JSTrace: Fast Reproducing Web Application Errors

Jie Wang[†][‡], Wensheng Dou[†], Chushu Gao[†][*], Jun Wei[†][‡]
[†] State Key Lab of Computer Science, Institute of Software, Chinese Academy of Sciences, China
[‡] University of Chinese Academy of Sciences, China
{wangjie12, wsdou, gaochushu, wj}@otcaix.iscas.ac.cn

Abstract—**JavaScript has become the most popular language for client-side web applications. Due to JavaScript's highly-dynamic and event-driven features, it is challenging to diagnose web application errors. Record-replay techniques are used to reproduce errors in web applications. After a long run, these techniques will record a long event trace that triggers an error. Although the error-related events are few, they are interleaved with other massive error-irrelevant events. It is time-consuming to diagnose errors with long event traces.**

In this article, we present JSTrace, which effectively removes error-irrelevant events from the long event trace, and further facilitates error diagnosis. Based on fine-grained dependences of JavaScript and DOM instructions, we develop a novel dynamic slicing technique that can remove events irrelevant to the error. We further present rules to remove irrelevant events, which cannot be removed by dynamic slicing. In this process, many events and related instructions are removed without losing the error reproducing accuracy. Our evaluation on 13 real-world web application errors shows that the reduced event traces can faithfully reproduce errors with an average reduction rate of 97%. We further performed case studies on 4 real-world errors, and the result shows that JSTrace is useful to diagnose web application errors.

Keywords—record-replay; dynamic slicing; event trace reduction; dependence analysis

1. Introduction

JavaScript has been widely used in web applications. JavaScript-based web applications, e.g., Gmail [1], Google Doc [2] and FaceBook [3] can provide rich and highly interactive user experience. However, due to JavaScript's event-driven features and complicated DOM manipulations, a variety of bugs could be easily introduced into web applications [4][5][6]. These bugs can cause serious errors, such as abnormal functionality, missing UI elements, and so on [4][5].

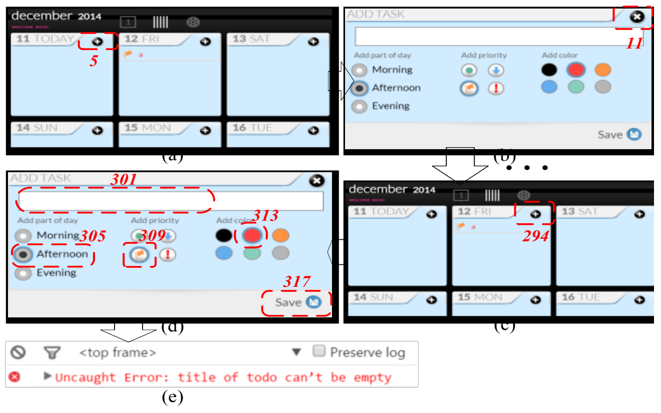
JavaScript-based web applications are event-driven. Errors in web applications are usually triggered by a specific sequence of events (e.g., user interactions) [7]. In order to facilitate diagnosis of web application errors, various record-replay techniques are used to faithfully reproduce them [8][9][10]. There are two kinds of record-replay techniques. The event-based record-replay techniques [8][9] record user interactions (events) and use them to drive the execution during replay. The memory-based record-replay techniques [10] record every value loaded from memory during record and replace memory loads with these values during replay. These techniques are useful for error diagnosis when the event traces are short.

However, JavaScript-based web applications usually keep running for a long time (e.g., writing a document in Google Doc may take an hour). Thus, amounts of events are generated, and the current record-replay techniques [8][9][10] will generate a very long event trace. For example, Mugshot [8] can generate 75-795KB uncompressed event trace (nearly 3,000 events) per minute. In order to diagnose an error, developers have to replay and inspect all the events in the event trace. This would be time-consuming and exhausting. Delta debugging can be adopted to reduce event traces. However, without knowing the relationship among events in the event trace, delta debugging blindly generates subtraces and validates them. Due to large search space, delta debugging cannot scale to long event traces [11].

In this article, we focus on how to speed up the web application error reproducing by removing error-irrelevant events in the event trace. Our key observation is that most of the events in an event trace are irrelevant to an error. After removing these irrelevant events, the error can still be *faithfully* reproduced. Our basic idea is to build a *Dynamic Dependence Graph* (*DDG*) to trace JavaScript instruction dependences by analyzing the use-def relationship of these instructions, and an *Event Dependence Graph* (*EDG*) to trace event dependences based on *DDG*. Then, we backward traverse *EDG* from the event *e* where the error occurs, and only keep the *key* events that are depended by event *e*. The remaining events are considered irrelevant to the error, and removed from the event trace. Finally, a short event trace (error related events) with related dependences is provided to developers, so that they can quickly reproduce an error and save time for error diagnosis.

The key challenge in the above process is how to *precisely capture dynamic dependence of JavaScript instructions* and *determine whether an event is irrelevant to the error*. Specially, we need to address four challenges. First, JavaScript is a dynamic and weak-typing language, which magnifies the difficulty to perform dependence analysis. Second, the DOM APIs (a special kind of JavaScript instructions) used for manipulating the DOM tree (tree-structure representation of a HTML page) are implemented by native code [12]. Treating these DOM APIs as general JavaScript instructions without knowing their semantics can miss key dependences. For example, we cannot obtain the dependence between *div.getAttribute*("*key*") and *div.setAttribute*("*key*", "*value*") without considering the semantics of *getAttribute* and *setAttribute.* Third, simply treating the whole DOM tree as a single global JavaScript object can introduce false dependences. Fourth, although an event e_i depends on another event e_j in *EDG*, event e_j can be removed even if event e_i is selected. We use a simple 3-event example to illustrate this: e_1: {$a = 1$}, e_2: {a++; b=2}, e_3 {if ($b > 0$) throw new Error()}. In this example, e_3

[*] Corresponding author



```
Uncaught Error: title of todo can't be empty
```

Fig. 1. TodoList application. The buggy event trace is shown in Fig. 2.

ID	Event	Event handler	Description
1	**Load page**		Load page
…			Click *add* button and directly close it (Fig. 1a and Fig. 1b).
5	Click *add* button on Dec 11	onAdd (Line 20)	
…			
11	Click *close* button		
…			Change settings (Omitted events between Fig. 1b and Fig. 1c)
73	Click *setting* button		
…			
294	**Click *add* button on Dec 13**	onAdd (Line 20)	Click *add* task button and fill in the form in the popup dialog. The bug is triggered when click the *save* button (Fig. 1c and Fig. 1d).
…			
301	**Input task name**		
…			
305	Select task time		
…			
309	Select task priority		
…			
313	Select task color		
…			
317	**Click *save* button**	onSave (Line 28)	

**The 4 events in bold font are minimal events to reproduce the error.*

Fig. 2. An event trace that triggers the error in Fig. 1e. The first column

depends on e_2 (e_3 uses variable b that is modified by e_2) and e_2 depends on e_1 (e_2 uses variable a that is modified by e_1). However, e_1 can still be removed, because the condition (b>0) in e_3 is true even if event e_1 is removed.

To address the first three challenges, we abstract the JavaScript instructions and DOM instructions (a special kind of JavaScript instructions) to precisely capture dependences among them and produce a dynamic dependence graph. First, we model JavaScript instructions and provide the dependence propagation rules among them. Second, we model all the DOM APIs and map them to a fine-grained DOM dependence model. Then, we perform dependence analysis on the fine-grained DOM dependence model. For the last challenge, we introduce a rule-based approach to further remove those irrelevant events that cannot be removed by only analyzing event dependences.

Our approach is implemented into a tool *JSTrace*, which can dramatically remove irrelevant events while keeping the reduced trace reproducible. JSTrace is implemented in pure JavaScript, and enables easy integration into the client-side web applications and executed in any browser. We have evaluated it on 13 real-world errors in 10 popular web applications from different domains. The evaluation shows that we can efficiently remove 97% irrelevant events, and still faithfully reproduce all these errors.

In summary, the contributions of this article are as follows:

- We propose a novel approach to analyze dependences by abstracting JavaScript and fine-grained DOM instructions, and design dependence propagation rules among these instructions.
- We propose an effective event slicing approach to filter out irrelevant events based on event dependence graph.
- We propose a rule-based approach to further remove irrelevant events that are still depended by others in the event dependence graph (i.e., they cannot be removed by event dependence analysis).
- We have implemented our approach in the tool JSTrace. The evaluation on 13 real-world web applications errors shows that JSTrace can remove 97% of irrelevant events, and reproduce the errors faithfully.
- As the ultimate goal of JSTrace, we applied our dependence analysis on several real-world web application errors. The result shows that our dependence analysis is helpful in diagnosing these errors.

An earlier version of this work appeared at ISSRE 2015 [13]. In this article, we extend the earlier version in three aspects. (1) We further study irrelevant events that cannot be removed by our ISSRE version, and come up a rule-based approach to further remove irrelevant events (28% of all remaining irrelevant events by our ISSRE version can be removed). (2) We perform our experiments on more subjects (from 7 to 10 applications, and 10 to 13 errors), and further validated JSTrace's effectiveness. (3) We perform four case studies about how JSTrace is used to help diagnose JavaScript-based web application errors, and validated the usefulness of JSTrace.

The remaining of this article is organized as follows. Section 2 presents our motivation. Section 3 introduces our approach. Section 4 describes JSTrace implementation. In Section 5 we evaluate our tool on real-world errors in terms of reproducibility, efficiency, performance and applicability. Section 6 describes the study on how JSTrace is used to help diagnose web application errors. Section 7 discusses threats to our evaluation. Section 8 describes related work and Section 9 concludes this article.

2. Motivation

In this section, we illustrate our motivation using a real-world example, and explain how to remove irrelevant events for this example.

