states $\mathbb{S}$ to symbolic states with *sealed symbolic values* Ω. A symbolic transition relation

$\rightsquigarrow_\omega \subseteq \mathbb{S}_\omega \times (\mathbb{S}_\omega \uplus \{\bot_\sigma\})$ is exactly the same with the original transition relation $\rightsquigarrow$ except when the relation requires the exact value of sealed symbolic values. In this case, the symbolic state has the relation a special exception state $\bot_\sigma$ to represent that it is impossible to interpret in a concrete way. Thus, a symbolic state *always* has a symbolic transition relation with a single symbolic state, and the sealed symbolic execution is linear until being terminated or reaching the exception state $\bot_\sigma$.

The main difference of sealed symbolic execution with the traditional symbolic execution [20] is that it only supports sealed symbolic values instead of symbolic expressions and path constraints. For example, the following trace represents the traditional symbolic execution of the running example in Figure 4:

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

It first assigns a symbolic value ω to the variable x at ℓ_1. For the conditional branch, it is forked by creating 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_4. However, the sealed symbolic execution stops at ℓ_1 as follows:

$$(\ell_0, 0) \rightsquigarrow_\omega (\ell_1, \omega) \rightsquigarrow_\omega \bot_\sigma$$

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

To freely convert a pair of view and an abstract state to its corresponding symbolic state and vice versa, we define two domain converters $\mathbb{S}_\omega \underset{\tau^\#}{\overset{\tau_\omega}{\leftrightarrows}} (\Pi \times \mathbb{D}^\#)$. In the converter τ_ω from abstract states to symbolic states, if an abstract value $v^\#$ represents a singleton value, the function transforms the abstract value to the corresponding concrete value. Otherwise, it converts the abstract value as a symbolic value in a result symbolic state. Two converters convert given elements without loss of information:

$$\tau^\# \circ \tau_\omega = \tau_\omega \circ \tau^\# = \text{id}$$

where id denotes the identity function.

3.5 Dynamic Shortcut

We define the the abstract interpretation with *dynamic shortcut*, which is a concrete semantics on sealed symbolic execution. With dynamic shortcut, we use the powerset of symbolic states as the domain $\widetilde{\mathbb{D}} = \mathcal{P}(\mathbb{S}_\omega)$, and extend the view transition as follows:

$$\begin{aligned} &\widetilde{[\![\pi \to \pi']\!]}(\widetilde{d}) = \\ &\quad \{\sigma'_\omega \mid \sigma_\omega \in S \wedge \sigma_\omega \rightsquigarrow_\omega \sigma'_\omega \wedge \tau^\#(\sigma'_\omega) = (\pi', \_)\} \\ &\quad \cup \begin{cases} \{\tau_\omega \circ [\![\pi \to \pi']\!]^\#(\bigsqcup D)\} & \text{if } D \neq \varnothing \\ \varnothing & \text{otherwise} \end{cases} \end{aligned}$$

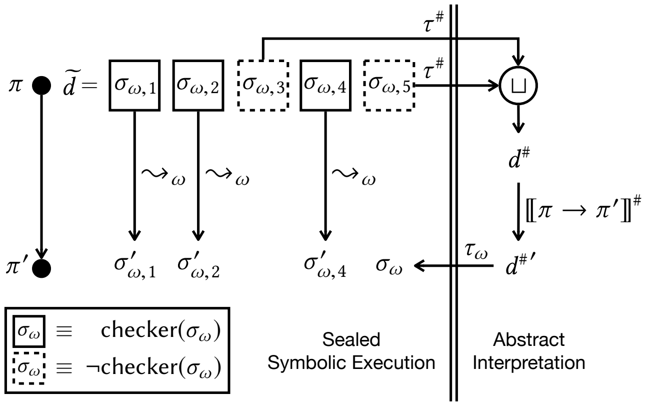


Figure 5. The diagram for the extended view transition for dynamic shortcut.

where $S = \widetilde{d}\ |_{\text{checker}}$ and $D = \dot{\tau}^\#(\widetilde{d}\ |_{\neg\text{checker}})$. The dot notation $\dot{f}$ denotes the element-wise extended function of a function f. The abstract interpretation with dynamic shortcut performs one-step symbolic transition for each symbolic state that passes the filter checker. On the other hand, it converts remaining symbolic states to the corresponding abstract states, merges them into a single abstract state, and performs the abstract one-step execution for the abstract state.

For example, Figure 5 depicts the diagram of the extended view transition from π to π' for dynamic shortcut. The element $\widetilde{d}$ in π consists of five symbolic states from $\sigma_{\omega,1}$ to $\sigma_{\omega,5}$. The solid box denotes that the corresponding symbolic state passes the filter checker and the dotted box denotes that it does not. For each symbolic state that passes the filter ($\sigma_{\omega,1}$, $\sigma_{\omega,2}$, and $\sigma_{\omega,4}$), it applies the symbolic transition $\rightsquigarrow_\omega$. For failed symbolic states ($\sigma_{\omega,3}$ and $\sigma_{\omega,5}$), it applies the converter $\tau^\#$ to convert each of them to the corresponding abstract state and merges results using the join operator ($\sqcup$). Then, it utilizes the abstract semantics via the original view transition $[\![\pi \to \pi']\!]^\#$ and the converter τ_ω to convert the result to the corresponding symbolic state.

