EventHandler-Based Analysis Framework for Web Apps Using Dynamically Collected States

Joonyoung Park[1(✉)], Kwangwon Sun[2], and Sukyoung Ryu[1(✉)]

[1] KAIST, Daejeon, Republic of Korea
{gmb55,sryu.cs}@kaist.ac.kr
[2] Samsung Electronics, Seoul, Republic of Korea
kwangwon.sun@samsung.com

Abstract. JavaScript web applications (apps) are prevalent these days, and quality assurance of web apps gets even more important. Even though researchers have studied various analysis techniques and software industries have developed code analyzers for their own code repositories, statically analyzing web apps in a sound and scalable manner is challenging. On top of dynamic features of JavaScript, abundant execution flows triggered by user events make a sound static analysis difficult.

In this paper, we propose a novel *EventHandler* (*EH*)-based static analysis for web apps using dynamically collected state information. Unlike traditional whole-program analyses, the *EH*-based analysis intentionally analyzes partial execution flows using concrete user events. Such analyses surely miss execution flows in the entire program, but they analyze less infeasible flows reporting less false positives. Moreover, they can finish analyzing partial flows of web apps that whole-program analyses often fail to finish analyzing, and produce partial bug reports. Our experimental results show that the *EH*-based analysis improves the precision dramatically compared with a state-of-the-art JavaScript whole-program analyzer, and it can finish analysis of partial execution flows in web apps that the whole-program analyzer fails to analyze within a timeout.

Keywords: JavaScript · Web applications · Event analysis Static analysis

1 Introduction

Web applications (apps) written in HTML, CSS, and JavaScript have become prevalent, and JavaScript is now the 7th most popular programming language [22]. Because web apps can run on any platforms and devices that provide any browsers, they are being used widely. The overall structure of web apps is specified in HTML, which is represented as a tree structure via Document Object Model (DOM) APIs. CSS describes visual effects like colors, positions, and animation of contents of the web app, and JavaScript handles events triggered by user interaction. JavaScript code can change the status of the web app by interoperation with HTML and CSS, load other JavaScript code dynamically, and access device-specific features via APIs provided by underlying platforms. JavaScript is the *de facto* standard language for web programming these days.

To help developers build high-quality web apps, researchers have studied various analysis techniques and software industries have developed in-house static analyzers. Static analyzers such as SAFE [12,15], TAJS [2,10], and WALA [19] analyze JavaScript web apps without concretely executing them, and dynamic analyzers such as Jalangi [20] utilize concrete values obtained by actually executing the apps. Thus, static analysis results aim to cover all the possible execution flows but they often contain infeasible execution flows, and dynamic analysis results contain only real execution flows but they often struggle to cover abundant execution flows. Such different analysis results are meaningful for different purposes: *sound* static analysis results are critical for verifying absence of bugs and *complete* dynamic analysis results are useful for detecting genuine bugs. In order to enhance the quality of their own software, IT companies develop in-house static analyzers like Infer from Facebook [4] and Tricorder from Google [18].

However, statically analyzing web apps in a sound and scalable manner is extremely challenging. Especially because JavaScript, the language that handles controls of web apps, is totally dynamic, purely static analysis has various limitations. While JavaScript can generate code to execute from string literals during evaluation, such code is not available for static analyzers before run time. In addition, dynamically adding and deleting object properties, and treating property names as values make statically analyzing them difficult [17]. Moreover, since execution flows triggered by user events are abundant, statically analyzing them often incurs analysis performance degradation [16].

Among many challenges in statically analyzing JavaScript web apps, we focus on analysis of event-driven execution flows in this paper. Most existing JavaScript static analyzers are focusing on analysis of web apps at loading time and they over-approximate event-driven execution flows to be sound. In order to consider all possible event sequences soundly, they abstract the event-driven semantics in a way that any events can happen in any order. Such a sound event modeling contains many infeasible event sequences, which lead to unnecessary operations computing imprecise analysis results. Thus, the state-of-the-art JavaScript static analyzers often fail to analyze event flows in web apps.

In this paper, we propose a novel *EventHandler-based (*EH*-based) static analysis* for web apps using *dynamically collected state information*. First, we present a new analysis unit, an *EH*. While traditional static analyzers perform whole-program analysis covering all possible execution flows, the *EH*-based analysis aims to analyze *partial* execution flows triggered by user events more precisely. In other words, unlike the whole-program analysis that starts analyzing from a single entry point of a given program, the *EH*-based analysis considers each event function call triggered by a user event as an entry point. Because the *EH*-based analysis enables a subset of the entire execution flows to be analyzed at a time, it can analyze less infeasible execution flows than the whole-program analysis, which balances soundness and precision. Moreover, since it considers a smaller set of execution flows, it may finish analysis of web apps that the whole-program analysis fails to analyze within a reasonable timeout. Second, in order to analyze each event function call in arbitrary call contexts, we present a hybrid approach to construct an abstract heap for the event function call. More specifically, to analyze each event function body, the analyzer should have information about non-local variables. Thus, for each event function, we construct a conservative abstract initial heap that holds abstract values of non-local variables by abstraction of dynamically collected states.

We formally present the mechanism as a framework, EHA, parameterized by a dynamic event generator and a static whole-program analyzer. After describing the high-level structure of EHA, we present its prototype implementation, $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$, instantiated with manual event generation and a state-of-the-art JavaScript static analyzer SAFE. Our experimental results show that $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ indeed reports less false positives than SAFE, and it can finish analysis of parts of web apps that SAFE fails to analyze within the timeout of 72 h.

Our paper makes the following contributions:

– We propose EHA, a bug detection framework that performs static analysis for each event handler as an entry point using an abstraction of dynamically collected states as an initial heap.
– We present $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$, an instantiation of EHA with manual event generation and SAFE, which is applicable to real-world web apps.
– We evaluate $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ in terms of analysis coverage and precision.