2.1. Example

Fig. 1 shows the TodoList [14] web application that manages a calendar for users. When a user clicks the *add* button on a day view (Fig. 1c), a dialog pops up to create a to-do task (Fig. 1d). After the user fills up all necessary information, he/she clicks the *save* button. The application checks the *title* of the to-do task, and an error will occur if the *title* can be trimmed to an empty string (Fig. 1e). The simplified source code for TodoList is shown in Listing 1. The event handler *onAdd* (Line 20) is invoked when clicking the *add* button in Fig. 1c, and the event

```
1  document.onload(function(){
2    new TodoList().init();
3  });

4  function TodoList(){   // Initialize TODO application
5    this.container = {
6      root: document.getElementById('#todolist'),
7      dayView: ...
8      popupView: ...
9      settingView:...
10   }
11   ...
12 }

13 TodoList.prototype.init = function(){ // Initialize day view
14   //Add onAdd handler for every day in the day view
15   day.getElementByClassName('add')[0].addEventListner('click',
           onAdd);
16   this.container.dayView.appendChild(day);
17   ...
18   defaultView.show();
19 }

20 function onAdd(){   // Event handler for add button in Fig. 1a
21   ...
22   popup = document.createElement('div');
23   popup.innerHTML = '<div id="title" style="tt"></div>...<div
           id="save"></div>';
24   this.container.popupView.appendChild(popup);
25   ...
26   popup.getElementById('save').addEventListener('click',onSave);
27 }

28 function onSave(){ // Event handler for save button in Fig. 1d
29   var todo = new TODO();  // Create a TODO object
30   todo.title = this.popup.getElementbyId('title').value;
31   tasklist = storage.getTaskList();
32   if(check(todo)){
33     tasklist.push(todo);     // storage is an object for persistent
34   }
35 }
36 function check(todo){
37   if(util.trimToEmpty(todo.title).length==0){
38     throw new Error("title of todo can't be empty");
39   }
40   return true;
41 }
```

Listing 1. Simplified source code for TodoList.

handler *onSave* (Line 28) is invoked when clicking the *save* button in Fig. 1d.

Fig. 2 lists a real event trace that triggers the above error. In this trace, the user clicks add button on the view of Dec 11(event 5, Fig. 1a), and then clicks close button to cancel this operation (event 11, Fig. 1b). He/she then performs a series of actions such as changing calendar settings (events 73~293). Afterwards, he/she clicks add button on the view of Dec 13 (event 294, Fig. 1c), and types an empty string with a blank space in the title field (event 301), fills in other fields in the form, and clicks save button (event 317), which finally triggers the error (Fig. 1e). Replaying the entire event trace can reproduce the error successfully. However, it is not efficient for debugging and error diagnosis. We observe that most of the events are irrelevant to the error, such as, clicking the add button and then clicking close button to close the popup window (events 5~11, Fig. 1a and Fig. 1b), changing settings (events 73~293) and filling in some form

```
div.className='left';
class=div.getAttribute('class');    //or class=div.className;
```

Listing 2. Example of inconsistent DOM interfaces.

fields (events 302~316). Removing these events will not affect the occurrence of the error. The key events {1,294,301,317} can faithfully reproduce this error.

2.2. Challenges

In order to remove the irrelevant events, we need to identify the precise dependences between events. In doing so, there are several challenges we should overcome.

1) The DOM APIs are defined by the W3C and implemented as native code in modern browsers. We cannot obtain the dependences among these DOM APIs and DOM elements. For example, the DOM API *getElementById* (Line 30) suggests the dependence between the DOM element *title* with JavaScript object *todo.title*. Without understanding the semantics of *getElementById*, we will miss this key dependence. Even worse, JavaScript has some inconsistent DOM APIs. Listing 2 shows a classical example. In Listing 2, after changing the *className* for the element *div* by setting the field *className*, we can access the *className* field by the operation *div.className*. But, we can also access the *className* field by calling the native function *getAttribute* with the attribute name *class*. These inconsistent DOM APIs makes dependence analysis in JavaScript challenging.

2) DOM is a tree object. One modification on a node may affect its parent node or its subtree. Therefore, only analyzing general JavaScript code is insufficient—the dependence analysis must subtly model how a DOM instruction depends on another to avoid missing dependences. For example in Listing 1, the *value* attribute of DOM element *title* (Line 30) depends on the operation that updates this field. Actually, it also depends on the operation that appends the DOM element *title* to the DOM element *popup* by assigning a HTML code segment to the attribute *innerHTML* of DOM element *popup* (Line 23). The operation (Line 23) ensures that a node with id *title* exists. To precisely capture these dependences, dependence analysis on the DOM model should be field-sensitive. In this way, we assume that modifying the attribute *style* (Line 23) of the DOM element *title* (Line 23) will not affect the reading of its attribute *value*.

3) Due to the dynamic and event-driven features in JavaScript-based web applications, it is hard to build the dependences of events statically. For example, in Listing 1, the variable *todo.tiltle* (Line 37) is defined at line 30 in the event handler function *onSave*. The event handler *onSave* is registered at line 26, which can only be triggered when the *add* button is clicked. In this example, the error occurs only when *onAdd* is called before *onSave*.

2.3. JSTrace Overview

In this article, we abstract JavaScript instructions and DOM instructions to precisely capture dependences among them. We further build the dependence between JavaScript instructions and DOM elements. Based on dynamic dependence analysis, we build an event dependence graph that describes the dependent relationship of events. Our slicing algorithm operates on the event dependence graph.

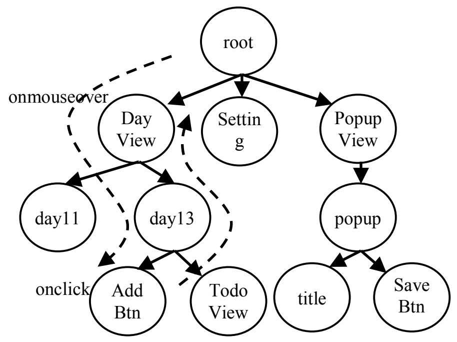
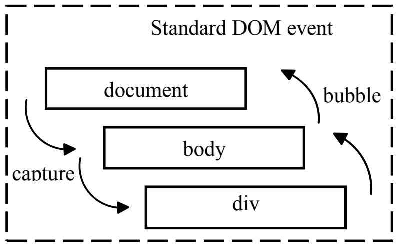


Fig. 3. Partial DOM tree of TodoList. *onmouseover* and *onclick* are event handlers that are bound to DOM element *DayView* and *AddBtn*, respectively.



1) div.addEventListener('click', f)
2) div.onclick=function(){…}
3) <div onclick='…'></div>

Fig. 4. Standard event flow model.

We have two key insights to perform the event trace reduction. 1) If a recorded event never triggers any listeners registered by the user, we can safely remove it (Section 3.1). We present this idea using Fig. 3. Fig. 3 shows the partial DOM tree for the TodoList example in Fig. 1. Dispatching the *click* event on the DOM element *addBtn* will trigger the event handler *onAdd* in Listing 1. However, dispatching a *click* event on DOM element *dayView* will not trigger any handler and this event can be safely removed. 2) If an event *e* does not affect the variables that will be used by the erroneous event's handler directly or indirectly, the event *e* can be removed. We trace backward from the error behavior and identify the operations that affect the occurrence of the error. As shown in Listing 1, the error behavior is an exception thrown at line 38, and depends on the value of variable *todo.title*. The function *onSave* (Line 28) is triggered by event 317 with the value that is set by event 301. Therefore, event 317 depends on event 301. Since the event handler *onSave* is registered in function *onAdd* (Line 26), which is triggered by event 294. Thus, event 317 depends on event 294 as well. Similarly, event 294 depends on the event 1 because event 1 defines the variable *todolist.container* and the DOM element variable *day* (Line 15), and registers event handler *onAdd* (Line 15). Finally, we obtain the event dependences {317⟶[301,294], 294⟶1}. Therefore, we get the key events {1,294,301,317} and all the other irrelevant events can be removed. 3) Some events can still be removed even if they are depended by the erroneous event. We come up a rule-based approach to remove such irrelevant events. Suppose that the user firstly removes a task (this would update the variable *tasklist*), and then triggers the above error (this would read the variable *tasklist* at line 31) when creating a new task. Thus, this removing-task event is depended by the error event. However, we consider this removing-task event irrelevant, because the modification to the variable *tasklist* is not used in all the subsequent path conditions (e.g., the path condition at line 37) and will not affect the occurrence of the error.

3. Approach

Our overall approach consists of three steps:

Step 1: Unhandled event analysis. If an event never triggers any user-defined event handler, we can safely remove it (Section 3.1).

Step 2: Event dependence analysis. We abstract the JavaScript operations into intermediate instructions and design a dependence propagation model in JavaScript-based web applications (Section 3.2). As mentioned earlier, simply treating DOM manipulations as black box is insufficient, and can miss dependences. Thus, we perform fine-grained JavaScript dependence analysis (Section 3.2.1) and DOM dependence analysis (Section 3.2.2). Further, we use these dependences to build the Event Dependence Graph (EDG).

Step 3: The key event generation. We perform dynamic slicing on EDG (Section 3.3) to obtain the key events related to an error. We perform rule-based weak dependence analysis to further remove irrelevant events (Section 3.4).

3.1. Unhandled Event Analysis

According to the DOM3 event model [15], an event could be propagated from the root DOM element along the tree structure to the target DOM element (capture phase), and then bubbles up to the root DOM element (bubble phase). All the event handlers with the same type as the event at the capture phase or bubble phase will be triggered if the event is not canceled in the middle. Fig. 4 shows this process.

However, even an event is fired by the user, it may not trigger any event handler registered by the user. Thus, the event cannot make any change to the application states. Thus, this event can be safely removed. For example, in Fig. 3, a mouseover event on addBtn can trigger the event handler binding to its ancestor dayView, so the mouseover event cannot be removed. A click event on popup will not trigger any event handler, so this click event should be removed.

