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
[‡]University of Chinese Academy of Sciences
{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 features and event-driven design, it is not easy to debug web application errors. Record-replay techniques are widely used to reproduce errors in web applications. However, the key events related to an error are hidden in the massive event trace collected during a long running. As a result, error diagnosis with the long event trace is exhausting and time-consuming.**

We present a tool JSTrace that can effectively cut down the web application error reproducing time and facilitate the diagnosis. Based on the dynamic dependencies of JavaScript and DOM instructions, we develop a novel dynamic slicing technique that can remove events irrelevant to the error reproducing. In this process, many events and related instructions are removed without losing the reproducing accuracy. Our evaluation shows that the reduced event trace can faithfully reproduce errors with an average reduction rate of 96%.

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

I. Introduction

Web applications have been growing fast and brought unprecedented convenience, yet come along with a variety of bugs. These bugs may cause varied errors, such as abnormal functionality, missing UI elements, and failing to handle asynchronous events [1][2]. Specially, some errors can only be triggered by a specific sequence of events in web applications.

Record-replay techniques are used to faithfully reproduce the errors in web applications [3][4][5][6]. Event-based record-replay approaches [3][4] record user interactions (user events) and use them to drive the execution during replay. The memory-based record-replay approach [6] records every value loaded from memory during an execution and uses these values to replace memory loads during replay. The above approaches work well in the short execution scenarios.

Web applications are event-driven, and usually keep running for a long time. The current record-replay techniques [3][4][5][6] could generate a very long event trace. For example, Mugshot [3] could generate 75-795KB uncompressed event trace per minute, which contains nearly 3000 events. In order to diagnose an error, users have to replay and inspect all the events, which is time-consuming and exhausting.

In this paper, we focus on how to speed up the web application error reproducing. Our key observation is that most of the events in an event trace are not related to the error, and only the key events in the event trace can reproduce the error properly. In order to speed up web application error reproducing, we need to know what the key events are and irrelevant events in the event trace. In order to facilitate debugging, we need to remove the irrelevant events and make sure that the remained key events are reproducible.

The most related work with ours is EFF [7], which combines dynamic slicing and checkpoint techniques to reduce the event trace for UNIX applications. EFF builds a Dynamic Dependency Graph (DDG) and an Event Dependency Graph (EDG) to trace dependencies by statically analyzing the use-def relationship of executed statements. Their events refer to system calls rather than user interaction events. EFF is not suitable for complicated JavaScript features and DOM manipulation interfaces. Firstly, JavaScript is a prototype-based, dynamic, and weak-typing scripting language, which magnifies the difficulty to perform dynamic dependency analysis. Secondly, the APIs used for manipulating the DOM tree (tree-structure representation of a HTML page) are implemented by native code [8]. Treating these DOM APIs as general JavaScript code without knowing their semantics could miss dependencies. Thirdly, treating the whole DOM tree as a single global object is too coarse-grained and may cause many false dependencies.

In order to reduce the event trace of a web application error precisely, we abstract the JavaScript instructions and DOM manipulation instructions to precisely capture dependencies and produce a dynamic dependency graph. Based on the dynamic dependency graph, we build an event dependency graph that describes the dependency relations of events. Then, a novel event slicing approach is used to compute the key events. We also find that some key events are not always necessary for the error diagnosis. We present a more progressive event slicing approach that allows developers to make assertions for the symptoms or specify the specific part of execution instead of tracing from the whole executed code when the erroneous event is triggered. Thus, we can get a slimmer event slice of the event trace.

Our approach JSTrace is an enhanced record-replay tool that can significantly remove irrelevant events while keeping the trace reproducible. It is implemented in pure JavaScript, and enables easy integration into the client-side source code and execution on any browser. We have evaluated it on 10 real-world errors from 7 popular web applications that belong to different domains. The evaluation shows that we can efficiently remove irrelevant events to the errors.