The remainder of this paper is organized as follows. We first explain the concrete semantics of event handlers in web apps, describe how existing whole-program analyzers handle events in a sound but unscalable manner, and present an overview of our approach using concrete code examples in Sect. 2. We describe EHA and its prototype implementation in Sect. 3 and Sect. 4, respectively. We evaluate the EHA instance using real-world web apps in Sect. 5, discuss related work in Sect. 6, and conclude in Sect. 7 with future work.

2 Analyses of Event Handlers

2.1 Event Handlers in Web Apps

Web apps may receive *events* from their execution environments like browsers or from users[1]. When a web app receives an event, it reacts to the event by executing JavaScript code registered as a handler (or a listener) of the event. An *event handler* consists of three components: an event target, an event type, and a callback function. An event target may be any DOM object like `Element`, `window`, and `XMLHttpRequest`. An event type is a string representation of the event action type such as `"load"`, `"click"`, and `"keydown"`. Finally, a callback function is a JavaScript function to be executed when its corresponding event occurs.

[1] http://www.w3schools.com/js/js_events.asp.

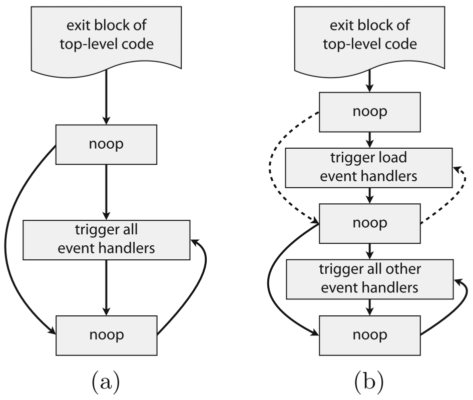

exit block of top-level code
noop
trigger all event handlers
noop
(a)
exit block of top-level code
noop
trigger load event handlers
noop
trigger all other event handlers
noop
(b)

Fig. 1. (a) A conservative modeling of event control flows (b) Modeling in TAJS [9]

Users execute web apps by triggering various events, thus we consider sequences of events triggered by users as user inputs to web apps. During execution, a set of event handlers that can be executed by a user may vary. First, because event handlers are dynamically registered to and removed from DOM objects, executable event handlers for an event change at run time. For example, when a DOM object has only the following event handler registered:

```
(A, "click", function f(){ B.addEventListener("click", function g(){}); })
```

if a user clicks the target `A`, a new event handler becomes registered, which makes two handlers executable. Second, changes in DOM states of a web app also change a set of executable event handlers for an event. For instance, an event target may be removed from `document` via DOM API calls, which makes the detached event target inaccessible from users. Also, events may not be captured depending on their capturing/bubbling options and CSS style settings of `visibility` or `display`. In addition, it is a common practice to manipulate CSS styles like the following:

- `HTMLElement.style.opacity = 0;`
- `HTMLElement.style.zIndex = n;`

to hide an element such as a button under another element, making it inaccessible from users. These various features affect event sequences that users can trigger and event handlers that are executed accordingly.

2.2 Analysis of Event Handlers in Whole-Program Analyzers

Most existing whole-program JavaScript analyzers handle event handlers in a sound but unscalable manner as illustrated in Fig. 1(a). They first analyze top-level code that is statically available in a given web app; event handlers may be registered during the analysis of top-level code. Then, after the "exit block of top-level code" node, they analyze code initiated by event handlers in any order as denoted by the "trigger all event handlers" node in any number of times. According to this modeling of event control flows, all possible event sequences that occur after loading the top-level code are soundly analyzed. Note that even though whole-program analyzers use this sound event modeling, the analyzers themselves may not be sound because of other features like dynamic code generation. However, because registered event handlers may be removed during evaluation and they may be even inaccessible due to some CSS styles as discussed in Sect. 2.1, the event modeling in Fig. 1(a) may contain too many infeasible event sequences that are impossible in concrete executions. Analysis with lots of infeasible event sequences involves unnecessary computation that wastes analysis time, and often results in imprecise analysis results. Such a conservative modeling of event control flows indeed reports many false positives [16].

To reduce the amount of infeasible event sequences to analyze, TAJS uses a refined modeling of event control flows as shown in Fig. 1(b). Among various event handlers, this modeling distinguishes "load event handlers" and analyzes them before all the other event handlers. While this modeling is technically unsound because non-load events may precede load events [15], most web apps satisfy this modeling in practice. Moreover, because load event handlers often initialize top-level variables, the event modeling in Fig. 1(a) often produces false positives by analyzing non-load event functions before load event functions initialize top-level variables. On the contrary, the TAJS modeling reduces such false positives by analyzing load event handlers before non-load event handlers. Although the TAJS modeling distinguishes a load event, the over-approximation of the other event handler calls still brings analysis precision and scalability issues.

2.3 Analysis of Event Handlers in *EH*-Based Analyzers

To alleviate the analysis precision and scalability problem due to event modeling, we propose the EHA framework, which aims to analyze a subset of execution flows within a limited time budget to detect bugs in partial execution flows rather than to analyze all execution flows. EHA presents two key points to achieve the goal. First, it slices the entire execution flows by using each event handler as an individual entry point, which amounts to consider a given web app as a collection of smaller web apps. This slicing brings the effect of breaking the loop structures in existing event modelings shown in Fig. 1. Second, in order to analyze sliced event control flows in various contexts, EHA constructs an initial abstract heap of each entry point that contains necessary information to analyze a given event control flow by abstracting dynamically collected states. More specifically, EHA takes two components—a *dynamic event generator* and a *static analyzer*—and collects concrete values of non-local variables of event functions via the dynamic event generator, and abstracts the collected values using the static analyzer.