The key issue in this step is how to obtain the registered event handlers for each DOM element. To resolve this problem, we treat each registered event handler as a special attribute of the corresponding DOM element. We keep a map for the DOM element to its corresponding event handlers. As shown in Fig. 4, an event handler can be registered in 3 different ways: (1) Event handlers can be registered and unregistered by two native functions *addEventListener* and *removeEventListener*. For this case, we can directly rewrite these two native functions to intercept the event handlers that are registered or unregistered. This could be done by taking advantage of JavaScript's dynamic feature. (2) For the second and third cases, event handlers can be set by a property name that is concatenated by a prefix "on" and the corresponding event type, such as *onclick*. For this case, we intercept such property operations and identify event handlers

Table 1. JavaScript dependence analysis.

op	***JSDep*(*op*)**	**Description**
$v = cons$	$def(v)$=$op.id$; $dp(op)$=$\emptyset$	Constant variables do not depend on others.
$v_1 = v_2$	$def(v_1)$=$op.id$; $dp(op)$=$\{def(v_2)\}$	Assign operation depends on the definition of v_2.
$v = \{op^*\}$	$def(v_1)$=$op.id$; $dp = \emptyset$	Function variables do not depend on others.
$v_1 = v_2.v_3$	$def(v_1)$=$op.id$; $dp(op)$=$\{def(v_2), def(v_3), def(v_2.v_3)\}$	Get property operation depends on object v_2, property name v_3, and the property value $v_2.v_3$.
$v_1.v_2 = v_3$	$def(v_1.v_2)$=$op.id$; $dp(op)$=$\{def(v_1), def(v_2), def(v_3)\}$	Set property operation depends on object v_1, property name v_2, and value v_3.
$v_1 = v_2 \otimes v_3$	$def(v_1)$=$op.id$; $dp(op)$=$\{def(v_2), def(v_3)\}$	Binary operation depends on the two input values v_1, and v_2.
$v_1 = \odot\, v_2$	$def(v_1)$=$op.id$; $dp(op)$=$\{def(v_2)\}$	Unary operation depends on the input value v_2.
$v_l.v_2\ (\{v_{p1},\ldots,v_{pn}\})$	$dps(op)$=$\{def(v_1), def(v_1.v_2), def(v_{p1}), \ldots, def(v_{pn})\}$	Function call operation depends on object v_1, the function object v_2, and the input parameters $v_{p1}, \ldots, v_{pn}$.
v=$v_l.v_2(\{v_{p1},\ldots, v_{pn}\})$	$def(v_1)$=$op.id$; $dp(op)$=$\{def(v_1), def(v_1.v_2), def(v_{p1}), \ldots, def(v_{pn}), def(ret)\}$	Function call operation depends on object v_1, the function object v_2, and the input parameters $v_{p1}, \ldots, v_{pn}$. *ret* represents the return value of function v_2.

JavaScript abstract instructions

$cons$ ::= $num \mid str \mid bool \mid undefined \mid null$	
v ::= *object* variable	
op ::= $v = cons$	// Assign a constant to v
$\mid v_1 = v_2$	// Assign variable v_2 to v_1
$\mid v = \{op^*\}$	// Assign a function object to v
$\mid v_1 = v_2.v_3$	// Get property v_3 of object v_2
$\mid v_1.v_2 = v_3$	// Put property v_2 of object v_1
$\mid v_1 = v_2 \otimes v_3$	// Binary operation, $\otimes \in \{+, -, *, /, \text{etc.}\}$
$\mid v_1 = \odot\, v_2$	// Unary operation, $\odot \in \{!, \text{etc.}\}$
$\mid v_1.v_2\ (\{v_{p1}, \ldots, v_{pn}\})$	// Call object v_1's function v_2 without return
$\mid v = v_1.v_2\ (\{v_{p1}, \ldots, v_{pn}\})$	// Call object v_1's function v_2 with return

Fig. 5. JavaScript abstract instructions.

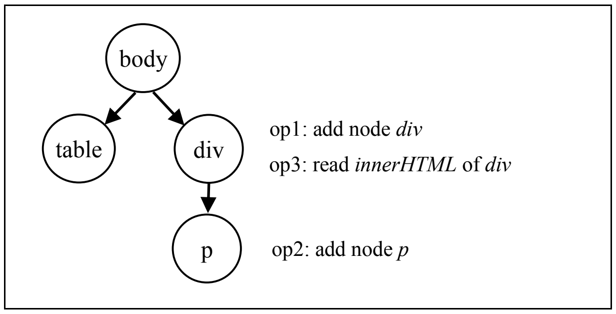


Fig. 6. Dependence example for DOM instructions.

registered. Through the above setting, we can associate all the registered event handlers to their corresponding DOM elements.

3.2. Event Dependence Analysis

If an event e_i depends on another event e_j, one of the following two conditions should be satisfied: (1) The event handler of e_i reads some JavaScript variables defined or written by the event handler of e_j. (2) The event handler of e_i reads some DOM elements appended or modified by the event handler of e_j. Thus, we perform JavaScript dependence analysis (Section 3.2.1) and DOM dependence analysis (Section 3.2.2) in the following.

3.2.1. JavaScript Dependence Analysis

JavaScript dependence analysis builds the dependences among JavaScript instructions. If a JavaScript instruction op_1 uses a variable that is defined or modified by another instruction op_2, we say that op_1 depends on op_2. Note that each DOM instruction is also a JavaScript instruction, here we only consider the dependences among JavaScript instructions without considering DOM instructions' semantics. We will discuss DOM dependence analysis in Section 3.2.2.

We summarize the abstract JavaScript instructions that can affect JavaScript dependence in Fig. 5. A constant value *cons* can be a number *num*, a string *str*, and the special constant *undefined* or *null*. A variable *v* represents an object in JavaScript. The instructions include constant assignment, variable assignment, function definition, property reading and writing, binary operation, unary operation and function call. Each instruction (*op*) is assigned a unique id (*op.id*). Note that a DOM API could be either property access or function call. Therefore, each DOM instruction is also a JavaScript instruction, which is *Get Property*, *Put Property* or *Function Call*.

Table 1 lists all the rules for JavaScript dependence analysis (*JSDep*(*op*)). We use *def*(*v*) to denote the instruction *op* that defines or recently changes variable *v*, and *dp*(*op*) to denote the instruction *op*'s dependences. For example, $def(x) = 5$ denotes that an variable *x* is modified by an instruction with $id = 5$. We then execute the instruction "$y = x$" (with id=10). According to the rules in Table 1, $def(y) = 10$, and $dp(10) = \{5\}$. Thus, we build the dependence between instructions 10 and 5.

For an event handler, it should depend on the instruction that registers the event handler. Losing this dependence may cause mistakenly pruning the registering events and thus fail to replay. We treat each registered event handler as a property of the related DOM element. It is initialized when it is registered and accessed when the event handler is triggered. So, we can resolve this dependence in DOM dependence analysis.

3.2.2. DOM Dependence Analysis

The DOM APIs are designed for manipulating web page's state (the DOM tree). Our JavaScript dependence rules (Section 3.2.1) cannot precisely capture the dependence among these DOM APIs. We summarize the problems as follows:

- ***Inconsistent ways to modify the DOM tree.*** The DOM tree can be modified by an assigning statement or a native function call. As shown in Listing 2, an attribute of a DOM element may be set by a name *className* but accessed by a native function call with the attribute name *class*. Dependences may be missing without understanding the semantics of these DOM APIs.
- Since DOM is a tree structure, a DOM instruction on a node may depend on previous modifications on this node's ancestor nodes or child nodes. For example, in Fig. 6, after

Table 2. DOM dependence analysis.

Id	*dop*	*DOMDep*(*dop*)	Description
1	**DNRead** *ele*	*dp*(*dop.op*)={*SearchDOMDep*(*ele*)}	DOM element read depends on instructions that create/modify *ele*.
2	**DNAdd** *pEle*, *ele*	*dp*(*dop.op*)={*SearchDOMDep*(*pEle*)} *bind*(*dop.op*, *ele*, *DNAdd*)	DOM element add depends on instructions that create/modify *ele*'s parent node *pEle*. It also binds a new operation on *ele*.
3	**DNRm** *ele*	*dp*(*dop.op*)={*SearchDOMDep*(*ele*)} *clearBind*(*ele*)	DOM element remove depends on instructions that create/modify *ele*. It also clears operations on *ele*.
4	**DNReplace** ele_1, ele_2	*dp*(*dop.op*)={*SearchDOMDep*(ele_1)} *clearBind*(ele_1) *bind*(*dop.op*, ele_2, *DNReplace*)	DOM element replace depends on instructions that create/modify the original ele_1. It also clears original operations on *ele*, and binds a new operation on *ele*.
5	**DSubTreeMod** *ele*	*dp*(*dop.op*)={*SearchDOMDep*(*ele*)} *clearSubTreeBind*(*ele*) *bind*(*dop.op*, *ele*, *DSubTreeMod*)	DOM subtree modify depends on instructions that create/modify the original *ele*. It also clears original operations on *ele* and its subtree, and binds a new operation on *ele*
6	**DAttrRead** *ele*, *attr*	*dp*(*dop.op*)={*SearchDOMDep*(*ele*, *attr*)}	DOM attribute read depends on instructions that create/modify *ele* and *attr*.
7	**DAttrWrite** *ele*, *attr*	*dp*(*dop.op*)={*SearchDOMDep*(*ele*, *attr*)} *clearBind*(*ele*, *attr*) *bind*(*dop.op*, *ele*, *attr*, *DAttrWrite*)	DOM attribute write depends on instructions that create/modify the original *ele* and *attr*. It also clears original operations on *ele* and *attr*, and binds a new operation on *ele.attr*.
8	**DAttrRm** *ele*, *attr*	*dp*(*dop.op*)={*SearchDOMDep*(*ele*, *attr*)} *clearBind*(*ele*, *attr*)	DOM attribute remove depends on instructions that create/modify the original *ele* and *attr*. It also clears original operations on *ele* and *attr*.

DOM abstract instructions

ele ::= DOM element
attr ::= DOM element attribute
dop ::=

| **DNRead** *ele*	// Read a node *ele*
| **DNAdd** *pEle*, *ele*	// Add node *ele* to parent node *pEle*
| **DNRm** *ele*	// Remove node *ele* from DOM tree
| **DNReplace** ele_1, ele_2	// Replace node ele_1 with node ele_2
| **DSubTreeMod** *ele*	// Modify the subtree of node *ele*
| **DAttrRead** *ele attr*	// Read attribute *attr* of node *ele*
| **DAttrWrite** *ele attr*	// Write attribute *attr* of node *ele*
| **DAttrRm** *ele attr*	// Remove attribute *attr* of node *ele*

Fig. 7. DOM abastract instructions.

Algorithm 1. Searching algorithm for *SearchDOMDep*

```
Input: dop (the DOM instruction), ele (the DOM element), attr
      (the attribute, if ncecssary)
Output: ids (dependent instruction ids)
//search ancestor nodes to make sure existence of ele, attr
//searching types: DNAdd, DNRm, DNReplace, DSubtreeMod
1. ids = searchAncestors(ele, attr);
2: if attr != NULL   //if accessing ele.attr
     //get the instruction ids that modify ele.attr
3:    ids = ids ∪ getMutations(ele, attr);
4: if needSearchSubtree(dop)
5:    ids = ids ∪ searchSubtree(ele);
6: return ids;
```

adding a node *div* to node *body* in *op*1, a new node *p* is added to node *div* by *op*2. Therefore, *op*2 depends *op*1. *op*3 depends on *op*1 because the operated DOM element *div* must exist, and *op*3 depends on *op*2 because the reading *innerHTML* operation on the node *div* will get the serialization string of its subtree. Thus, we need to scan the DOM tree to validate whether there is dependence between two DOM instructions in the DOM tree.