* Corresponding author

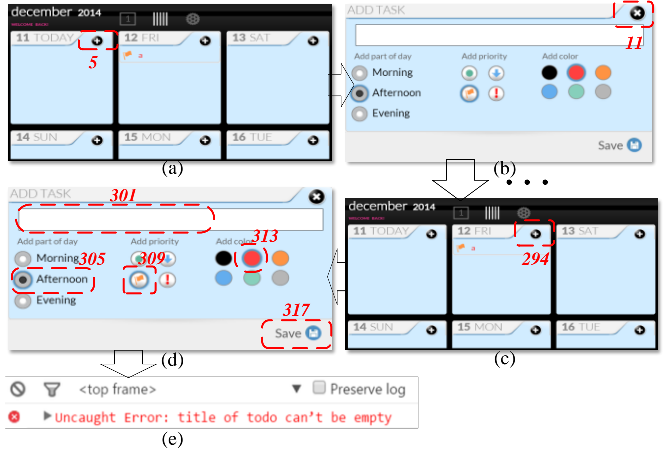


Fig. 1. Buggy application: TodoList

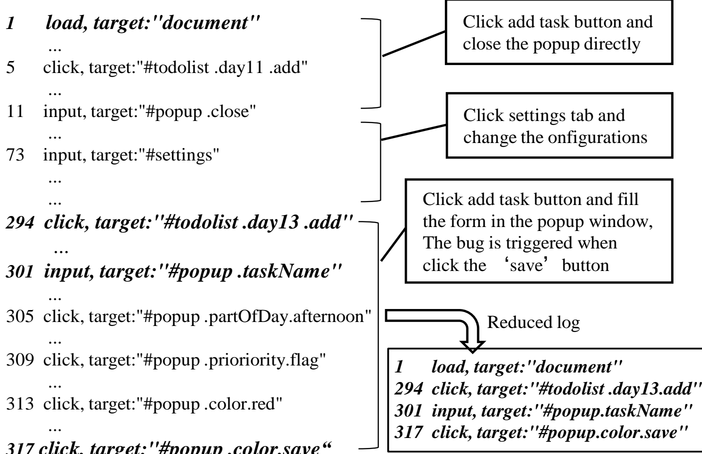


Fig. 2. Event trace reduction execution

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

- We propose a novel fine-grained approach to trace data dependencies by abstracting JavaScript and DOM instructions, and the dependency propagation rules.
- We propose an effective approach to filter out irrelevant events in an event trace. Error-directed aggressive event reduction further removes irrelevant events, and facilitates the error diagnosis.
- We have implemented our approach in the tool JSTrace. The evaluation on 10 real-world errors shows that JSTrace can remove about 96% irrelevant events, and reproduce the errors faithfully.

The remaining of this paper is organized as follows. Section II presents our motivation. Section III introduces our approach. Section IV describes JSTrace implementation. In section V we study several real-world errors and evaluate our tool on these errors. Section VI describes the study on aggressive event reduction. Section VII discusses the related work and section VIII concludes the paper.

II. Motivation Example and Overview

Fig. 1 shows the TodoList [9] web application that manages schedules in a calendar. When a user clicks the add button on a day view (Fig. 1(c)), a dialog pops up to create a to-do task (Fig. 1(d)). After the user fills up all necessary information, he/she clicks the save button. The program checks the title of the to-do task, and an error will occur if the title can be trimmed to an empty string (Fig. 1(e)). The simplified source code for TodoList can be found in Listing 1.

Fig. 2 lists the real event trace that triggers the above error. The full event trace is shown in the left part of Fig. 2. In this trace, the user clicks *add* button on the view of day 11(event 5, Fig. 1(a)), then clicks *close* button to cancel this operation (event 11, Fig. 1(b)). He/she then performs a series of operations such as changing configurations (events 73~293). Afterwards, he/she clicks *add* button on the view of day 13 (event 294, Fig. 1(c)), and types an empty string with a blank space as the title (event 301), fills in other fields in the form, and clicks *save* button (event 317), which finally triggers the serror. Replaying the entire execution with the full event trace could 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. For example, events such as clicking the *add* button and then clicking *close* button to close the popup window (events 5~11, Fig. 1(a) and Fig. 1(b)), changing the settings (events 73~293) and filling the form field (events 302~316) are irrelevant to the error. Removing these events will not affect the occurrence of the error. The key events {1,294,301,317} can faithfully reproduce this error.