Let us compare static, dynamic, and *EH*-based analyses with an example. We assume that a top-level code registers three event handlers: l, a, and b where l

denotes a load event handler, which precedes the others and runs once. In addition, a and b simulate a pop-up and its close button, respectively. Thus, we can represent possible event sequences as a regular expression: $l(ab)^*a?$. For a given event sequence *lababa*, Fig. 2 represents the event flows analyzed by each analysis technique. A conservative static analysis contains infeasible event sequences like the ones starting with a or b, whereas a dynamic analysis covers only short prefixes out of infinitely many flows. The *EH*-based analysis slices the web app into three handler units: l, a, and b. Hence, there is no loop in the event modeling; each handler considers every prefix of the given event sequence that ends with itself. For example, the handler a considers *la*, *laba*, and *lababa* as possible event sequences. Moreover, instead of abstracting the evaluation result of each sequence separately and merging them, it first merges the evaluation result of each sequence just before the handler a—l, *lab*, and *labab*—and uses its abstraction as the initial heap of analyzing a, which analyzes more event flows.

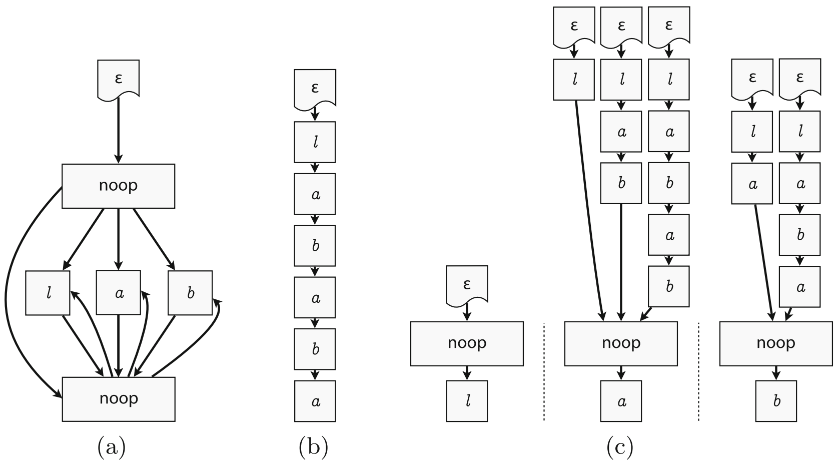


Fig. 2. Event flows analyzed by (a) static, (b) dynamic, and (c) *EH*-based analyses.

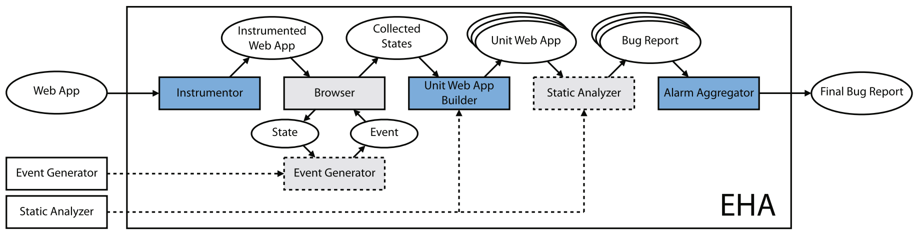


Fig. 3. Overall structure of EHA

3 Technical Details

This section discusses the EHA framework, which composes of five phases as shown in Fig. 3. Boxes denote modules and ellipses denote data. EHA takes three inputs: a web app (Web App) to analyze and find bugs in it, and two modules to use as its components—a dynamic event sequence generator (Event Generator) and a static analyzer (Static Analyzer). During the first *instrumentation* phase, Instrumentor inserts code that dynamically collects states into the input web app. Then, during the *execution* phase, the Instrumented Web App runs on a browser producing Collected States. One of the input module Event Generator repeatedly receives states of the running web app and sends user events to it during this phase. In the third *unit building* phase, Unit Web App Builder constructs a small Unit Web App for each event handler from Collected States. After analyzing the set of Unit Web Apps by another input module Static Analyzer in the *static analysis* phase, Alarm Aggregator summarizes the resulting set of Bug Reports and generates a Final Bug Report for the original input Web App in the final *alarm aggregation* phase. We now describe each phase in more detail.

$$
\begin{array}{lcl}
\mathit{Inst}\ (h \equiv \texttt{<head>}) & = & h\texttt{.addChildFront(<script src="helper" />)} \\
\mathit{Inst}\ (\texttt{function}\ f\texttt{(}\cdots\texttt{)}\ b) & = & \texttt{function}\ f\texttt{(}\cdots\texttt{)\{} \\
\multicolumn{3}{l}{\qquad\qquad \texttt{var envId} = \texttt{getNewEnvId(); var nonlocals = } \{x_1'\texttt{:}x_1\ \cdots\}\texttt{;}} \\
\multicolumn{3}{l}{\qquad\qquad \texttt{pushCallStack(); collectState(nonlocals);}\ b\texttt{; popCallStack(); \}}} \\
\mathit{Inst}\ (\texttt{return}\ x\texttt{;}) & = & \texttt{\{ var retVal = }x\texttt{; popCallStack(); return retVal; \}} \\
\mathit{Inst}\ (\texttt{catch(}e\texttt{)\{}\ b\ \texttt{\}}) & = & \texttt{\{ popCallStack();}\ b\ \texttt{\}} \\
\mathit{Inst}\ (x \texttt{ = } e) & = & x \texttt{ = } e\texttt{; update(}x'\texttt{,}x\texttt{)} \\
\mathit{Inst}\ (x \oplus) & = & x \oplus \texttt{; update(}x'\texttt{,}x\texttt{,} \oplus \texttt{)} \\
\mathit{Inst}\ (\oplus\ x) & = & \oplus\ x\texttt{; update(}x'\texttt{,}x\texttt{)}
\end{array}
$$

Fig. 4. Instrumentation rules (partial)

Instrumentation Phase. The first phase instruments a given web app so that the instrumented web app can record dynamically collected states during execution. Figure 4 presents the instrumentation rules for the most important cases where the unary operator $\oplus$ is either `++` or `--`. For presentation brevity, we abuse the notation and write x' to denote the string representation of a variable name x. The *Inst* function converts necessary JavaScript language constructs to others that perform dynamic logging. For example, for each function declaration of f, *Inst* inserts four statements before the function body and one statement after the function body to keep track of non-local variables of the function f.