To resolve these problems, we abstract the DOM instructions and extend the JavaScript dependence analysis to propagate DOM-specific dependences. We summarize the DOM APIs into eight instructions in Fig. 7. DOM instructions can (1) read, add, remove, and replace a DOM element, and (2) modify a subtree of a DOM element, and (3) read, write, and remove an attribute of a DOM element. Each DOM instruction *dop* associates with a JavaScript instruction *op*.

DOM dependence analysis builds the dependences among DOM instructions. If a DOM instruction dop_1 uses a DOM element that is defined or modified by another DOM instruction dop_2, we say that dop_1 depends on dop_2. Table 2 presents the dependence propagation rules of DOM dependence analysis. The first column shows DOM instructions. The second column shows the DOM dependence rules *DOMDep*(*dop*). We treat the DOM tree as a fine-grained variable that contains nodes and attributes. We associate the DOM elements with the instruction *id* that defines or modifies the DOM elements. In Table 2, we use the functions *bind*(*op*, *ele*, *dop*) and *bind*(*op*, *ele*, *attr*, *dop*) to build the association, *clearBind*(*ele*), *clearBind*(*ele*, *attr*) and *clearSubTreeBind*(*ele*) to clear the association when necessary, and *dp*(*op*) to denote the instruction *op*'s DOM dependences.

To trace DOM-specific dependences, we introduce the function *SearchDOMDep* to search for DOM-specific dependences. Algorithm 1 presents our searching algorithm for *SearchDOMDep*. The algorithm first searches the ancestors of *ele* for the DOM instructions that have the types of *DNAdd, DNRm, DNReplace* and *DSubtreeMod* (Line 1). Thus, we can guarantee the structural integrity (all its ancestor nodes already exist) when we access *ele*. Next, if the instruction *dop* is reading an attribute *attr* of *ele*, then current *dop* will depend on the instructions that modified this attribute (Line 3). Finally, if the current DOM instruction *dop* is related to the subtree (such as reading *innerHTML*), the subtree nodes will be searched (Line 5). We have manually inspected the DOM APIs according to DOM3 specification [15] to decide the DOM instruction type and whether it is necessary to search the subtree of *ele* (*needSearchSubtree*).

Since a DOM instruction *dop* is also a JavaScript instruction *op*, the dependence of a DOM instruction *dop* includes two parts: JavaScript dependences calculated by *JSDep*(*op*) and DOM dependences calculated by *DOMDep*(*dop*).

3.2.3. DDG and EDG

In this section, we describe how to build the Dynamic Dependence Graph (*DDG*) and the Event Dependence Graph (*EDG*).

The Dynamic Dependence Graph *DDG*(*N*, *E*) consists of a set of nodes *N* and a set of directed edges *E*. Nodes *N* are

```
onLoad:
   1. popupView = getElementById(…);
onAdd:
   2. popup = document.createElement('div')
   3. popup.innerHTML = '<div id="title"></div>...<div id="save"></div>';
   4. popupView.appendChild(popup)
   5. popup.getElementById('save').addEventListener('click',onSave);
onSave:
   6. todo.title = popup.getElementbyId('title').value;
```

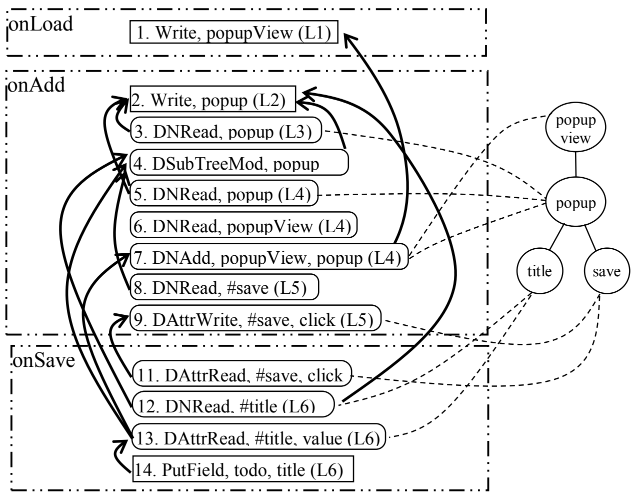


Fig. 8. Constructing DDG and EDG. The above part shows a simiplied version of source code in Listing 1. Event handers are shown in the dashed boxes. JavaScript instructions are shown in the rectangles, and DOM instructions are shown in the round rectangles. The solid arrows show dependences between instructions. Each DOM instruction is associated with its manipulating DOM elements (dashed lines). The right part shows the simplied DOM tree used by this example.

JavaScript instructions or DOM instructions, and edges *E* are the directed dependences among nodes *N*.

We build the Event Dependence Graph *EDG*(*N*, *E*) based on *DDG*. *EDG* consists of a set of event nodes *N* and a set of directed edges *E*. Each edge $e_i \rightarrow e_j$ in *E* denotes that at least one instruction in event e_i depends on an instruction in event e_j.

The example in Fig. 8 illustrates *DDG* and *EDG* including JavaScript dependence and DOM dependence. In order to clearly demonstrate DDG and EDG, we use a simplified version of the source code in Listing 1. This simplified example has three event handers: *onLoad*, *onAdd* and *onSave*. Fig. 8 shows the dependences of JavaScript and DOM instructions. Note that, in Fig. 8, each DOM instruction is associated with its manipulating DOM elements and the dependences are retrieved by searching the DOM tree in Algorithm 1. As shown in Fig. 8, the set property *innerHTML* of *popup* at line 3 (*op4*) depends on *op*2 according to the JavaScript property writing rule by resolving *JSDep*. The operation *getElementById* (Line 5) reads the value attribute of element *title* (*op*13). Thus, according to the *DAttrRead* rule, *op*13 depends on *op2* (by resolving *JSDep*) and *op*4 (by resolving *DOMDep*). The registering event handler *onSave* is treated as an attribute with a unique name (*op*9). When it is triggered by an event, an property read to this event handler is recorded (*op*11), and this operation depends on the registering operation *op*9.

EDG is built based on the *DDG*. For example, the event that triggers event handler *onSave* depends on the event that triggers *onAdd*, because there are edges between them. As we can see, the resulted *DDG* of combined JavaScript dependence analysis and DOM dependence analysis consists of 4 kinds of dependences: JS node to DOM node, DOM node to JS node, JS node to JS node, and DOM node to DOM node. JavaScript dependence analysis or DOM dependence analysis alone only form a subgraph of *DDG* in Fig. 8.

Algorithm 2. Slicing algorithm *DS*

```
Input: g (event dependence graph), e (erroneous event node to
       trace from)
Output: result (key events)
1: Set result = {}
2: Queue q = {e}; //initialized as the erroneous event
3: while q.length>0
4:     $e_i$ = q.deQueue();
5:     if !result.contains($e_i$)
6:         result.add($e_i$);// add adjacent nodes of $e_i$ in graph g
7:         for each $e_j$ in adjacentNodes(g, $e_i$)
8:             if !result.contains($e_j$)
9:                 q.enQueue($e_j$);
```

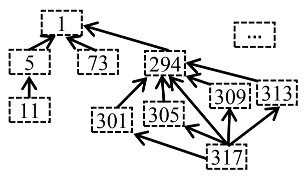


Fig. 9. A simplied EDG for the example in Fig. 2.

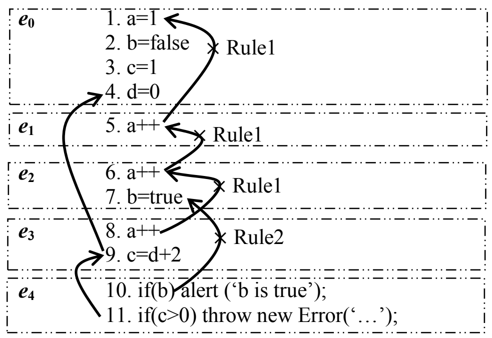
3.3. Key Event Generation

Calculating the key events related to an error is a graph reaching problem. Algorithm 2 gives our slicing algorithm *DS*. The algorithm backward traverses the EDG to find all events that the erroneous event depends on. *q* is a queue that contains the nodes that need to trace back, and is initialized as the erroneous event (Line 2). If *q* contains at least one node, then *q* dequeues a node e_i and adds it to the result set *result* (Line 6), and adds all nodes that e_i can reach in the *EDG* to the queue *q* (Lines 7~9). Therefore, we can iteratively trace from these nodes and find more reachable nodes. A simplified event dependence graph of our motivation example in Fig. 2 is shown in Fig. 9. Based on Algorithm 2, we can calculate the key events as [1, 294, 301, 305, 309, 313, 317].

3.4. Weak Dependence Analysis

Our slicing-based approach in Section 3.3 conservatively assumes that e_i depends on e_j if there is any dependence between them, without considering whether the dependence is necessary. Thus, lots of irrelevant events are still kept in the event slice. Fig. 10 shows an interesting example. Based on Algorithm 2, all events $\{e_0, e_1, e_2, e_3, e_4\}$ should be selected to reproduce the error in e_4. Although e_4 depends on e_1, e_2, and e_3, if we remove e_1, e_2, and e_3, we can still reproduce the error in e_4. This is because the condition (c>0) in e_4 can still be true after we remove e_1, e_2, and e_3. Since these events (e.g., e_1, e_2, and e_3) can be removed and do not affect the error reproducing, we say these dependences caused by these events are *weak*.

It is challenging to remove all weak dependences by statically analyzing the event trace, since we need to know whether each instruction can affect the occurrence of the error. For example, we need to know whether the path condition in Line 11 is true after removing e_3.We observe that, for an



```
e0   1. a=1
     2. b=false
     3. c=1
     4. d=0
e1   5. a++
e2   6. a++
     7. b=true
e3   8. a++
     9. c=d+2
e4   10. if(b) alert ('b is true');
     11. if(c>0) throw new Error('…');
```

Fig. 10. Code snippet for weak dependence analysis. The dependences are shown as arrows. An arrow with × and Rule1/Rule2 denotes that dependence can be removed by Rule1/Rule2. e_4 is the erroneous event.

instruction *op* that modifies a variable, if deleting *op* has no effect on the subsequent execution, then *op* can be removed without affecting the error reproducing. Based on this observation, we develop two heuristics to conservatively remove weak dependences in common cases.