In order to remove the irrelevant events, we need to identify the precise dependencies 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 easily obtain the dependencies among these DOM APIs and DOM elements. For example, the DOM API *getElementById* (line 30) suggests the data dependency between the DOM element *title* with JavaScript object *todo.title*. Without the semantics of *getElementById*, we would miss this key dependency. 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 could also access the *className* field by calling the native function *getAttribute* with the exact attribute name *class*. These inconsistent DOM APIs would make data dependency analysis 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 dependency analysis must subtly model how one DOM manipulation depends on another to avoid missing dependencies. 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

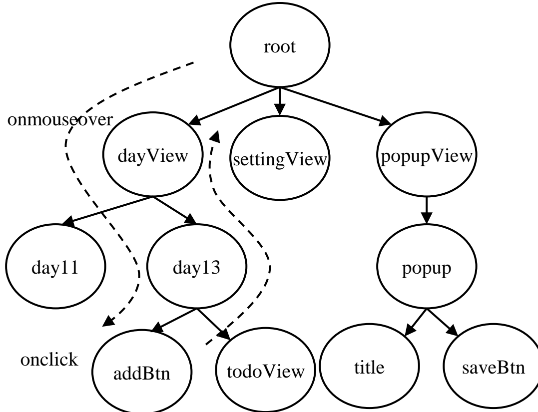


Fig. 3. DOM tree of TodoList

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 dependencies, dependency 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 feature of JavaScript and event-driven feature of web applications, it is hard to build the dependencies 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 function *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*.

In this paper, we abstract the JavaScript instructions and DOM manipulation instructions to precisely capture data dependencies. We also build the dependency between JavaScript instructions and DOM elements. Based on the dynamic dependency graph, we build an event dependency graph that describes the dependent relations of events. Our slicing algorithm operates on the event dependency graph.

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 III.A). Fig. 3 represents the corresponding DOM tree for the TodoList example, dispatching the *click* event on the DOM element *addBtn* will trigger the handler *onSave* in Listing 1. However, dispatching a *click* event on DOM element *dayView* will not trigger any handlers 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 may trace backward from the error behavior and identify the operations that affect the occurrence of this symptom. As shown in Listing 1, the error behavior is an exception thrown at line 38 that is affected by 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) that is the event handler 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 *todolist.container* variable,

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

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

13 TodoList.prototype.init = function(){
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(){
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(){
29   var todo = new TODO();   // new a TODO object
30   todo.title = this.popup.getElementbyId('title').value;
31   ...
32   if(check(todo)){
33     storage.save(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

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

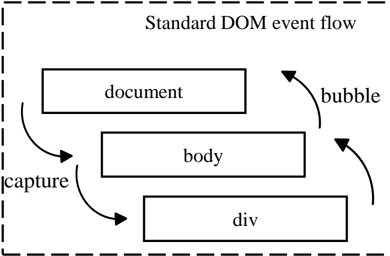
Listing 2. Example of inconsistent DOM interfaces

the DOM element *day* variable (line 15) and registers *onAdd* event handler (line 15) that will be used by event 294. Finally, we get the dependencies {317⟶[301,294], 294⟶1}. Therefore, we get the key events {1,294,301,317} and all the other irrelevant events can be removed.

III. Event Trace Reduction

In this section, we describe our approach to remove the irrelevant events. The overall process consists of two steps:

Step 1: Untriggered event analysis. If a recorded event never triggers any user-defined event handler, we can safely remove it (section A).



1) div.addEventListener('click', f); (DOM2)
2) div.onclick=function(){…} (DOM1)
3) <div onclick='…'></div> (DOM1)

Fig. 4. Standard event flow model

JavaScript abstract instructions

cons ::= *num* | *str* | *bool* | *undefined* | *null*
v ::= *object* variable

op ::= $v = cons$	// Assign a constant to v
$\mid v_1 = v_2$	// Assign variable v_2 to variable 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

Step 2: Event dependency analysis. We abstract the JavaScript operations into a series of intermediate instructions and design a dependency propagation model of web applications (section B). As we have mentioned, simply treating DOM manipulations as black box is insufficient. Thus, in order to avoid missing dependencies, we perform JavaScript dependency analysis (section B.1) and DOM dependency analysis (section B.2). Finally, we use these dependencies to build the Event Dependency Graph (EDG). We perform dynamic slicing on EDG (section B.3) to obtain the key events related to an error.