We could configure when the sealed symbolic execution or the abstract interpretation is performed based on the definition of the checker function and its negation ¬checker. For example, we could define the checker that passes only function entry, call, and exit points to perform conversions in a function level. However, for the soundness and the termination of the the abstract interpretation with dynamic shortcut, it should satisfy the following condition:

Theorem 3.1. *The abstract interpretation with dynamic shortcut is sound and terminates in a finite time if its abstract semantics is sound and checker satisfies the following condition:*

$$\textit{checker}(\sigma_\omega) \Rightarrow \sigma_\omega \rightsquigarrow^k_\omega \bot_\sigma \wedge 1 < k \leq N$$

where N is a pre-defined maximum length of the sealed symbolic execution.

$$\boxed{\sigma \rightsquigarrow \sigma}$$

$$\frac{P(\ell) = r = e \qquad \sigma \vdash_r r \Rightarrow l \qquad \sigma \vdash_e e \Rightarrow v}{\sigma = (\ell, m, c, a) \rightsquigarrow (\text{next}(\ell), m[l \mapsto v], c, a)} \qquad \frac{\begin{array}{c} P(\ell) = r = \{\} \\ \sigma \vdash_r r \Rightarrow l \qquad a' = \text{(a fresh object address)} \end{array}}{\sigma = (\ell, m, c, a) \rightsquigarrow (\text{next}(\ell), m[l \mapsto a'], c, a)}$$

$$\frac{\begin{array}{c} P(\ell) = r = e_f(e_a) \qquad \sigma \vdash_r r \Rightarrow l \qquad \sigma \vdash_e e_f \Rightarrow \lambda x.\ell_b \\ \sigma \vdash_e e_a \Rightarrow v_a \qquad a' = \text{(a fresh environment address)} \end{array}}{\sigma = (\ell, m, c, a) \rightsquigarrow (\ell_b, m[(a', x) \mapsto v_a], c[a' \mapsto (a, \text{next}(\ell), l)], a')} \qquad \frac{P(\ell) = \texttt{ret}\ e \qquad \sigma \vdash_e e \Rightarrow v \qquad c(a) = (a', \ell', l)}{\sigma = (\ell, m, c, a) \rightsquigarrow (\ell', m[l \mapsto v], c, a')}$$

$$\frac{P(\ell) = \texttt{if}\ e\ \ell' \qquad \sigma \vdash_e e \Rightarrow \texttt{true}}{\sigma = (\ell, m, c, a) \rightsquigarrow (\ell', m, c, a)} \qquad \frac{P(\ell) = \texttt{if}\ e\ \ell' \qquad \sigma \vdash_e e \Rightarrow \texttt{false}}{\sigma = (\ell, m, c, a) \rightsquigarrow (\text{next}(\ell), m, c, a)}$$

$$\boxed{\sigma \vdash_r r \Rightarrow l}$$

$$\sigma = (\ell, m, c, a) \vdash_r x \Rightarrow (a, x) \qquad \frac{\sigma \vdash_e e_0 \Rightarrow a_0 \qquad \sigma \vdash_e e_1 \Rightarrow v_1 \qquad v_1 \in \mathbb{V}_{\text{str}}}{\sigma = (\ell, m, c, a) \vdash_r e_0[e_1] \Rightarrow (a_0, v_1)}$$

$$\boxed{\sigma \vdash_e e \Rightarrow v}$$

$$\sigma = (\ell, m, c, a) \vdash_e v_{\text{p}} \Rightarrow v_{\text{p}} \qquad \sigma = (\ell, m, c, a) \vdash_e \lambda x.\ell' \Rightarrow \lambda x.\ell' \qquad \frac{\sigma \vdash_r r \Rightarrow l \qquad l \in \text{Dom}(m)}{\sigma = (\ell, m, c, a) \vdash_e r \Rightarrow m(l)}$$

$$\frac{\sigma \vdash_e e_1 \Rightarrow v_1 \qquad \cdots \qquad \sigma \vdash_e e_n \Rightarrow v_n}{\sigma = (\ell, m, c, a) \vdash_e \text{op}(e_1, \cdots, e_n) \Rightarrow \text{op}(v_1, \cdots, v_n)}$$

Figure 6. The transition relation for the core language of JavaScript

We proved Theorem 3.1 using the Kleene's fixed-point theorem [11] with the finite height of abstract domains. Because of the page limitation, we omit the proof in this paper and include it in a companion report [38].

4 Dynamic Shortcut 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 symbolic execution of the core language for dynamic shortcut.

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	$e ::= v_{\text{p}} \mid \lambda x.\ \ell \mid r \mid \text{op}(e^*)$

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

States $\mathbb{S}$ consist of labels $\mathcal{L}$, memories $\mathbb{M}$, contexts $\mathbb{C}$, and environment addresses $\mathbb{A}_{\text{env}}$. A memory $m \in \mathbb{M}$ 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 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 the function body.

$$\boxed{\llbracket \pi \to \pi' \rrbracket^\# : \mathbb{D}^\# \to \mathbb{D}^\#}$$

$$\frac{\begin{array}{c} d^\# = (m^\#, c^\#, a^\#, n^\#) \\ P(\ell) = r = e \qquad \llbracket r \rrbracket_r^\#(d^\#) = L \qquad \llbracket e \rrbracket_e^\#(d^\#) = v^\# \end{array}}{\llbracket \ell \to \mathsf{next}(\ell) \rrbracket^\#(d^\#) = (m^\#[L \mathrel{\dot{\mapsto}} v^\#], c^\#, a^\#, n^\#)}$$

$$\frac{\begin{array}{c} d^\# = (m^\#, c^\#, a^\#, n^\#) \\ P(\ell) = r = \{\} \qquad \llbracket r \rrbracket_r^\#(d^\#) = L \qquad a^\#_{\mathsf{obj}} = \ell \end{array}}{\llbracket \ell \to \mathsf{next}(\ell) \rrbracket^\#(d^\#) = (m^\#[L \mathrel{\dot{\mapsto}} \{a^\#_{\mathsf{obj}}\}], c^\#, a^\#, \mathsf{inc}(n^\#, a^\#_{\mathsf{obj}}))}$$