- **Rule 1.** If a variable *v* is defined / modified by an instruction *op* in event *e*, and it is not used in all the path conditions in following events, and *v* is not concerned by the erroneous event, then all the dependences that depend on *op* are weak. Since variable *v* is not used in the path conditions, its value cannot change the control flow for the following events, and will not affect the error occurrence. For example, in Fig. 10, variable *a* (in e_0, e_1, e_2, and e_3) is not used in the all the subsequent condition statements and is not concerned by the e_4. Thus, three dependencies that uses variable *a* (i.e., Line8→Line6, Line6→Line5, Line5→Line1) are weak, and can be removed.
- **Rule 2.** Assume that a variable *v* is defined / modified by an instruction *op* in event *e*, and it is used in the subsequent path condition *pc*. If the code block in *pc* does not modify any state of the web application, e.g., only read some data, we can safely remove the relevant dependences. No matter whether the path condition *pc* is true or not, the state of the web application will not be changed. So, this will not affect the error occurrence. For example, in Fig. 10, e_2 modifies variable *b*, and then e_4 uses variable *b* in Line 10. This dependence (i.e., Line10→Line7) can be removed since the code block in Line 10 does not modify any state of the web applications.

Weak dependence analysis can be performed on the JavaScript or DOM dependence analysis. Note that our rules only cover common cases but cannot remove all the weak dependences. For example, in Fig. 10, e_4 depends on e_3 (Line11→Line9). This dependence (Line11→Line9) should be identified as weak dependence, because if we remove e_3, the path condition (Line 11) in e_4 still holds true. However, it is challenging to know whether the path condition in Line 11 holds true after removing e_3 by static analysis. Thus, our current rules cannot remove e_3.

3.5 Replayable Criteria

We classify symptoms of web application errors into the following three cases. 1) *The rendering errors* (e.g., missing UI components). Such errors can always be observed on the web pages. 2) *Unhandled exception thrown by the program*. Such errors can cause the code termination or be observed by the debugging tools. 3) *A specific piece of code specified at source code.* It can also be the specific piece of code that is expected to be executed.

The criterion for web application error reproducibility is whether the assertions of the above symptoms hold. Our tool can automatically insert user-defined assertions for a given event when replaying the event trace.

4. Implementation

Our implementation JSTrace is based on JavaScript record-replay. We adopted a similar record-replay mechanism to Mugshot [8]. We extended the replay phase to support event trace reduction. JSTrace is entirely written in JavaScript and transparent to users. Thus, JSTrace is easy to deploy.

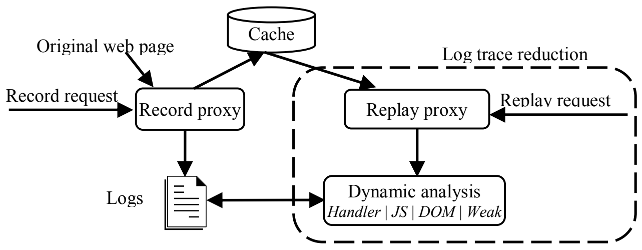


Fig. 11. The architecture of JSTrace

Architecture. Fig. 11 shows JSTrace's architecture. At the record phase, the *record proxy* retrieves the original web page and instruments it to record the event trace. The recorded event trace is periodically updated to the server and stored in a log file. The *Cache* is used to store web pages and nondeterministic data from web server, and makes sure that the replay phase will get the same data as that at the record phase. The *replay proxy* is designed to replay a given event trace. *Dynamic analysis* module can perform event trace reduction at several levels, i.e., unhandled event analysis (*Handler*), JavaScript dependence analysis (*JS*), DOM dependence analysis (*DOM*), and weak dependence analysis (*Weak*). Users can record or replay an event trace at a given level by simply switching a proxy in their browser. JSTrace also provides a web interface to show all recorded event traces and corresponding reduced traces with corresponding dependence details.

Instrumentation. We instrument JavaScript code using Jalangi [10] to capture all executed JavaScript (including DOM) instructions. The instrumentation is patched to the client side code, thus the dynamically generated code in JavaScript can be instrumented as well, such as *eval*, *setTimeout* and *setTimeinterval*. All the JavaScript instructions in libraries are instrumented, too.

JavaScript dependence analysis. To trace dependences dynamically, we incorporate the idea of shadow execution [10], in which the analysis can update and access the shadow value of a variable *v*. The shadow value records the value of *def*(*v*). For simplicity, we define a shadow member for each value using the JavaScript API *defineProperties* with the option enumerable configured as *false*, since this added shadow member should not be seen by the original code. When the value of variable *v* is

Table 3. Real-world applications and errors.

Apps	Description	Stars	JS size	Error	Issue
Chart.js[17]	Basic charts	14,803	105K	1	503
				2	920
Handsontable[18]	Excel-like data grid editor	4,989	4.7M	3	1366
				4	638
				5	2231
JPushMenu[19]	A menu library	134	1.5M	6	1
TodoList[14]	Offline calendar	19	312K	7	
FullPage[20]	Create full screen scrolling websites	9,518	882K	8	146
Editor.md[21]	A markdown editor	530	257K	9	18
My-mind[22]	Online mind mapping	1,449	223K	10	12
Foundation[23]	Responsive front-end framework	22,885	576K	11	7528
Reveal[24]	HTML presentation framework	26,893	424k	12	463
KodExplorer[25]	Online file manager	585	5.8M	13	

Table 4. Web application errors collected from [11].

Subjects	Error type	LOC	Error
Canada	Incorrect values	105	14
OnlineShopping	Cannot add items to cart	30	15
AgeCaculate	Invalid calculation	114	16
CarRental	Unresponsive DatePicker	125	17
Insurance	Form submitted with empty field	93	18
StudentInfo	Invalid input	92	19
Airport	Invalid Input	44	20
BestCars	Invalid Input	38	21
Game	Faulty button click	68	22
Numbers	Incorrect calculation	118	23
Patient form	Form submitted with empty field	93	24
Dentist form	Invalid Input	86	25

modified, the shadow value is updated with the *id* of the operating instruction.

DOM dependence analysis. We regard a JavaScript instruction as a DOM instruction if the operating object or returned value is DOM element. The information that are used by *SearchDOMDep* (Algorithm 1) is determined by the semantic of the API according to DOM specification [15], e.g., *input.value* is a DNRead instruction performed on DOM element *input*, and it is unnecessary to search the subtree of *input* to find dependences.

Weak dependence analysis. Weak dependence analysis is an optimization process that is performed on the basis of other analysis such as JS or DOM analysis. For Rule 1 in Section 3.4, we ignore the instructions whose operated variables are not concerned by the erroneous event and are not depended by the variables in the subsequent path conditions. For Rule 2 in Section 3.4, we modify the instrumentation module of Jalangi [10] to capture the enter point and the exit point of each branch for the corresponding path condition, we further judge whether there are writing instructions between these two points.

5. Evaluation

In this section, we measure reproducibility and efficiency of JSTrace on 13 real-world errors from 10 different open source web applications in GitHub. Besides, we also evaluate JSTrace on 12 errors used by [11], which were collected from StackOverflow [16]. Thus, we can measure whether JSTrace is applicable to other error dataset. Specifically, we investigate the following four research questions:

RQ1: Can the reduced event traces faithfully reproduce the errors?

RQ2: Can JSTrace efficiently reduce the event trace?

RQ3: Is JSTrace's performance acceptable?

RQ4: Is JSTrace applicable to other error dataset?

5.1. Experimental Subjects and Methodology

To answer research questions RQ1-3, We evaluated JSTrace on 13 real-world errors from 10 different open source web applications in GitHub. In order to select representative web applications, we first used the condition "language:JavaScript tag:bug comment:>2" and the keywords *repro*, *steps,* or *sample* (i.e., common used words in a bug report describing reproducing steps) to collect web applications that satisfy the following criteria: they contain JavaScript; the issues are marked as bugs with multiple comments; and they have descriptions for reproducing. From the result, we made further study on the errors, and selected the errors that can be manually reproduced and have certain difficulty to diagnose (with multiple steps to reproduce). Finally, we classified them to different categories and bias the applications that are more popular, weighed by the number of stargazers in GitHub. Since some applications do not maintain issue lists in Github (e.g., KodExplorer and TODOList in Table 3), but they document their changes in a change log file, we also search the change logs to find bugs using the above criteria. Finally, we have selected 13 real-world errors form 10 web applications.Table 3 provides an overview of our evaluated web applications. These web applications are designed for different purpose and functionality. For example, chart.js [17] is a drawing library, Handsontable [18] is an excel-like application, and TodoList is an offline HTML5 application that works like a calendar. These web applications are complicated, for example, KodExplorer uses 5.8M JavaScript code.

To answer RQ4, we used web application errors that were collected from StackOverflow by [11]. We chose 12 out 30 of their subjects from their project website [26]. The other 18 errors were removed for the following reasons: (1) The length of reduced event trace is 1. We prior to selecting complicated errors which depends on other events. (2) We cannot reproduce the errors. Table 4 shows the details about the 12 selected web application errors. As we can see, these applications are relatively small, but they provide diverse functionality. Since they have already been used in [11], we regard them as representative errors in real-world web applications.

Since we did not have the original event trace to trigger these 13 errors in Table 3, we ran these applications for a while, and finally triggered the errors. Table 5 shows the details about event traces and the reduced event traces. The column *ALL* shows the number of events in the original event traces. The column *Expected* shows the number of the minimal events to reproduce the corresponding errors. In order to evaluate how effective JSTrace is, we compare JSTrace in different granularity of JavaScript and DOM dependence, e.g., unhandled event analysis (*Handler*) and JavaScript dependence analysis without consider DOM dependence (*JS*), combination of JS and coarse-

Table 5. Reduction results on real-world errors.