Execution Phase. The execution phase runs an instrumented web app on a browser using events generated by Event Generator. Because EHA is parameterized by the input Event Generator, it may be an automated testing tool or manual efforts. The following definitions formally specify the concepts being used in the execution phase and the rest of this section:

$$\begin{array}{llllll} \textit{Execution} & \sigma \in \mathbb{S}^* & \textit{State} & s \in \mathbb{S} = \mathbb{P} \times \mathbb{H} & \textit{ProgramPoint} & p \in \mathbb{P} \\ \textit{Heap} & h \in \mathbb{H} = \mathbb{A} \to \mathbb{O} & \textit{Address} & @x \in \mathbb{A} & \textit{Object} & \mathbb{O} = \mathbb{F} \to \mathbb{V} \\ \textit{Field} & x \in \mathbb{F} & \textit{Value} & \mathbb{V} = \mathbb{V}_b \uplus \mathbb{A} & \textit{PrimitiveValue} & \mathbb{V}_b \end{array}$$

An execution of a web app σ is a sequence of states that are results of evaluation of the web app code. We omit how states change according to the evaluation of different language constructs, but focus on which states are collected during execution. A state s is a pair of a program point p denoting the source location of the code being evaluated and a heap h denoting a memory status. A heap is a map from addresses to objects. An address is a unique identifier assigned whenever an object is created, and an object is a map from fields to values. A field is an object property name and a value is either a primitive value or an address that denotes an object. For presentation brevity, we abuse *Object* to represent *Environment* as well, which is a map from variables to values. Then, EHA collects states at event callback entries during execution:

$$\textit{Collected States}(\sigma) = \{s \mid s \in \sigma \text{ s.t. } s \text{ is at an event callback entry}\}$$

the program points of which are function entries and the call stack depths are 1.

Unit Building Phase. As shown in Fig. 3, this phase constructs a set of sliced unit web apps using dynamically collected states. More specifically, it divides the collected states into *EH* units, and then for each *EH* unit u, it constructs an *initial summary* $\hat{s}_I^u$ that contains merged values about non-local variables from the states in u. As discussed in Sect. 2.1, an event handler consists of three components: an event target, an event type, and a callback function. Thus, we design an *EH* unit u with an abstract event target ϕ, an event type τ, and a program point p:

$$\begin{array}{ll} u \in \mathbb{U} & = \textit{AbsEventTarget} \times \textit{EventType} \times \mathbb{P} \\ \phi \in \textit{AbsEventTarget} & = \textit{DOMTreePosition} \uplus \mathbb{A} \\ \tau \in \textit{EventType} & \end{array}$$

While we use the same concrete event types and program points for *EH*s, we abstract concrete event targets to maintain a modest number of event targets. We assume the static analyzer expresses analysis results as summaries. A summary $\hat{s}$ is a map from a pair of a program point and a context to an abstract heap:

$$\hat{s} \in \hat{S} = \mathbb{P} \times \textit{Context} \to \hat{\mathbb{H}} \qquad\qquad c \in \textit{Context}$$

where *Context* is parameterized by an input static analyzer of EHA.

For each dynamically collected state $s = (p, h)$ with an event target o and an event type τ both contained in h, Unit Web App Builder calculates an *EH* unit u as follows:

$$\begin{array}{l} u = \alpha_s(s) = (\alpha_o(o),\ \tau,\ p) \\ \text{where } \alpha_o(o) = \begin{cases} \textit{DOMTreePosition(o)} & \text{if } o \text{ is attached on DOM} \\ o & \text{otherwise} \end{cases} \end{array}$$

where *DOMTreePosition*(o) represents the DOM tree position of o in terms of sequences of child indices from the root node of DOM. Then, it constructs an initial summary for each unit u, $\hat{s}_I^u$, as follows:

$$\hat{s}_I^u(p, c) = \begin{cases} \widehat{h}_u^{\texttt{init}} & \text{if } p \text{ is the global entry point } \wedge\ c = \epsilon \\ \bot_{\mathbb{H}} & \text{otherwise} \end{cases}$$

The initial summary maps all pairs of program points and contexts to the heap bottom $\bot_{\mathbb{H}}$ denoting no information, but it keeps a single map from a pair of the global entry program point and the empty context ϵ to the initial abstract heap $\widehat{h}_u^{\texttt{init}} = \bigsqcup_i \alpha_h(h_i)$ where $s_i \in \textsf{Collected States} \ \wedge\ \alpha_s(s_i) = u \ \wedge\ s_i = (p_i, h_i)$. The initial abstract heap for a unit u is a join of all abstraction results of the heaps in the collected states that are mapped to the same u. The heap abstraction α_h and the abstract heap join $\bigsqcup$ are parameterized by the input static analyzer.