$$\frac{\begin{array}{c} d^\# = (m^\#, c^\#, a^\#, n^\#) \qquad P(\ell) = r = e_f(e_a) \\ \llbracket r \rrbracket_r^\#(d^\#) = L \qquad \lambda x.\ell_b \in \llbracket e_f \rrbracket_e^\#(d^\#) \qquad \llbracket e_a \rrbracket_e^\#(d^\#) = v_a^\# \\ a^\#_{\mathsf{env}} = \ell_b \qquad c^\#_{\mathsf{env}} = c^\#[a^\#_{\mathsf{env}} \mapsto c^\#(a^\#_{\mathsf{env}}) \cup \{(a^\#, \mathsf{next}(\ell), L)\}] \end{array}}{\llbracket \ell \to \ell_b \rrbracket^\#(d^\#) = (m^\#[(a^\#_{\mathsf{env}}, x) \mapsto v_a^\#], c^\#_{\mathsf{env}}, a^\#_{\mathsf{env}}, \mathsf{inc}(n^\#, a^\#_{\mathsf{env}}))}$$

$$\frac{\begin{array}{c} d^\# = (m^\#, c^\#, a^\#, n^\#) \\ P(\ell) = \mathtt{ret}\ e \qquad \llbracket e \rrbracket_e^\#(d^\#) = v^\# \qquad (a^\#_{\mathsf{ret}}, \ell', L) \in c^\#(a^\#) \end{array}}{\llbracket \ell \to \ell' \rrbracket^\#(d^\#) = (m^\#[L \mathrel{\dot{\mapsto}} v^\#], c^\#, a^\#_{\mathsf{ret}}, n^\#)}$$

$$\frac{P(\ell) = \mathtt{if}\ e\ \ell' \qquad \mathtt{true} \in \llbracket e \rrbracket_e^\#(d^\#)}{\llbracket \ell \to \ell' \rrbracket^\#(d^\#) = d^\#} \qquad \frac{P(\ell) = \mathtt{if}\ e\ \ell' \qquad \mathtt{false} \in \llbracket e \rrbracket_e^\#(d^\#)}{\llbracket \ell \to \mathsf{next}(\ell) \rrbracket^\#(d^\#) = d^\#}$$

$$\boxed{\llbracket r \rrbracket_r^\# : \mathbb{D}^\# \to \mathcal{P}(\mathbb{L}^\#)}$$

$$\frac{d^\# = (m^\#, c^\#, a^\#, n^\#)}{\llbracket x \rrbracket_r^\#(d^\#) = \{(a^\#, x)\}} \qquad \frac{\begin{array}{c} d^\# = (m^\#, c^\#, a^\#, n^\#) \\ A = \llbracket e_0 \rrbracket_e^\#(d^\#) \cap \mathbb{A}^\# \qquad S = \llbracket e_1 \rrbracket_e^\#(d^\#) \cap \mathbb{V}_{\mathsf{str}} \end{array}}{\llbracket e_0[e_1] \rrbracket_r^\#(d^\#) = A \times S}$$

$$\boxed{\llbracket e \rrbracket_e^\# : \mathbb{D}^\# \to \mathbb{V}^\#}$$

$$\llbracket v_{\mathsf{p}} \rrbracket_e^\#(d^\#) = \{v_{\mathsf{p}}\} \qquad \llbracket \lambda x.\ell \rrbracket_e^\#(d^\#) = \{\lambda x.\ell\} \qquad \frac{d^\# = (m^\#, c^\#, a^\#, n^\#) \qquad v^\# = \bigsqcup \{m^\#(l^\#) \mid l^\# \in \llbracket r \rrbracket_r^\#(d^\#)\}}{\llbracket r \rrbracket_e^\#(d^\#) = v^\#}$$

$$\frac{\llbracket e_1 \rrbracket_e^\#(d^\#) = v_1^\# \qquad \cdots \qquad \llbracket e_n \rrbracket_e^\#(d^\#) = v_n^\#}{\llbracket \mathsf{op}(e_1, \cdots, e_n) \rrbracket_e^\#(d^\#) = \dot{\mathsf{op}}(v_1^\#, \cdots, v_n^\#)}$$

Figure 7. The semantics of view transition for the core language of JavaScript

We formulate the concrete semantics of the core language as described in Figure 6. 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. The function next returns the next label of a given label in the current program P.

4.2 Abstract Semantics

In 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 set of singleton abstract addresses as follows:

Abstract States	$d^\# \in \mathbb{D}^\#$	$= \mathbb{M}^\# \times \mathbb{C}^\# \times \mathbb{A}^\# \times \mathbb{N}^\#$
Abstract Memories	$m^\# \in \mathbb{M}^\#$	$= \mathbb{L}^\# \xrightarrow{\text{fin}} \mathbb{V}^\#$
Abstract Contexts	$c^\# \in \mathbb{C}^\#$	$= \mathbb{A}^\# \xrightarrow{\text{fin}} \mathcal{P}(\mathbb{A}^\# \times \Pi \times \mathcal{P}(\mathbb{L}^\#))$
Abstract Locations	$l^\# \in \mathbb{L}^\#$	$= (\mathbb{A}^\# \times \mathbb{X}) \uplus (\mathbb{A}^\# \times \mathbb{V}_{\mathsf{str}})$
Abstract Values	$v^\# \in \mathbb{V}^\#$	$= \mathcal{P}(\mathbb{V}_{\mathsf{p}} \uplus \mathbb{A}^\# \uplus \mathbb{F})$
Abstract Addresses	$a^\# \in \mathbb{A}^\#$	$= \mathcal{L}$
Abstract Counters	$n^\# \in \mathbb{N}^\#$	$= \mathbb{A}^\# \to \{0^\#, 1^\#, \geq 2^\#\}$

An abstract memory $m^\# \in \mathbb{M}^\#$ 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 [25, 36] in static analysis, we define abstract counters $\mathbb{N}^\#$ that maps 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 equals to twice.