Error	Event trace		Slicing-based approach					Weak dependence		
	#All	#Expected	#Handler/R	#JS/R	# CG_DOM/R	# FG_DOM/R	Rate1	#Weak/R	Rate2	Improvement
1	1139	6	351/Y	52/N	79/Y	58/Y	95.4%	44/Y	96.6%	27%
2	1168	6	770/Y	139/Y	139/Y	92/Y	92.6%	82/Y	93.5%	12%
3	403	5	345/Y	29/Y	297/Y	59/Y	86.4%	37/Y	92.0%	41%
4	694	3	668/Y	28/Y	81/Y	58/Y	92.0%	27/Y	96.5%	56%
5	606	6	462/Y	34/Y	51/Y	34/Y	95.3%	15/Y	98.5%	68%
6	342	2	6/Y	2/Y	6/Y	2/Y	100.0%	2/Y	100.0%	0%
7	1410	3	851/Y	21/N	43/Y	24/Y	98.5%	23/Y	98.6%	5%
8	398	3	39/Y	30/Y	36/Y	30/Y	93.2%	29/Y	93.4%	4%
9	1023	2	791/Y	9/Y	72/Y	9/Y	99.3%	8/Y	99.4%	14%
10	1454	6	351/Y	8/Y	22/Y	8/Y	99.9%	8/Y	99.9%	0%
11	567	2	62/Y	30/Y	56/Y	32/Y	91.8%	5/Y	99.2%	90%
12	617	5	500/Y	12/Y	12/Y	12/Y	98.9%	12/Y	98.9%	0%
13	3461	3	2118/Y	17/N	27/Y	26/Y	99.3%	16/Y	99.6%	43%
Average reduction rate							96%		97%	28%

grained DOM analysis (*CG_DOM*), combination of JS and fine-grained DOM analysis (*FG_DOM*), and weak dependence analysis performed on the basis of FG_DOM (*Weak*). In coarse-grained DOM dependence analysis, we manually identifies whether an DOM operation is writing or reading DOM elements. We simply consider the DOM as a whole object when building DOM dependence. In fine-grained DOM dependence analysis, we use the approach in Section 3.2.2. The rate columns (i.e., *Rate1* and *Rate2*) measure how many irrelevant events to the corresponding error are removed. The column *Improvement* shows the improvement for each error when performing FG_DOM analysis (JavaScript and fine-grained DOM dependence) with/without weak dependence analysis.

5.2. RQ1: Reproducibility

To address research question RQ1, we validated whether the reduced trace can faithfully reproduce the corresponding error. Table 5 shows the result on 13 web application errors. The *R* flag with value *Y* or *N* represents whether the reduced trace can successfully reproduce the corresponding error or not.

As shown in Table 5, the unhandled event analysis can significantly reduce event traces, and the resulted event traces are reproducible (*Handler/R*). However, the resulted event traces are still quite long to diagnose. For example, for errors 2 and 9 we remove only less than a half irrelevant events, but the length of expected event trace is usually no more than 6 (*Excepted*).

JavaScript dependence analysis (*JS/R*) and fine-grained DOM dependence analysis (*FG_DOM/R*) can further reduce the event traces. However the resulted event traces of JavaScript dependence analysis may not be reproducible since JavaScript dependence analysis ignore DOM dependence. In the 13 errors, 3 of them cannot be reproduced. The fine-grained DOM dependence analysis (*FG_DOM/R*) and weak dependence analysis (Weak/R) performs best in reproducibility, all the errors are successfully reproduced. The weak dependence analysis safely removes irrelevant events and obtains the closest event traces to the expected ones.

Due to the missing DOM dependences, the reduced event traces of JavaScript dependence analysis failed to reproduce some errors. This indicates that DOM dependence analysis is necessary for event trace reduction in web applications.

5.3. RQ2: Efficiency

We use the reduction rate to evaluate the efficiency of event trace reduction. Table 5 shows the result. The column *Rate1* shows the reduction rate of our combined JavaScript and fine-grained DOM dependence analysis without applying weak analysis, and is calculated by (ALL – FG_DOM) / (ALL – Expected). The 100% reduction rate indicates that all the error-irrelevant events are removed. In Table 5, the average reduction rate is 96%. Note that fine-grained DOM dependence (FG_DOM) can remove more irrelevant events compared to the coarse-grained DOM dependence (CG_DOM). On average, FG_DOM can remove 7% (calculated by (CG_DOM – FG_DOM) / (ALL – Expected)) more of irrelevant events. This shows that our fine-grained DOM analysis is necessary.

However, the result of FG_DOM still contains irrelevant events. We dug into the errors and found that the unremoved irrelevant events are mostly caused by the following cases: 1) *Array operation*, since we treat an array as a single object. The applications Chart.js and HandsonTable suffer from this problem. 2) *setTimeout and setTimeInterval*. Such function calls are often used to periodically check and modify a shared data or showing animations, and result in large amount of dependences and lower the reduction rate. The applications Chart.js and FullPage suffer from this problem. 3) *Redundant data dependences*. For example, an event e_{i+1} reads the value of variable *guid*, which is set by a previous event e_i. Thus, there is a JavaScript dependence between e_i and e_{i+1}. However, the error does not care about the exact value of variable *guid*. Thus, the event e_i is not expected to be included in the final reduced event trace. Almost all the applications suffer from this problem.

Our weak dependence analysis can remove some irrelevant events in the above cases. The whole reduction rate is 97% when applying weak dependence analysis (column *Rate2* in Table 5). Our weak dependence analysis can remove 28% of irrelevant events from FG_DOM (column *Improvement* in Table 5, calculated by (FG_DOM – Weak) / (FG_DOM – Expected)). As we can see in Table 5, 10 out 13 errors benefit from our rule-based weak dependence analysis (with positive improvement) and more than a half of irrelevant events are removed in some cases (e.g., errors 5 and 11). This shows that our rules cover many cases although it cannot remove all the weak dependencies. For 3 errors (i.e., errors 6, 10, 12), we have no improvement. It

Table 6. Overhead.

Error	Original		Dynamic slicing			
	Time(s)	Mem(MB)	Time(s)	Time(X)	Mem(MB)	Mem(X)
1	8	4.5	14	1.75	12.2	2.71
2	10	4.7	18	1.8	30.5	6.49
3	11	10.3	47	4.3	64.1	6.22
4	12	7.5	38	3.2	74.4	9.92
5	15	23.4	83	5.5	45.3	1.94
6	4	12.5	6	1.5	21.3	1.70
7	21	12.4	237	11.3	161	12.98
8	7	4.8	15	2.1	64	13.33
9	17	11.2	143	8.4	144	12.90
10	13	11.6	64	4.9	16.3	1.41
11	5	5.4	30	6	30.6	5.67
12	9	11.3	45	5	33.1	2.93
13	17	7.7	72	4.2	40.8	5.30

is because the reduced event trace of error 6 is already minimal, and our rules do not work for errors 10 and 12, since our rules do not cover the case that a modified variable is used in a subsequent path condition and the corresponding blocks contain write operations (as discussed in Section 3.4).

In summary, JSTrace can remove 97% irrelevant events on average. Our weak dependence analysis can further remove 28% of irrelevant events that cannot be removed by our previous work [13].

5.4. RQ3: Performance

Time overhead. The time cost for the original run with / without JSTrace analysis is shown in Table 6. As we can see, the time overhead is 1.5~11.3X.

Memory overhead. We have evaluated the memory usage of the 13 errors on Google Chrome and taken a heap snapshot on the profiles tab of the developer tool when the execution is ended. We used this profiler to take snapshot of JavaScript heap only, thus this size does not include the images, canvas, audio files, plugin data or native memory.

Table 6 shows the memory usage of the original and JSTrace. The extra memory is used to record shadow values and DOM searching information. The result shows that the overhead of memory is 1.4~13.3X.

Although the time overhead and memory overhead is huge for the 13 errors in our experiments, JSTrace can still be used in practice. In common cases, JavaScript code for a single web page should not be very large, and 1MB+ will be considered large. The size of our evaluated application "handsontable" has exceeded 4.7 MB, and our approach can handle it well. Therefore, our approach can handle practical web applications properly.

5.5. RQ4: Applicability

We applied JSTrace on the 12 application errors, which were also used in recent work [11]. Table 7 shows our result. We can see that, in these errors, all the irrelevant events are removed, and the corresponding errors can still be reproduced. Because these 12 errors have simple data flow and many of the events are user-input events, JSTrace performs much better on these errors than our 13 real-world errors.

Table 7. Reduction result on applications from [11].

Error	#Original	#Excepted	#Reduced/R
14	17	2	2/Y
15	10	2	2/Y
16	9	3	3/Y
17	11	2	2/Y
18	10	2	2/Y
19	9	2	2/Y
20	6	2	2/Y
21	4	2	2/Y
22	13	2	2/Y
23	8	2	2/Y
24	8	2	2/Y
25	9	2	2/Y

```
1.   showInfoDialog : function() {
2.       ...
3.       var infoDialog = editor.find("." + classPrefix + "dialog-info");
4.       if (infoDialog.length < 1)
5.       {
6.           this.createInfoDialog(); // create an InfoDialog
7. +         infoDialog  = editor.find("."+classPrefix+"dialog-info");//fix
8.       }
9.       infoDialog.show();
10.        ...
11.       return this;
```

Fig. 12. Code snippet for Editor.md.

6. Case Study

JSTrace can help diagnose web applications errors not only by removing irrelevant events, but also by providing summaries of dependences between statements and associated events. Thus, JSTrace can facilitate developers to inspect the error-related code. The previous user study [11] has investigated *to what extent a reduced recording assists programmers in the debugging process given a faulty web application*. The overall result shows that the reduced recordings significantly increased programmers' efficiency in failure detection, fault localization and fault correction. Therefore, in this section, we performed several case studies and focused on validating how the provided dependence summaries can help error diagnosis (usefulness of JSTrace). The summaries are denoted as a set of dependencies in form of $(s_i, e_i)\rightarrow(s_j, e_j)$, which means statement s_i depends on statement s_j, and s_i is executed in event e_i, s_j is executed in event e_j. In the following code snippets (e.g., Fig. 12), the code separated by a line is executed during different events.

6.1. Editor.md

Editor.md is an open source embeddable online markdown editor. There is a functionality error related to a button. When the button is clicked, a dialog is expected to show up. However, the dialog does not show up until it is clicked twice. Function *showInfoDialog* in Fig. 12 is the event handler for the click on the button.