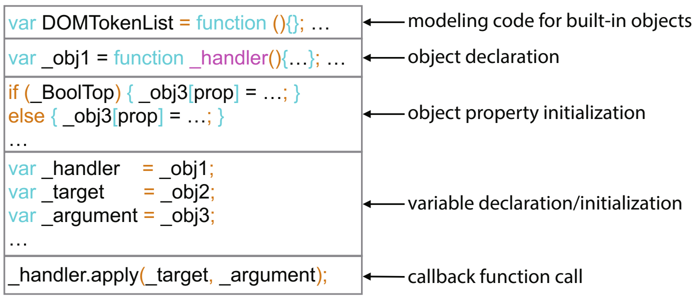
Static Analysis Phase. Now, the static analysis phase analyzes each sliced unit web app one by one, and detects any bugs in it. Let us call the static analyzer that EHA takes as its input SA. Without loss of generality, let us assume that SA performs a whole-program analysis to compute the analysis result $\hat{s}_{\texttt{final}}$ with the initial summary $\hat{s}_I$ by computing the least fixpoint of a semantics transfer function $\hat{F}$: $\hat{s}_{\texttt{final}} = \ \texttt{leastFix}\ \lambda\hat{s}.(\hat{s}_I \sqcup_{\widehat{S}} \hat{F}(\hat{s}))$ and then reports alarms for possible bugs in it. We call an instance of EHA that takes SA as its input static analyzer $\textsf{EHA}_{\textsf{SA}}$. Then, for each *EH* unit u, $\textsf{EHA}_{\textsf{SA}}$ performs an *EH*-based analysis to compute its analysis result $\hat{s}_{\texttt{final}}^u$ with the initial summary $\hat{s}_I^u$ constructed during the unit building phase by computing the least fixpoint of the same semantics transfer function $\hat{F}$: $\hat{s}_{\texttt{final}}^u = \texttt{leastFix}\ \lambda\hat{s}.(\hat{s}_I^u \sqcup_{\widehat{S}} \hat{F}(\hat{s}))$. It also reports alarms for possible bugs in each unit u.

Alarm Aggregation Phase. The final phase combines all bug reports from sliced unit web apps and constructs a final bug report. Because source locations of bugs in a bug report from a unit web app are different from those in an original input web app, Alarm Aggregator resolves such differences. Since a single source location in the original web app may appear multiple times in differently sliced unit web apps, Alarm Aggregator also merges bug reports for the same source locations.

4 Implementation

This section describes how we implemented concrete data representation and each module in dark boxes in Fig. 3 in our prototype implementation.

Instrumentor. The main idea of instrumentor is similar to that of Jalangi [20], a JavaScript dynamic analysis framework, and we implemented the rules (partially) shown in Fig. 4. An instrumented web app collects states during execution by stringifying them and writing them on files. Dynamically collected information may be ordinary JavaScript values or built-in objects of JavaScript engines or browsers, which are often implemented in non-JavaScript, native languages. Because such built-in values are inaccessible from JavaScript code, we omit their values in the collected states. On the contrary, ordinary JavaScript values are stringified in JSON format. A primitive value is stringified by `JSON.stringify` and stored in `ValueMap`. An object value is stored in two places—its pointer in `Storage` and its pointer identifier in `ValueMap`—and its property values are also recursively stringified and stored in `StorageMap`. The stringified document, `ValueMap`, and `StorageMap` are written in files at the end of execution, and Unit Web App Builder converts them to states in the unit building phase.



```
var DOMTokenList = function (){}; …
var _obj1 = function _handler(){…}; …
if (_BoolTop) { _obj3[prop] = …; }
else { _obj3[prop] = …; }
…
var _handler  = _obj1;
var _target   = _obj2;
var _argument = _obj3;
…
_handler.apply(_target, _argument);
```

modeling code for built-in objects
object declaration
object property initialization
variable declaration/initialization
callback function call

Fig. 5. Contents in a JavaScript file of a unit web app

Unit Web App Builder. In our prototype implementation, the unit web app builder parses the collected states as in JSON format and constructs a unit web app as multiple HTML files and one JavaScript file. A single JavaScript file contains all the information to build an initial abstract heap as Fig. 5. It contains modeling code for built-in objects on the top, declares objects recorded in `StorageMap` and initializes their properties, and then declares and initializes non-local variables, which are all the information needed to build an initial abstract heap. At the bottom, the handler function is being called.

Starting from the above 3 variables—`_handler`, `_target`, and `_arguments`—we can fill in contents of a unit web app using the collected states. For each variable, we get its value from the collected states and construct a corresponding JavaScript code. When the value of a variable is a primitive value, create a corresponding code fragment as a string literal. For an object value, get the value from `StorageMap` using its pointer id, and repeat the process for its property values. For a function object value, repeat the process for its non-local variables.

Alarm Aggregator. The alarm aggregator maintains a mapping between different source locations and eliminates duplicated alarms. It should map between locations in the original web app and in sliced unit web apps. Our implementation keeps track of corresponding AST nodes in different web apps, and utilizes the information for mapping locations. It identifies duplicated alarms by string comparison of their bug messages and locations after mapping the source locations.

5 Experimental Evaluation

In this section, we evaluate $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$, an instantiation of EHA with manual event generation and SAFE [12], to answer the following research questions:

In the case of providing dynamic events as many as possible,

- RQ1. **Full Coverage**: How many event flows does the *EH*-based analysis cover compared with the whole-program analysis?
- RQ2. **Precision**: How precise is the *EH*-based analysis compared with the whole-program analysis?
- RQ3. **Scalability**: What is the execution time of each phase in the analyses?
- RQ4. **Partial Coverage**: How many event flows does the *EH*-based analysis cover for timeout analyses?

5.1 Experimental Setup

We studied 8 open-source game web apps [8], which were used in the evaluation of SAFE. They have various buttons and show event-dependent behaviors. The first two columns of Table 1 show the names and lines of code of the apps, respectively. The first four apps do not use any JavaScript libraries, and the remaining apps use the jQuery library version 2.0.3. They are all cross-platform apps that can run on Chrome, Chrome-extension, and Tizen environments.