We define the semantics of view transition for the core language in Figure 7. For abstract memories, we use the notation $m^\#[L \mapsto v^\#]$ 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 soundness. Moreover, we define the increment function $\mathrm{inc} : \mathbb{N}^\# \times \mathbb{A}^\# \to \mathbb{N}^\#$ of the abstract counter defined as follows:

$$\mathrm{inc}(n^\#)(a^\#_0) = \lambda a^\# \in \mathbb{A}^\#. \begin{cases} 1^\# & \text{if } a^\# = a^\#_0 \land n^\#(a^\#_0) = 0^\# \\ \geq 2^\# & \text{if } a^\# = a^\#_0 \land n^\#(a^\#_0) = 1^\# \\ n^\#(a^\#) & \text{otherwise} \end{cases}$$

4.3 Sealed Symbolic Execution

We define the symbolic states with the extended concrete values and the abstract counters for addresses as follows:

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

We define the converter $\tau_\omega : (\Pi \times \mathbb{D}^\#) \to \mathbb{S}_\omega$ from abstract states to corresponding symbolic states as follows:

$$\tau_\omega((\ell, (m^\#, c^\#, a^\#, n^\#))) = (\ell, \tau^{\mathbb{M}}_\omega(m^\#), \tau^{\mathbb{C}}_\omega(c^\#), \tau^{\mathbb{A}}_\omega(a^\#), n^\#)$$

The converter for memories $\tau^{\mathbb{M}}_\omega$ is defined with the converter for values $\tau^{\mathbb{V}}_\omega$ where:

$$\tau^{\mathbb{V}}_\omega(v^\#) = \begin{cases} v & \text{if } v^\# = \{v\} \\ v^\# & \text{otherwise} \end{cases}$$

In a similar way, the converter for contexts $\tau^{\mathbb{C}}_\omega$ is defined as follows:

$$\tau^{\mathbb{C}}_\omega(c^\#)(a^\#) = \begin{cases} (a^\#, \ell, l) & \text{if } c^\#(a^\#) = \{(a^\#, \ell, \{l\})\} \\ c^\#(a^\#) & \text{otherwise} \end{cases}$$

The opposite converter $\tau^\# : \mathbb{S}_\omega \to (\Pi \times \mathbb{D}^\#)$ from symbolic states to abstract states is defined as the inverse function of τ_ω: $\tau^\# = \tau^{-1}_\omega$. The symbolic transition relation $\rightsquigarrow_\omega$ is an extension of the original concrete transition relation $\rightsquigarrow$. When the exact values of symbolic values are required, symbolic states have the symbolic transition relation with the exception state $\bot_\sigma$. For example, we add the following rule for `ret` statements:

$$\frac{P(\ell) = \mathtt{ret}\ e \qquad \sigma \vdash_e e \Rightarrow v \qquad c(a) \in \Omega}{\sigma = (\ell, m, c, a) \rightsquigarrow \bot_\sigma}$$

We extend each rule of concrete semantics to support such behaviors of symbolic values.

5 Implementation

We have implemented the dynamic shortcut for JavaScript static analysis presented in Section 4 in a prototype implementation called SAFE$_{\mathrm{DS}}$. The tool is an extension of an existing state-of-the-art JavaScript static analyzer SAFE [29, 37] with a dynamic analyzer Jalangi [41], and it is open source and available online [4]. In this section, we introduce challenges and solutions when implementing dynamic shortcut on existing JavaScript analyzers.

Sealed Symbolic Values. The main challenge of implementing dynamic shortcut is to support the sealed symbolic execution on the existing JavaScript engine. We leverage the `Proxy` object introduced in ECMAScript 6 (2015, ES6) [7], which allows developers to handle the internal behaviors of specific objects (such as property reads and writes, implicit conversions, etc.). When the dynamic analyzer constructs the execution environments at the start of the dynamic shortcut, it creates revoked `Proxy` objects to represent abstract values via the following `generateSymbol` function:

```
1 function generateSymbol() {
2   const r = Proxy.revocable(function() {}, {});
3   r.revoke();
4   return r.proxy;
5 }
6 var x = generateSymbol();
7 var y = x;
8 var z = x + 1;
```

The function creates a sealed symbol as a revocable proxy object with a dummy function object without any handlers and revokes it by `revocable.revoke()` to not allow any access of the symbol. Thus, the variable `y` successfully points to the same symbol stored in `x` but the program throws a `TypeError` in line 8 because `x + 1` requires the actual value of the symbol. Using this idea, we successfully extend the JavaScript engine to support a sealed symbolic execution.

Abstract Locations for Objects. During dynamic analysis, each JavaScript object should be designated by the corresponding abstract location used in the static analysis. In this paper, we define abstract locations using the allocation site abstraction [12] with heap cloning [23] for each abstract context in the static analyzer. Thus, we instrument the given JavaScript program to annotate each object created during dynamic analysis with its allocation site and context information. After the sealed symbolic execution, the dynamic analyzer collects the objects based on their annotated information for each abstract location.

Optimizations. To perform dynamic shortcut, the static analyzer should communicate with the dynamic analyzer by passing the current abstract state and receiving the final result of the sealed symbolic execution. When abstract states are massive, it is burden to send them as they stand. Thus,

[4]The URL of the tool is anonymized due to a double-blind review process.