For this problem, we may guess that the error is caused by forgetting to show up the dialog. We can exclude this possibility since JSTrace observes that *infoDialog.show*() (Line 9 in Fig. 12) is executed twice during the two click events. When we inspect Line 9 and its dependent statements, it is easy to find that, the code is written in a common singleton pattern judge-create-and-use (i.e., create the dialog if it does not exist, and then use it). For the first click e_1, *infoDialog* is an array with 0 element since

```
1.  $(that.options.item).attr('data-value', item) //initilize the dropdown items
2.  …
3.  instance.autocompleteEditor.bindTemporaryEvents(td, row, col, prop…);
4.  …
------------------------------------------------------------------
5.  mouseenter = function(){  //mouseover a dropdown item
6.      ...
7.      $(e.currentTarget).addClass('active')
8.  }
------------------------------------------------------------------
9.  HandsontableAutocompleteEditorClass.prototype.bindTemporaryEvents
10. = function(td, row, col, prop, value, cellProperties){
11.     this.typeahead.select = function () {
12.         var val = this.$menu.find('.active').attr('data-value');
13.         ...
14.         that.instance.setDataAtRowProp(row, prop, val);
15.         return output;
16.     };
17. }
18. HandsontableTextEditorClass.prototype.finishEditing
19. =function (isCancelled, ctrlDown) {
20.     ...
21.     var val = [[$.trim(this.TEXTAREA.value)]];
22.     ...
23.     populateFromArray({row: this.row, col:this.col},val…);
24. };
25.  HandsontableAutocompleteEditorClass.prototype.finishEditing
26. = function(isCancelled, ctrlDown){
27.     this.typeahead.select();
28.     this.isCellEdited = false;
29.     HandsontableTextEditorClass.prototype.finishEditing.call(this …);
30. }
```

Fig. 13. Code snippet for Handsontable.

```
1.  removeData : function(){
2.      ...
3.      this.valuesCount--; //when remove a data from a chart
4.  }
------------------------------------------------------------------
5.  calculateX : function(index){
6.      var isRotated = (this.xLabelRotation > 0),
7.      innerWidth = this.width - (this.xScalePaddingLeft +
  this.xScalePaddingRight),
8.      valueWidth = innerWidth/(this.valuesCount - //devision by 0
         ((this.offsetGridLines) ? 0 : 1)),
9.      valueOffset = (valueWidth * index) +
         this.xScalePaddingLeft;
10.     ...
11.     return Math.round(valueOffset);
12. }
13. calculateBarX : function(datasetCount, datasetIndex, barIndex){
14.     var xWidth = this.calculateBaseWidth(),
15.     xAbsolute = this.calculateX(barIndex) - (xWidth/2),
16.     barWidth = this.calculateBarWidth(datasetCount);
17.     return xAbsolute + (barWidth * datasetIndex) +
                (datasetIndex * options.barDatasetSpacing) + barWidth/2;
18. }
19. addData : function(valuesArray,label){
20.     ...
21.     this.datasets[datasetIndex].bars.push(new this.BarClass({
22.         x: this.scale.calculateBarX(this.datasets.length, dataset
              Index, this.scale.valuesCount+1),
23.         ...
24.     }));
25.     this.scale.addXLabel(label);
26.     this.update(); // will read array this.datasets
27. }
```

Fig. 14. Code snippet for Chart.js.

no dialog with class name "dialog-info" exists (Line 3). Then the dialog with class name "dialog-info" is created at Line 6, however the *infoDialog* is not updated as expected (like the fix at Line 7) and still holds nothing. Thus, no dialog shows up when calling *infoDialog.show* (Line 9). While for the second click e_2, *infoDialog* is an array with 1 element since a "dialog-info" dialog has been created after the first click (Line 6), thus a dialog can popup (Line 9).

Summary. This case shows that JSTrace can facilitate error diagnosis by providing dependence information: (Line 3, e_2)→(Line 6, e_1), (Line 9, e_2)→(Line 3, e_2). The developers only need to inspect the code in the dependence chain, which significantly reduces the amount of code to inspect.

6.2. HandsonTable

HandsonTable is an excel-like web application. We use <x, y> to refer to the cell at row x and column y. The following steps reveal an error. e_1: A user clicks the dropdown component of cell <3, 3> and a dropdown list pops up. e_2: The user moves his mouse over an item in the dropdown list. e_3: The user clicks another cell <3, 2>. Surprisingly, both <3, 2> and <3, 3> have been changed to an unexpected value while they are expected to keep unchanged.

JSTrace can trace where the unexpected value come from. As the source code shown in Fig. 13, cell <3, 3> is mistakenly updated at Line 14 with the value of the item, which the user moves mouse over. The parameter *row* (Line 14) comes from the function call parameter at Line 3 (i.e., (Line 14, e_3)→(Line 3, e_1)). The parameter *val* comes from Line 12 (i.e., (Line 14, e_3)→(Line 12, e_3)) which read the value of the item that the user moves mouse over (i.e., (Line 12, e_3)→(Line 7, e_2)).

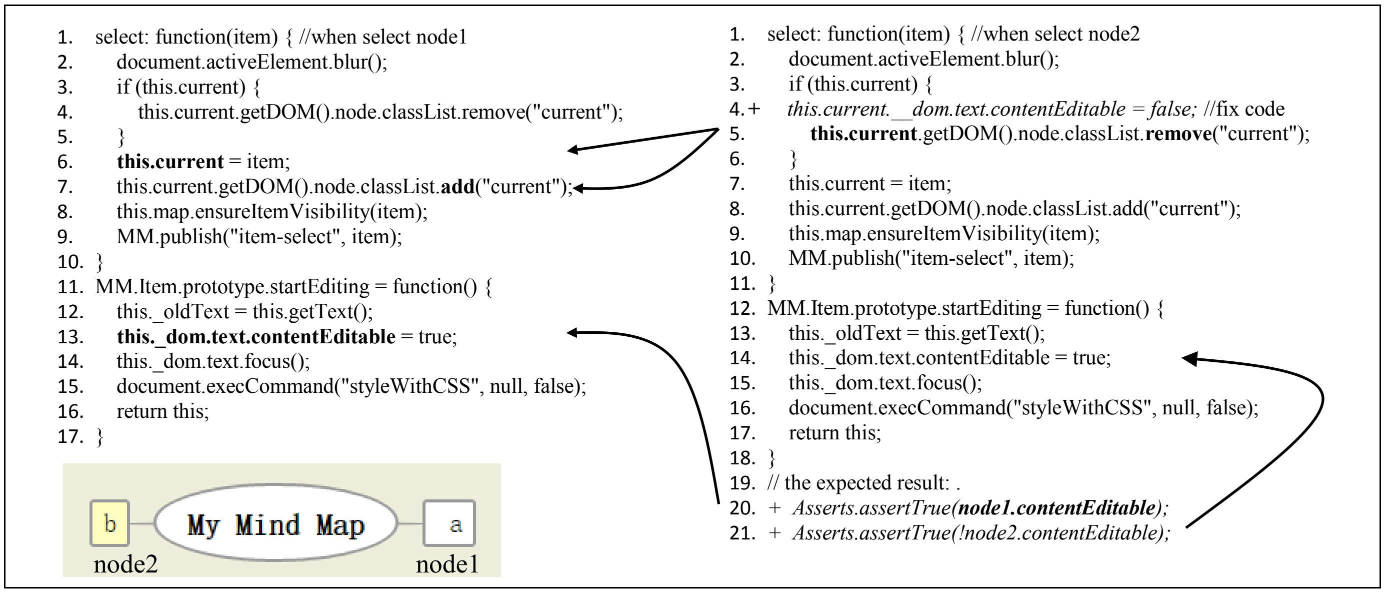
Summary. Although web application uses large amount of puzzling function closures and variable scopes (such as the function in Lines 3, 9 and 11, which make the control flow really hard to understand), JSTrace can make it easier to trace the abnormal variables.

6.3. Chart.js

Chart.js is a HTML5 drawing library using web canvas [27]. An error occurs when a user clears a chart and then adds a data to it. From the user's view, no data has been added to the chart since the x coordination is mistakenly calculated as a value that exceeds the visible boundary of the screen.

As shown in Fig. 14, function *addData* (Line 19) is called when adding a data to the chart, and function *removeData* (Line 1) is called when removing a data from the chart. We set statement at Line 26 to trace from since this line of code is responsible for rendering the chart. By inspecting the statements along the dependence chain, the developer can efficiently locate the error which is caused by the division by zero error at Line 8 and finally inducing a value *Infinity* as the x coordination which is out of the visible boundary.

Summary. The root cause of an error may be far from where it manifests. By inspecting the statements along the dependence chain, developers can reduce the diagnosis complexity.



```
1.  select: function(item) { //when select node1
2.      document.activeElement.blur();
3.      if (this.current) {
4.          this.current.getDOM().node.classList.remove("current");
5.      }
6.      this.current = item;
7.      this.current.getDOM().node.classList.add("current");
8.      this.map.ensureItemVisibility(item);
9.      MM.publish("item-select", item);
10. }
11. MM.Item.prototype.startEditing = function() {
12.     this._oldText = this.getText();
13.     this._dom.text.contentEditable = true;
14.     this._dom.text.focus();
15.     document.execCommand("styleWithCSS", null, false);
16.     return this;
17. }
```

```
1.  select: function(item) { //when select node2
2.      document.activeElement.blur();
3.      if (this.current) {
4.+     this.current.__dom.text.contentEditable = false; //fix code
5.          this.current.getDOM().node.classList.remove("current");
6.      }
7.      this.current = item;
8.      this.current.getDOM().node.classList.add("current");
9.      this.map.ensureItemVisibility(item);
10.     MM.publish("item-select", item);
11. }
12. MM.Item.prototype.startEditing = function() {
13.     this._oldText = this.getText();
14.     this._dom.text.contentEditable = true;
15.     this._dom.text.focus();
16.     document.execCommand("styleWithCSS", null, false);
17.     return this;
18. }
19. // the expected result: .
20. + Asserts.assertTrue(node1.contentEditable);
21. + Asserts.assertTrue(!node2.contentEditable);
```

Fig. 15. Code snippet for my-mind.

6.4. My-mind

My-mind is a web application for creating and managing mind maps. Fig. 15 shows an error snapshot and the related code snippet. When a user selects a node and edits it, function *select* (Line 1) and *startEditing* (Line 11) will be executed, respectively. If the user selects *node1* and edits it (without pressing an enter button to finish editing), then directly selects *node2* and starts editing. Then the user will see an error: *node2* is not editable and the input content is appended to *node1*.

To diagnose this error with JSTrace, we added two assertions about the observed symptoms that node1 is editable and node2 is not (Line 19~20). As the figure shows, the left part of the code is executed when selecting and editing *node1* (e_1), and the right part of the code is executed when selecting and editing *node2* (e_2). By inspecting the related statements, we find a violation: *node1* is set editable when editing *node1* (i.e., (Line 19, e_2)→(Line 13, e_1)). The user could see *node2* is colored yellow as expected since it is rendered with proper style (Line 5). However, the invisible attribute *contentEditable* is wrong. A possible bug fix is the code at Line 4 at right part of the figure.