To perform experiments, we instantiated EHA with two inputs. As an Event Generator input, we chose manual event generation by one undergraduate researcher who was ignorant of EHA. He was instructed to explore behaviors of web apps as much as possible, and he could check the number of functions being called during execution as a guidance. In order to make execution environments simple enough to reproduce multiple times, we collected dynamic states from a browser without any cached data. As a Static Analyzer input, we use SAFE

Table 1. Analysis coverage of SAFE and $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$.

App	LoC	#Analyzed Handler Ftn			#Analyzed Ftn			Total
(Id) App name		Both	SAFE only	$\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ only	Both	SAFE only	$\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ only	
(01) HangOnMan	1326	20	0	11	67	3	19	89
(02) MakeAMonster	1405	22	0	5	63	5	7	75
(03) Mancala	1546	28	0	4	67	4	5	76
(04) Rabbit	1403	34	0	2	76	22	2	100
(05) Bubblewrap	7220	-	-	8	-	-	10	10
(06) CountingBeads	6949	-	-	9	-	-	11	11
(07) MemoryGameForOlderKids	6955	-	-	7	-	-	9	9
(08) WordsSwarm	7557	-	-	9	-	-	48	48
Total	34363	104	0	55	273	34	111	418

because it can analyze the most JavaScript web apps among existing analyzers via the state-of-the-art DOM tree abstraction [14,15] and it supports a bug detector [16]. We ran the apps with Chrome on a 2.9 GHz quad-core Intel Core i7 with 16 GB memory in the execution phase. The other phases are conducted on Ubuntu 16.04.1 with intel Core i7 and 32 GB memory.

5.2 Answers to RQs

Answer to RQ1. For the analysis coverage, we measured the numbers of analyzed functions and true positives by SAFE and $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$. Because SAFE could not analyze 4 apps that use jQuery within the timeout of 72 h, we considered only the other apps for SAFE.

Table 1 summarizes the result of analyzed functions. The 3rd to the 5th columns show the numbers of registered event handler functions analyzed by both, SAFE only, and $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ only, respectively. Similarly, the 6th to the 8th columns show the numbers of functions analyzed by both, SAFE only, and $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ only, respectively. When we compare only the registered event handler functions among all the analyzed functions, $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ outperforms SAFE. Even though SAFE was designed to be sound, it missed some behaviors. Our investigation showed that the causes of the unsoundness were due to incomplete DOM modeling. For the numbers of analyzed functions, the analyses covered more than 75% of the functions in common. $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ analyzed more functions for the first 3 subjects than SAFE due to missing event registrations caused by incomplete DOM modeling in SAFE. On the other hand, SAFE analyzed more functions for the 4th subject because $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ missed flows during the execution phase. We studied the analysis result of the 4th subject in more detail, and found flows that resume previously suspended execution by using cached data in a `localStorage` object. $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ could not analyze the flows because it does not contain cached data, while SAFE could use a sound modeling of `localStorage`. Lastly, $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ did not miss any true positives that SAFE detected, and $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ could detect four more true positives in common functions as shown in Table 2, which implies that $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ analyzed execution flows in those functions that SAFE missed. We explain Table 2 in more detail in the next answer.

Answer to RQ2. To compare the analysis precision, we measured the numbers of false positives (FPs) in alarm reports by SAFE and $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$. Note that true positives (TPs) may not be considered as "bugs" by app developers. For example, while SAFE reports a warning when the `undefined` value is implicitly converted to a number because it is a well-known error-prone pattern, it may be an intentional behavior of a developer. Thus, TPs denote they are reproducible in concrete executions while FPs denote it is impossible to reproduce them in feasible executions. Similarly for RQ1, we compare the analysis precision for four apps that do not use jQuery.

Tables 2 and 3 categorize alarms in three categories: alarms reported by both SAFE and $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$, alarms in functions commonly analyzed by both, and alarms in functions that are analyzed by only one. Table 2 shows numbers of TPs and

Table 2. Alarms reported by SAFE and $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$.

App Id	Common alarms		Different alarms							
			Common functions				Different functions			
			SAFE		$\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$		SAFE		$\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$	
	#TP	#FP	#TP	#FP	#TP	#FP	#TP	#FP	#TP	#FP
01	1	3	0	10	3	2	0	0	0	2
02	1	2	0	0	1	8	0	5	0	1
03	1	3	0	30	0	6	0	0	0	2
04	3	7	0	1	0	0	0	0	0	0
05	-	-	-	-	-	-	-	-	0	1
06	-	-	-	-	-	-	-	-	0	3
07	-	-	-	-	-	-	-	-	0	1
08	-	-	-	-	-	-	-	-	0	1
Total	6	15	0	41	4	16	0	5	0	11

Table 3. False alarms categorized by causes

Cause	Common alarms	Different alarms			
		Common functions		Different functions	
		SAFE	$\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$	SAFE	$\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$
Infeasible event flow	-	40	-	0	-
ECMAScript 5	1	0	0	0	0
Object join	0	0	3	0	0
Handler unit abstraction	-	-	3	-	0
Omitted property	-	-	0	-	2
Absence of DOM model	14	1	10	5	9
Total	15	41	16	5	11