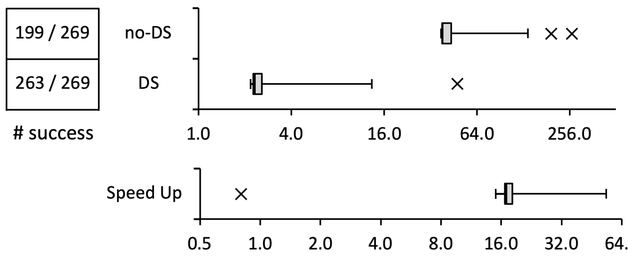


Figure 8. Analysis time for Lodash 4 *original* tests without (no-DS) and with (DS) the dynamic shortcut within 5 minutes.

we perform mark-and-sweep garbage collection to remove unreachable information in the current abstract state. Moreover, the abstract values are converted to the sealed symbolic values and their contents are not necessary in the dynamic analysis. Thus, we replace abstract values with their unique identifiers and stores the table from identifiers to abstract values before sending the abstract state from the static analyzer. Thus, the dynamic analyzer only receives identifiers of abstract values and wrap them as `Proxy` objects. After finishing the sealed symbolic execution, the static analyzer recovers the abstract values in the result abstract state using the table.

Termination. To guarantee the termination of static analysis with dynamic shortcut, the filter checker should pass the state σ_ω only when it terminates in a bound time N. Since it is difficult to statically check its termination, we just performs the sealed symbolic execution with a time limit. When it runs over the time limit, we treat it fails to pass checker, otherwise we utilizes the result of the execution. In the experiments, we set the time limit as 5 seconds for each sealed symbolic execution.

6 Evaluation

We evaluated our tool based on the following research questions:

- **RQ1) Analysis Speed-up:** How much analysis time is reduced by adding dynamic shortcut to static analysis?
- **RQ2) Precision Improvement:** How much analysis precision is improved by replacing the manual modeling with dynamic shortcut?
- **RQ3) Opaque Function Coverage:** How many opaque functions are covered by dynamic shortcut without using manual modeling?

We targeted the official 306 tests of Lodash 4 (v.4.17.20)[5] used in motivating examples (Section 2). The most recent papers for JavaScript static analysis techniques [26, 44] also evaluated their techniques based on them. Among them, we filtered out 37 tests including JavaScript language features SAFE does not support, such as dynamic code generation using `Function`, getter/setter, and browser specific features (e.g. `__proto__`). Thus, we only targeted 269 out of 306 tests for

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

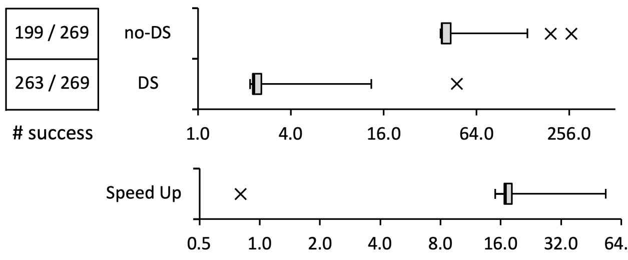


Figure 9. Analysis time for Lodash 4 *abstracted* tests without (no-DS) and with (DS) the dynamic shortcut within 5 minutes.

the evaluation. We performed our experiments on an Ubuntu machine equipped with 4.2GHz Quad-Core Intel Core i7 and 64GB of RAM.

6.1 Analysis Speed-up

To evaluate the effectiveness of the dynamic shortcut, we performed static analysis for 269 Lodash 4 tests with and without dynamic shortcut. Figure 8 depicts the box plot chart for their analysis time and for speed up after applying the dynamic shortcut in a logarithmic scale. While the base analysis (no-DS) finished 199 out of 269 tests within 5 minutes, our tool finished all tests with the dynamic shortcut (DS). For 199 tests analyzable by both analyses, the analysis took 168.4 seconds and 12.2 seconds on average without dynamic shortcut (no-DS) and with dynamic shortcut (DS), and the dynamic shortcut accelerates 20.16x the static analysis on average. Only for one test using `_.sample` (a Lodash 4 library that randomly samples a value from a given array), the static analysis using dynamic shortcut had 0.81x speed of the base analysis because of the frequent use of dynamic shortcut (24 times).

Unfortunately, since most of Lodash 4 tests use concrete values instead of non-deterministic user inputs, they could be analyzed via a few number of dynamic shortcut. In fact, among 269 tests, 262 tests analyzed via a single usage of dynamic shortcut without using abstract semantics. However, the arguments of library functions might include non-deterministic user inputs in the real-world JavaScript programs. Thus, we modified Lodash 4 official tests with abstract values to mimic the use patterns of library functions. We randomly selected literals and replace subset of them to their corresponding typed abstract values. For example, if we pick a numerical literal `42`, we modified it to the abstract numeric value $\top_{\mathsf{num}}$, which represents the all numerical values. In the remaining section, we evaluated our tool based on the randomly abstracted tests of Lodash 4.

For abstracted tests, the dynamic shortcut successfully accelerates the static analysis. Figure 9 shows the analysis time for the abstracted tests in a logarithmic scale. Among 269 abstracted tests, the base analysis (no-DS) finished 82 tests within 5 minutes. On the other hand, the analysis using