Summary. This case implies that the provided dependence information especially DOM dependence is helpful for inspecting concerned variables.

7. Discussion

While our evaluation shows that JSTrace is promising for reducing event traces and diagnosing errors in web applications, we discuss some threats and issues in our approach and evaluation.

7.1. Threats to Validaty

A threat to our evaluation is that we only evaluated our approach on the 13 real-world errors from 10 applications. These 10 applications were randomly selected from real-world open source projects, and developed for different purposes. They have detailed descriptions for the reproducing of the errors, and make our evaluation repeatable. Hence, they have reasonable representativeness. Besides, multiple steps are needed to reproduce the errors, thus they also have reasonable complexity.

Non-determinism may make our approach fail. Our approach can record all the non-deterministic sources. This makes the replaying and analyzing deterministic and repeatable.

Additionally, all the errors considered occur in a single page that may be a potential source of bias. However, our approach has recorded all the non-determinism to make sure that the replaying starts with a determined environment and the execution will load the same data [8]. Therefore, from the point of reproducibility, there is no need to trace events across pages. Execution on each page will generate an independent event trace.

7.2. Limitations

Our rules for weak dependence analysis only cover common cases and are incomplete, thus we cannot remove all the weak dependences. Removing all weak dependences is out of the scope of this paper. We leave this for future work.

8. Related Work

In this section, we focus on those pieces of work that concern record/replay in web applications, web application dynamic slicing, and other techniques on event-based application debugging.

Record/replay in web applications. Mugshot [8] is a high performance record-replay system that captures all events and nondeterministic information (e.g., AJAX requests, random calls, and timers) to make sure the replaying phase loads the same data. DoDOM [28] records user interaction events, thus web applications can be repeatedly executed using the captured event sequence. WaRR [29] records user interactions in a web application and uses the recorded interaction trace to perform high-fidelity replay of the web application. Ripley [30] replicates execution of the client-side JavaScript application on the server replica to automatically preserve the integrity of a distributed computation. Our record-replay component applies the same technique as Mugshot [8]. For all these work, event trace reduction are out of their scope.

Web application dynamic slicing. Dynamic slicing [31][32][33] is more useful in program debugging and testing than static slicing, several approaches for computing dynamic slicing through building a reachability-graph or a dynamic dependence graph [31][32]. Josip [34] utilizes dynamic slicing to extract client-side web application code for the purpose of program understanding, debugging and feature extraction. They use the statements that reveal target behavior as slicing criteria and perform dynamic program slicing to identify the related CSS, HTML, and JavaScript. However, their work does not care which event the execution is performed, their approach of capturing dependences is inefficient to resolve our challenges. They use parent-child relation to form structural dependence edges between DOM elements and use the parsed AST to build dependences between JavaScript statements. CLEMATIS [7] corporates the dynamic slicing technique to assist developers understand the root cause of test assertion failures by linking a test assertion failure to the JavaScript statements that are responsible for the checked DOM elements, but they do not consider the non-DOM test assertions. Autoflox [35] performs dynamic slicing for locating the DOM access that introduces a fault, rather than pruning events. It traces the execution of JavaScript code, and analyze this trace backward until one possible DOM access that returns incorrect value is found. While our approach prunes and does not only focus on DOM.

Other techniques on event-based application debugging. For example, EFF [31] combines dynamic slicing and checkpoint techniques to provide a record-replay tool that can reduce event trace and thus support long executions. They also build an EDG to calculate the event slice, however, their approach does not fit our cases. The word 'event' in their context refers to system calls and their way to build data dependences is not suitable for web applications. Our work differs from theirs for DOM-specific challenges and the way in which the graph is constructed. We combine dynamic slicing and shadow execution techniques to trace JavaScript programs. Thus, we can avoid resolving the complex dynamic features of JavaScript. AppDoctor [36] uses heuristic rules to reduce the event sequence. It takes advantage of the specific characteristics of Android events and compares the states of Android UI that is relatively simple. Their rules are simple and the approach will fall back to the worst cases if the rules do not work. Our work presents the fine-grained dependence models to build and propagate data dependences. The work [11] aims to reduce event traces using delta debugging technique that treats the operations as black boxes. Their approach relies on trial and error to decide which inputs to discard, instead of dependence analysis.

9. Conclusion

In this article, we propose a tool JSTrace to identify the key events related to a web application error. Given the expected symptom that we should reproduce, we precisely trace the dependences between JavaScript and DOM instructions, and develop a novel dynamic slicing approach to filter out irrelevant events. Further, we can remove irrelevant events that are still depended by the error. The evaluation on real-world web application errors shows that JSTrace can greatly (97%) reduce the event trace, and achieves 100% reproducibility. Case studies reveal that our dependence analysis is also useful for error diagnosis.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (61672506), National Key Research and Development Plan (2016YFB1000803), and Beijing Natural Science Foundation (4164104).

References

[1] "Gmail." [Online]. Available: https://mail.google.com.

[2] "Google Doc." [Online]. Available: https://docs.google.com.

[3] "Facebook." [Online]. Available: https://www.facebook.com.

[4] S. Thummalapenta, P. Devaki, S. Sathishkumar, S. Sinha, S. Chandra, S. Gnanasundaram, D. D. Nagaraj, S. S. Kumar, and S. S. Kumar, "Efficient and change-resilient test automation: An industrial case study," in Proceedings of International Conference on Software Engineering(ICSE), 2013, pp. 1002–1011.

[5] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "An Empirical Study of Client-side JavaScript Bugs," in International Symposium on Empirical Software Engineering and Measurement(ESEM), 2013, pp. 55–64.

[6] G. Li, E. Andreasen, and I. Ghosh, "SymJS: Automatic Symbolic Testing of JavaScript Web Applications," in Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering(FSE), 2014, pp. 449–459.

[7] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman, "Understanding JavaScript Event-based Interactions," in Proceedings of International Conference on Software Engineering (ICSE), 2014, pp. 367–377.

[8] J. Mickens, J. Elson, and J. Howell, "Mugshot : Deterministic Capture and Replay for JavaScript Applications," in Proceedings of the USENIX Conference on Networked Systems Design and Implementation(NSDI), 2010, pp. 159–174.

[9] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst, "Interactive Record/Replay for Web Application Debugging," in Preceedings of User Interface Software and Technology (UIST), 2013, pp. 473–484.

[10] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: a selective record-replay and dynamic analysis framework for JavaScript," in Proceedings of the Joint Meeting on Foundations of Software Engineering(ESEC/FSE), 2013, pp. 488–498.

[11] M. Hammoudi, B. Burg, G. Bae, and G. Rothermel, "On the Use of Delta Debugging to Reduce Recordings and Facilitate Debugging of Web Applications," in Proceedings of Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software (ESEC/FSE), 2015, pp. 333–344.

[12] "JavaScript." [Online]. Available: http://en.wikipedia.org/wiki/JavaScript.

[13] J. Wang, W. Dou, C. Gao, and J. Wei, "Fast Reproducing Web Application Errors," in Preceedings of International Symposium on Software Reliability Engineering(ISSRE), 2015, pp. 530–540.

[14] "TodoList." [Online]. Available: https://github.com/01org/webapps-todo-list.

[15] "Document Object Model Core." [Online]. Available: http://www.w3.org/TR/DOM-Level-3-Core/core.html.

[16] "stackoverflow." [Online]. Available: http://stackoverflow.com/.

[17] "Chart.js." [Online]. Available: https://github.com/nnnick/Chart.js.

[18] "handsontable." [Online]. Available: https://github.com/handsontable/handsontable.

[19] "jPushMenu." [Online]. Available: https://github.com/takien/jPushMenu.

[20] "fullPage.js." [Online]. Available: https://github.com/alvarotrigo/fullPage.js/issues/146.

[21] "editor.md." [Online]. Available: https://github.com/pandao/editor.md.

[22] "my-mind." [Online]. Available: https://github.com/ondras/my-mind.

[23] "foundation-sites." [Online]. Available: https://github.com/zurb/foundation-sites.

[24] "reveal.js." [Online]. Available: https://github.com/hakimel/reveal.js.

[25] "KODExplorer." [Online]. Available: https://github.com/kalcaddle/KODExplorer.

[26] "webapps-delta-debugging." [Online]. Available: https://github.com/gigony/webapps-delta-debugging.

[27] "html5 canvas." [Online]. Available: http://www.w3schools.com/html/html5_canvas.asp.

[28] K. Pattabiraman and B. Zorn, "DoDOM: Leveraging DOM Invariants for Web 2.0 Application Robustness Testing," in Proceedings of the International Symposium on Software Reliability Engineering(ISSRE), 2010, pp. 191–200.

[29] S. Andrica and G. Candea, "WaRR: A tool for high-fidelity web application record and replay," in IEEE/IFIP International Conference on Dependable Systems & Networks (DSN), 2011, pp. 403–410.

[30] K. Vikram, A. Prateek, and B. Livshits, "Ripley : Automatically Securing Web 2 . 0 Applications Through Replicated Execution Categories and Subject Descriptors," in Proceedings of ACM conference on Computer and communications security (CCS), 2009, pp. 173–186.

[31] X. Zhang, S. Tallam, and R. Gupta, "Dynamic Slicing Long Running Programs Through Execution Fast Forwarding," in Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2006, pp. 81–91.

[32] M. Weiser, "Program slicing," in Proceedings of the International Conference on Software Engineering(ICSE), 1981, pp. 439–449.

[33] A. De Lucia, "Program slicing: methods and applications," in Proceedings of IEEE International Workshop on Source Code Analysis and Manipulation, 2001, pp. 142–149.

[34] J. Maras, J. Carlson, and I. Crnkovi, "Extracting client-side web application code," in Proceedings of international Conference on World Wide Web(WWW), 2012, pp. 819–828.

[35] F. S. Ocariza, K. Pattabiraman, and A. Mesbah, "AutoFLox: An automatic fault localizer for client-side JavaScript," in Proceedings of IEEE International Conference on Software Testing, Verification and Validation(ICST), 2012, pp. 31–40.

[36] G. Hu, X. Yuan, Y. Tang, and J. Yang, "Efficiently, effectively detecting mobile app bugs with AppDoctor," in Proceedings of the European Conference on Computer Systems(EuroSys), 2014, pp. 1–15.