FPs for each app, and Table 3 further categorizes alarms in terms of their causes. Out of 21 common alarms, 6 are TPs and 15 are FPs. Among 15 common FPs, 14 are due to absence of DOM modeling and 1 is due to the unsupported getter and setter semantics. For the functions commonly analyzed by both, they may report different alarms because they are based on different abstract heaps. We observed that 40 FPs from SAFE are due to the over-approximated event system modeling. Especially, the causes of FPs in the 01 and 03 apps are because top-level variables are initialized when non-load event handler functions are called, which implies that the event modeling of Fig. 1(b) would have a similar imprecision problem. On the contrary, $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ reported only 16 FPs mostly (10 FPs) due to absence of DOM modeling. The remaining three FPs from object joins and three FPs by handler unit abstraction are due to inherent problems of static analysis that merges multiple values losing precision. Finally, for the functions analyzed by only one analyzer, all the reported alarms are FPs due to absence of DOM modeling and omitted properties in the $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ implementation. In short, $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ could partially analyze more subjects than SAFE, and it improved the analysis precision by finding four TPs and less FPs for commonly analyzed functions. Especially, its *handler unit abstraction* produced three FPs which are considerably fewer than 40 FPs from over-approximated event modeling in SAFE without missing any TPs.

Answer to RQ3. To compare the analysis scalability, we measured the execution time of each phase for the both analyzers as summarized in Table 4.

Table 4. Execution time (seconds) of each phase for SAFE and $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$

Id	SAFE			$\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$							
	Total	Top-Level	Event Loop	Execution			Unit build	Static analysis			
				Total	#Call	Ave.		Total	#EH	#TO	Ave.
01	375.7	8.9	366.8	465.41	682	0.68	10.0	33038.4	130	9	96.6
02	282.0	8.2	273.8	252.86	135	1.87	6.0	6379.7	33	0	70.4
03	850.2	15.5	834.7	82.70	168	0.49	2.0	7894.1	43	3	68.8
04	1276.6	325.3	951.3	302.36	589	0.51	2.1	16223.9	95	7	54.2
05	✗	137.3	✗	1713.61	151	11.35	287.2	66238.5	63	55	10.4
06	✗	86.9	✗	383.08	85	4.51	221.5	17257.1	27	9	146.5
07	✗	119.3	✗	2836.05	242	11.72	348.2	104583.5	94	87	7.7
08	✗	82.4	✗	1074.73	146	7.36	1158.5	39506.3	41	32	33.5
Ave.	696.1	98.0	606.6	888.85	275	3.24	254.4	3076.5	66	25	76.0

For SAFE, we measured the time took for analyses of the entire code, top-level code, and event loops: $\mathsf{Total} = \mathsf{Top\text{-}Level} + \mathsf{Event\ Loop}$. For four subjects that do not use any JavaScript libraries, the total analysis took at most 1276.6 s among which 951.3 s took for analyzing event loops. While SAFE finished analyzing the top-level code of the other subjects that use jQuery in 137.3 s at the maximum, it could not finish analyzing their entire code within the time of 72 h (259,200 s).

For $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$, because the maximum execution time of the instrumentation phase and the alarm aggregation phase are 10.3 s and 4.9 s, respectively, much smaller than the other phases, the table shows only the other phases. For the execution phase, we present the overhead to collect states:

$$\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}} \text{ (Execution Phase)}: \mathsf{Total} = \mathsf{\#Call} \times \mathsf{Ave.}$$

The 6th column presents the numbers of event handler function calls that Event Generator executed; each event handler function pauses for 3.24 s on average. In order to understand the performance overhead due to the instrumentation, we measured its slowdown effect by replacing all the instrumented helper functions with a function with the empty body. With the Sunspider benchmark, Jalangi showed x30 slowdown and $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ showed x178 slowdown on average. We observed that collecting non-local variables for each function incurs much performance overhead, and more function calls make more overhead.

The unit building phase takes time to generate unit web app code. Our investigation showed that the time heavily depends on the size of collected data. For the static analysis phase, we measured the analysis time of unit web apps except timeout (TO):

$$\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}\ \text{(Static Analysis Phase)}: \mathsf{Total} = (\#\mathsf{EH} - \#\mathsf{TO}) \times \mathsf{Ave.} + 1200 \times \#\mathsf{TO}$$

We analyzed each unit web app with the timeout of 1200 s. While the 02 app has no timeout, the 07 app has 87 timeouts out of 94 unit web apps. On average, analysis of 38% (25/66) of the unit web apps was timeout. Note that even for the first four apps that SAFE finished analysis, $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ had some timeouts. We conjecture that SAFE finished analysis quickly since it missed some flows because of unsupported DOM modeling. By contrast, because $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ analyzes more flows using dynamically collected data, it had several timeouts.

Answer to RQ4. To see how many event flows $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ covers with a limited time budget, let us consider four apps that SAFE did not finish in 72 h from Tables 1 and 4. $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ finished 19% (42/225) of the units within the timeout of 1200 s as shown in Table 4, and the average analysis time excluding timeouts was 76.0 s. Because it implies that web apps have event flows that can be analyzed in about 76 s, it may be meaningful to analyze such simple event flows quickly first to find bugs in them. Starting with 42 units, $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ covered 78 functions as shown in Table 1. While SAFE could not provide any bug reports for four apps using jQuery, $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ reported 6 alarms from the analzyed functions.

6 Related Work

Researchers have studied event dependencies to analyze event flows more precisely. Madsen *et al.* [13] proposed event-based call graphs, which extend traditional call graphs with behaviors of event handlers such as registration and trigger of events. While they do not consider analysis of DOM state changes and event capturing/bubbling behaviors, EHA addresses them by utilizing dynamically collected states. Sung *et al.* [21] introduced DOM event dependency and exploited it to test JavaScript web apps. Their tool improved the efficiency of event testing but it has not yet been applied for static analysis of event loops.