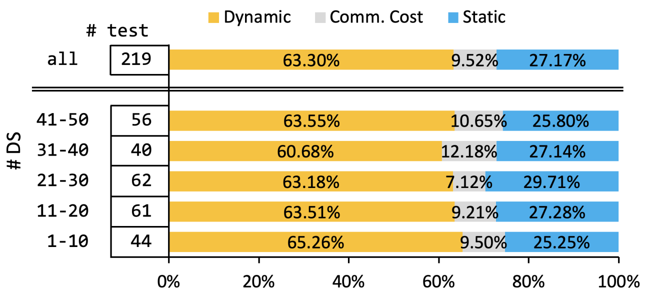
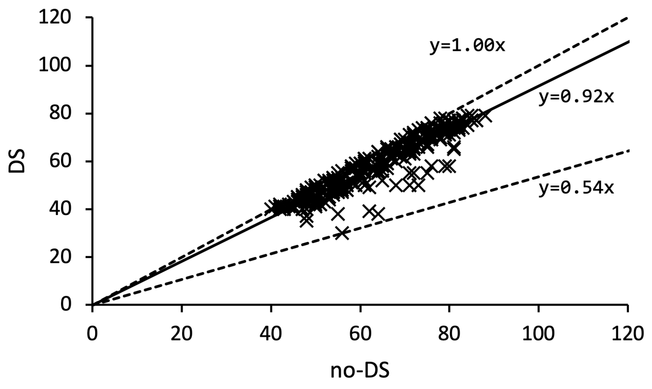
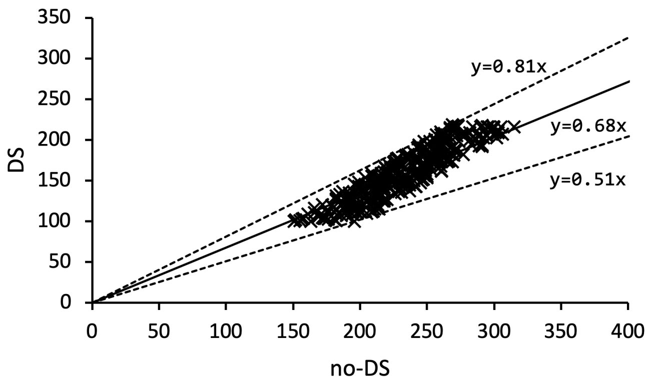


Figure 10. Time ratio of the finished analysis with dynamic shortcut for 219 Lodash 4 *abstracted* tests.



(a) # Failed Assertions.



(b) # Covered Branches.

Figure 11. The comparison of analysis precision without (no-DS) and with (DS) dynamic shortcut for 82 *abstracted* tests analyzable by both analyses

dynamic shortcut (DS) finished 219 tests. For 82 tests analyzable by both analyses, the analyses took 168.4 seconds and 12.2 seconds on average without dynamic shortcut (no-DS) and with dynamic shortcut (DS), and the dynamic shortcut accelerates 20.16x the static analysis on average.

For 219 abstracted tests successfully analyzed with the dynamic shortcut, communication costs between static and dynamic analysis grew relatively to the number of dynamic shortcuts. To communicate with the dynamic analyzer, the static analysis should dump the current abstract state, send it to the dynamic analyzer, receive the result from the dynamic analysis, and apply the given update information to the abstract state. Thus, the communication costs will increase dependent on the number of dynamic shortcuts. Figure 1 shows the time ratio of dynamic and static analysis parts with communication costs in total analysis time. On average, the dynamic analysis occupied 63.30%, communication 9.52%, and static analysis 27.17% in total analysis time. If the dynamic shortcuts are performed less than 10 times, the ratio of communication in total analysis time is 9.50% on average. The more dynamic shortcuts performed, the more communication cost is required. When the number is between 41 and 50, the ratio of average communication cost is rised to 10.65% in the total analysis time.

6.2 Precision Improvement

To evaluate the precision improvement of dynamic shortcut, we used two different metrics to measure the precision of the analysis results: 1) the number of false alarms for assertions in tests and 2) the number of branch covered by analysis. Figure 11 depicts the comparison of analysis precision without (no-DS) and with (DS) dynamic shortcut using two different metrics.

Lodash 4 official tests check the assertions based on QUnit[6], which is a powerful, easy-to-use JavaScript testing framework. Since all assertions in each test are passed in the concrete execution, any assertion failures during static analysis should be false alarms. Although we analyzed *abstracted* Lodash 4 tests, the number of assertion failures during static anlysis could be a metric of imprecision of the analysis. Figure 11a shows the number of assertion failures in each test analyzable by both analysis with and without dynamic shortcut. The x- and y-axis denote the analysis without (no-DS) and with (DS) dynamic shortcut and each × mark denotes each test. The top and bottom dotted lines denote the worst and best precision improvement, and the middle solid line denotes the average improvement. Thus, dynamic shortcut reduces assertion failures to at least 1.00x, at most 0.54x, and 0.92x on average.

Another metric to measure the precision of static analysis is the number of branches covered during analysis. If the static analysis is sound, the more precise analysis covers the less number of branches. In the similar way with the number of assertion failures, Figure 11b depicts the the comparison of the number of branches during static analysis without (no-DS) and with (DS) dynamic shortcut. In the view point of the branch coverage, dynamic shortcut successfully cut down the number of covered branches at least 0.81x, at most 0.51x, and 0.68x on average.

6.3 Opaque Function Coverage

TODO

[6]https://qunitjs.com/

Object	Function	# Replaced
Boolean	`Boolean`	214 / 214 (100%)
	`new Boolean`	214 / 214 (100%)
Array	`isArray`	214 / 214 (100%)
Array .prototype	`toString`	214 / 214 (100%)
	`toLocaleString`	214 / 214 (100%)
	`concat`	214 / 214 (100%)
	`join`	214 / 214 (100%)
	`pop`	214 / 214 (100%)
	`push`	214 / 214 (100%)
	`slice`	214 / 214 (100%)
	`splice`	214 / 214 (100%)
	`map`	214 / 214 (100%)
	`reduce`	214 / 214 (100%)
global	`eval`	214 / 214 (100%)
Date	`new Date`	214 / 214 (100%)
	`Date.parse`	214 / 214 (100%)
	`Date.now`	214 / 214 (100%)
Object	`new Object`	214 / 214 (100%)
	`getPrototypeOf`	214 / 214 (100%)
	`create`	214 / 214 (100%)
	`preventExtensions`	214 / 214 (100%)
	`isFrozen`	214 / 214 (100%)
	`isExtensible`	214 / 214 (100%)
	`keys`	214 / 214 (100%)
Object .prototype	`valueOf`	214 / 214 (100%)
	`isPrototypeOf`	214 / 214 (100%)
Math	`acos`	214 / 214 (100%)
	`atan2`	214 / 214 (100%)
	`cos`	214 / 214 (100%)
	`exp`	214 / 214 (100%)
	`floor`	214 / 214 (100%)
	`max`	214 / 214 (100%)
	`min`	214 / 214 (100%)
Number .prototype	`valueOf`	214 / 214 (100%)
RegExp .prototype	`toString`	214 / 214 (100%)
String	`String`	214 / 214 (100%)
	`fromCharCode`	214 / 214 (100%)
String	`charAt`	214 / 214 (100%)
	`charCodeAt`	214 / 214 (100%)
	`indexOf`	214 / 214 (100%)
	`match`	214 / 214 (100%)
	`replace`	214 / 214 (100%)
	`slice`	214 / 214 (100%)
	`split`	214 / 214 (100%)
	`substring`	214 / 214 (100%)
	`toLowerCase`	214 / 214 (100%)
	`toUpperCase`	214 / 214 (100%)

Table 1. TODO

7 Related Work

Several researchers have introduced analysis technique to utilize dynamic analysis for static analysis in three different ways: combined analysis, automatic modeling, and pruning states.

Combined Analysis The most related previous work is the combined analysis that utilizes dynamic analysis during Java static analysis introduced by Toman and Grossman [45]. 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 have several limitations compared to the dynamic shortcut. First, they syntactically divides a given program to *applications* parts for static analysis and *frameworks* parts for dynamic analysis. Thus, it is impossible to freely switching between static and dynamic and even impossible to perform both of static and dynamic analysis to the same program part in different contexts. Besides, they introduced *mostly-concrete interpretation* similar with our sealed symbolic execution. However, it supports only a special *unknown* value that represents any possible value. Thus, it cannot preserves the precision of complex abstract domains [22, 27, 36] frequently used in JavaScript static analysis. On the other hand, the sealed symbolic execution automatically detects whether the abstract semantics are required for abstract values. Finally, Concertopreserves the soundness when the program satisfies the *state separation hypothesis*. They assume that states of application and framework parts do not interrogated or manipulated by each other. While it is reasonable for static analysis of Java applications using external libraries, the assumption is not satisfied for JavaScript programs in general. Unlike their approach, our approach does not have any assumption between static and dynamic analysis parts.

Automatic Modeling For static analysis of JavaScript programs, modeling behaviors of built-in libraries or host-dependent functions is required because they are opaque codes. Since manual modeling is error-prone, tedious, and labor-intensive, several researchers [10, 32] utilize type information to automatically model their behaviors. However, type information is too imprecise to reflect the detailed semantics and is difficult to represent the side-effects. To alleviate the problem, Heule et al. [16] introduced the technique to infer JavaScript codes for opaque codes using concrete execution. They captured the effects of opaque codes on user objects by collecting partial execution traces and synthesized JavaScript codes based on the extracted behaviors. They also leveraged ES6 `Proxy` objects to capture the effects on user objects. Instead of synthesizing JavaScript codes, Park et al. [34] presented a *Sample-Run-Abstract (SRA)* approach for on-demand modeling thus they focused on the current abstract state during static analysis. It first *samples* concrete states in a well-distributed way, *runs* each sampled state on a JavaScript engine, and *abstract* executions results. However, all of previous works sacrifice the soundness of static analysis. On the other hand, while the dynamic shortcut is not applicable for all invocation of opaque functions, it is sound if it is applicable.

Pruning Analysis Scopes Another approach to utilize dynamic analysis for JavaScript static analysis is to prune the scope of analysis. Wei and Ryder [47] introduced *blended taint analysis* that specializes JavaScript dynamic language features such as dynamic code generation (e.g. `eval`) or variadic function calls. They first performs dynamic analysis to collect traces with concrete values used in dynamic language features and restrict the semantics of features based on the collected traces during static analysis. Park et al. [35, 39] utilizes three different points to reduce analysis scopes: initial states, dynamically loaded files, and event handlers. They dumped the initial states from a specific web browser to focus on analyzing the behaviors of web browsers running on the browser. Then, they collected paths of dynamically loaded files via concrete executions and utilized the path information in static analysis. For event handlers, they intentionally analyzed partial execution flows using concrete user events. They collected concrete states for each entry of event handler during dynamic analysis, merged them to a single abstract state, and analyzed the event handler with the abstract state. Unfortunately, all of them does not preserve soundness of static analysis unlike the dynamic shortcut.

8 Conclusion

We have presented a novel technique *dynamic shortcut* for JavaScript static analysis. It could significantly accelerate the JavaScript static analysis by freely leveraging high performance of dynamic analysis for the concretely executable program parts. To maximize the usage of dynamic analysis, we augmented the concrete execution to *sealed symbolic execution* with symbolic values for abstract values required abstract semantics. We formally defined the static analysis with dynamic shortcut using sealed symbolic execution and proved its soundness on the formalization. We developed SAFE_{DS} as a prototype implementation of the dynamic shortcut by combining state-of-the-art static and dynamic analyzers SAFE and Jalangi. Our tool accelerates the speed of static analysis X.Xx for original tests and X.Xx for abstracted tests of Lodash 4 library. Moreover, it improves XX.XX% of analysis precision by using dynamic analysis instead of static analysis for XX.X opaque functions on average.