Taking advantage of both static analysis and dynamic analysis is not a new idea [5]. For JavaScript analysis, researches tried to analyze dynamic features of JavaScript [7] and DOM values of web apps [23,24] precisely. Alimadadi *et al.* [1] proposed a DOM-sensitive change impact analysis for JavaScript web apps. JavaScript Blended Analysis Framework (JSBAF) [26] collects dynamic traces of a given app, specializes dynamic features of JavaScript like `eval` calls and reflective property accesses utilizing the collected traces. JSBAF analyzes each trace separately and combines the results, but EHA abstracts the collected states on each *EH* first and then analyzes the units to get generalized contexts. Finally, Ko *et al.* [11] proposed a tunable static analysis framework that utilizes a light-weight pre-analysis. Similarly, our work builds an approximation of selected executions by constructing an initial abstract heap utilizing dynamic information, which enables to analyze complex event flows although partially.

7 Conclusion and Future Work

Because existing JavaScript static analyzers conservatively approximate event-driven flows, even state-of-the-art analyzers often fail to analyze event flows in web apps within a timeout of several hours. We present EHA, a bug detection framework that performs a novel *EH*-based static analysis using dynamically collected state information. As a general framework, EHA is parameterized by a way to generate event sequences and a JavaScript static analyzer. We present $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$, an instantiation of EHA with manual event generation and the SAFE JavaScript static analyzer. Our experimental evaluation shows that the *EH*-based analysis ($\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$) reduced false positives reported by the whole-program analysis (SAFE) due to its over-approximation of the event system modeling. Moreover, $\mathsf{EHA}^{\mathsf{man}}_{\mathsf{SAFE}}$ finished analyzing partial execution flows of the web apps that SAFE failed to analyze within the timeout of 72 h. We plan to inspect the soundness issues due to the lack of DOM modeling in whole-program analyzers with systematic ways via dynamic analyses [3,6,25], and to use an automated testing tool as a dynamic event generator instead of the manual generation.

Acknowledgment. The research leading to these results has received funding from National Research Foundation of Korea (NRF) (Grants NRF-2017R1A2B3012020 and 2017M3C4A7068177).

References

1. Alimadadi, S., Mesbah, A., Pattabiraman, K.: Hybrid DOM-sensitive change impact analysis for JavaScript. In: ECOOP 2015 (2015)
2. Andreasen, E., Møller, A.: Determinacy in static analysis for jQuery. In: OOPSLA 2014 (2014)
3. Andreasen, E.S., Møller, A., Nielsen, B.B.: Systematic approaches for increasing soundness and precision of static analyzers. In: SOAP 2017 (2017)
4. Calcagno, C., et al.: Moving fast with software verification. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 3–11. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_1
5. Ernst, M.D.: Static and dynamic analysis: synergy and duality. In: PASTE 2004 (2004)
6. Grech, N., Fourtounis, G., Francalanza, A., Smaragdakis, Y.: Heaps don't lie: countering unsoundness with heap snapshots. In: OOPSLA 2017 (2017)
7. Guarnieri, S., Livshits, B.: GATEKEEPER: mostly static enforcement of security and reliability policies for JavasSript code. In: SSYM 2009 (2009)
8. Intel: HTML5 web apps (2017). https://01.org/html5webapps/webapps
9. Jensen, S.H., Madsen, M., Møller, A.: Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In: ESEC/FSE 2011 (2011)
10. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for JavaScript. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 238–255. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03237-0_17
11. Ko, Y., Lee, H., Dolby, J., Ryu, S.: Practically tunable static analysis framework for large-scale JavaScript applications. In: ASE 2015 (2015)

12. Lee, H., Won, S., Jin, J., Cho, J., Ryu, S.: SAFE: formal specification and implementation of a scalable analysis framework for ECMAScript. In: FOOL 2012 (2012)
13. Madsen, M., Tip, F., Lhoták, O.: Static analysis of event-driven Node.js JavaScript applications. In: OOPSLA 2015 (2015)
14. Park, C., Ryu, S.: Scalable and precise static analysis of JavaScript applications via loop-sensitivity. In: ECOOP 2015 (2015)
15. Park, C., Won, S., Jin, J., Ryu, S.: Static analysis of JavaScript web applications in the wild via practical DOM modeling. In: ASE 2015 (2015)
16. Park, J., Lim, I., Ryu, S.: Battles with false positives in static analysis of JavaScript web applications in the wild. In: ICSE-SEIP 2016 (2016)
17. Richards, G., Lebresne, S., Burg, B., Vitek, J.: An analysis of the dynamic behavior of JavaScript programs. In: PLDI 2010 (2010)
18. Sadowski, C., Van Gogh, J., Jaspan, C., Söderberg, E., Winter, C.: Tricorder: building a program analysis ecosystem. In: ICSE 2015 (2015)
19. Schäfer, M., Sridharan, M., Dolby, J., Tip, F.: Dynamic determinacy analysis. In: PLDI 2013 (2013)
20. Sen, K., Kalasapur, S., Brutch, T., Gibbs, S.: Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In: ESEC/FSE 2013 (2013)
21. Sung, C., Kusano, M., Sinha, N., Wang, C.: Static DOM event dependency analysis for testing web applications. In: FSE 2016 (2016)
22. TIOBE: TIOBE Index for September 2017. http://www.tiobe.com/tiobe-index
23. Tripp, O., Ferrara, P., Pistoia, M.: Hybrid security analysis of web JavaScript code via dynamic partial evaluation. In: ISSTA 2014 (2014)
24. Tripp, O., Weisman, O.: Hybrid analysis for JavaScript security assessment. In: ESEC/FSE 2011 (2011)
25. Wang, Y., Zhang, H., Rountev, A.: On the unsoundness of static analysis for android GUIs. In: SOAP 2016 (2016)
26. Wei, S., Ryder, B.G.: Practical blended taint analysis for JavaScript. In: ISSTA 2013 (2013)