References

[1] 2020. *Espruino - an open-source JavaScript interpreter for microcontrollers.* Retrieved November 20, 2020 from https://www.espruino.com/

[2] 2020. *Lodash: a modern JavaScript library delivering modularity, performance, and extras.* Retrieved November 20, 2020 from https://lodash.com/

[3] 2020. *Moddable - Tools to create open IoT products using standard JavaScript on low cast microcontrollers.* Retrieved November 20, 2020 from https://www.moddable.com/

[4] 2020. *Node.js - A JavaScript runtime built on Chrome's V8 JavaScript engine.* Retrieved November 20, 2020 from https://nodejs.org/

[5] 2020. *React Native - A framework for building native apps using React.* Retrieved November 20, 2020 from https://reactnative.dev/

[6] 2020. *React Native - A framework for cross-platform desktop apps with JavaScript, HTML, and CSS.* Retrieved November 20, 2020 from https://www.electronjs.org/

[7] 2020. *Standard ECMA-262 6th Edition, ECMAScript 2015 Language Specification.* Retrieved November 20, 2020 from http://ecma-international.org/ecma-262/6.0

[8] 2020. *Zoom: videotelephony software program developed by Zoom Video Communications.* Retrieved November 20, 2020 from https://zoom.us/

[9] 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 *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 41–57.

[10] 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*. 507–517.

[11] Andrei Baranga. 1991. The contraction principle as a particular case of Kleene's fixed point theorem. *Discrete Mathematics* 98, 1 (1991), 75–79.

[12] David R Chase, Mark Wegman, and F Kenneth Zadeck. 1990. Analysis of pointers and structures. *ACM SIGPLAN Notices* 25, 6 (1990), 296–310.

[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*. 238–252.

[14] Patrick Cousot and Radhia Cousot. 1992. Abstract interpretation frameworks. *Journal of logic and computation* 2, 4 (1992), 511–547.

[15] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015. DLint: dynamically checking bad coding practices in JavaScript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 94–105.

[16] Stefan Heule, Manu Sridharan, and Satish Chandra. 2015. Mimic: Computing models for opaque code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 710–720.

[17] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *In Proceedings of the International Symposium on Static Analysis*. 238–255.

[18] 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 *In Proceedings of the International Symposium on Foundations of Software Engineering*. 121–132.

[19] 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.

[20] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[21] Yoonseok Ko, Xavier Rival, and Sukyoung Ryu. 2017. Weakly sensitive analysis for unbounded iteration over JavaScript objects. In *Asian Symposium on Programming Languages and Systems*. Springer, 148–168.

[22] Yoonseok Ko, Xavier Rival, and Sukyoung Ryu. 2019. Weakly sensitive analysis for JavaScript object-manipulating programs. *Software: Practice and Experience* 49, 5 (2019), 840–884.

[23] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. *ACM SIGPLAN Notices* 42, 6 (2007), 278–289.

[24] Magnus Madsen and Esben Andreasen. 2014. String analysis for dynamic field access. In *International Conference on Compiler Construction*. Springer, 197–217.

[25] Matthew Might and Olin Shivers. 2006. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*. 13–25.

[26] Benjamin Barslev Nielsen and Anders Møller. 2020. Value Partitioning: A Lightweight Approach to Relational Static Analysis for JavaScript. In *Proc. 34th European Conference on Object-Oriented Programming (ECOOP)*.

[27] 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*. 25–36.

[28] 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* 48, 4 (2018), 911–944.

[29] Changhee Park and Sukyoung Ryu. 2015. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. In *In Proceedings of 29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 37)*, John Tang Boyland (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 735–756. https://doi.org/10.4230/LIPIcs.ECOOP.2015.735

[30] Changhee Park and Sukyoung Ryu. 2015. Scalable and precise static analysis of JavaScript applications via loop-sensitivity. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[31] Jihyeok Park. 2014. JavaScript API misuse detection by using typescript. In *Proceedings of the companion publication of the 13th international conference on Modularity*. 11–12.

[32] Jihyeok Park. 2014. JavaScript API misuse detection by using typescript. In *Proceedings of the companion publication of the 13th international conference on Modularity*. 11–12.

[33] Joonyoung Park, Alexander Jordan, and Sukyoung Ryu. 2019. Automatic Modeling of Opaque Code for JavaScript Static Analysis. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 43–60.

[34] Joonyoung Park, Alexander Jordan, and Sukyoung Ryu. 2019. Automatic Modeling of Opaque Code for JavaScript Static Analysis. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 43–60.

[35] Joonyoung Park, Inho Lim, and Sukyoung Ryu. 2016. Battles with false positives in static analysis of JavaScript web applications in the wild. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 61–70.

[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*. 1–6.

[37] Jihyeok Park, Yeonhee Ryou, Joonyoung Park, and Sukyoung Ryu. 2017. Analysis of JavaScript web applications using SAFE 2.0. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 59–62.

[38] Jihyeok Park and Sukyoung Ryu. 2020. *The soundness and termination of dynamic shortcut*. Technical Report.

[39] Joonyoung Park, Kwangwon Sun, and Sukyoung Ryu. 2018. EventHandler-Based Analysis Framework for Web Apps Using Dynamically Collected States. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 129–145.

[40] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Dynamic determinacy analysis. *Acm Sigplan Notices* 48, 6 (2013), 165–174.

[41] 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 2013 9th Joint Meeting on Foundations of Software Engineering*. 488–498.

[42] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation Tracking for Points-to Analysis of JavaScript. In *In Proceedings of the European Conference on Object-Oriented Programming*.

[43] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation tracking for points-to analysis of JavaScript. In *European Conference on Object-Oriented Programming*. Springer, 435–458.

[44] Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. 2019. Static analysis with demand-driven value refinement. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[45] John Toman and Dan Grossman. 2019. Concerto: a framework for combined concrete and abstract interpretation. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[46] Shiyi Wei and Barbara G Ryder. 2013. Practical blended taint analysis for JavaScript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 336–346.

[47] Shiyi Wei and Barbara G Ryder. 2013. Practical blended taint analysis for JavaScript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 336–346.