A. *Untriggered Event Analysis*

Some recorded events are fired by the user, but they never trigger any user-defined handlers registered by the user. According to the DOM3 event model [10], an event could be propagated from the root element along the tree structure to the target element, and then bubbles up to the root element. All the event handlers registered at the capture phase or bubble phase will be triggered if the event is not canceled in the middle (Fig. 4). Thus, if there do not exist an event handler registered by the user from root element to the target element, the event can be safely removed.

However, there is no DOM API that can be used to get the registered event handlers. To resolve this problem, we treat every registered handler as a special attribute of the corresponding DOM element. We keep a map for a DOM element to its corresponding event handlers. An event handler can be registered in 3 ways as Fig. 4 shows. For the first case, we override native *addEventListener* and *removeEventListener* functions to intercept the event registering or unregistering handlers. This could be done by taking advantage of JavaScript's dynamic feature. For the second and third cases, our instrumented code could intercept such operations and identify handlers registered. Thus, we can associate all the registered event handlers to their corresponding DOM elements. Based on this information, we can identify that an event will never trigger any event handler if all the DOM elements along the event flow path do not contain any event handler of the event type. In Fig. 3, a *mouseover* event on *addBtn* can trigger the event handler binding to its ancestor *dayView*, so this *mouseover* event should not be removed. A *keypress* event on *addBtn* will never trigger any event handlers, so this *keypress* event can be removed.

B. *Event Dependency 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. For these two cases, we perform JavaScript dependency analysis and DOM dependency analysis separately.

1) *JavaScript dependency analysis*

In Fig. 5, we summarize the abstract JavaScript instructions that can affect JavaScript dependency. A constant value *cons* can be a number *num*, a string *str*, and the special constant *undefined* or *null*. The variable v can represent the object in JavaScript, such as a common object, a function object. The instructions include constant assignment, variable assignment, function definition, property variable reading and writing, binary operation, unary operation and function call. Each instruction (*op*) is assigned a unique instruction id (*op.id*).

A DOM API could be either property access or function call to read or write the DOM tree. Therefore, each DOM instruction is also a JavaScript instruction, which is *Get Property*, *Put Property* or *Function Call*. In this section, we only consider the dependencies among JavaScript instructions, without considering the DOM dependencies.

JavaScript dependency analysis builds the dependencies among JavaScript instructions. If a JavaScript instruction op_1 uses an input variable that is defined or modified by another instruction op_2, we say that op_1 depends on op_2.

Table I lists all the rules for JavaScript dependency analysis. The first column shows JavaScript instructions. The second columns shows the dependency rules *JSDep*(*op*). We use *def*(v) to denote the instruction *op* that changes the variable v, and *dp*(*op*) to denote the instruction *op*'s dependencies.

For example, *def*(x)=5 denotes that an reference 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 I, *def*(y)=10, and *dp*(10)={5}. Thus, we build the dependency between instruction 10 and 5.

For an event handler, it should depend on the instruction that registers the event handler. Losing this dependency may cause mistakenly pruning the registering events and finally fail to replay. We treat each registered event handler as a property of the related DOM element. It is initialized when it is

TABLE I. JAVASCRIPT DEPENDENCY 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.
$v_1.v_2(\{v_{p1},\ldots,v_{pn}\})$	$dps(op)=\{def(v_1), def(v_1.v_2), def(v_{p1}), \ldots, def(v_{pn})\}$	Function call operation depends on object v_1, the function object v_2, and the input parameters $v_{p1}, \ldots, v_{pn}$.
$v=v_1.v_2(\{v_{p1},\ldots, v_{pn}\})$	$def(v_1)=op.id$; $dp(op)=\{def(v_1), def(v_1.v_2), def(v_{p1}), \ldots, def(v_{pn}), def(ret)\}$	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.

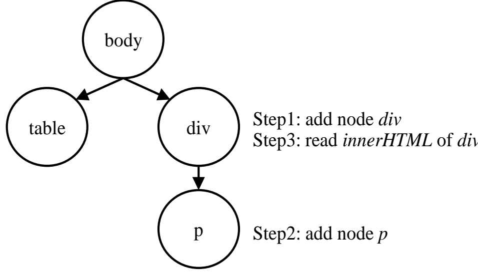


Fig. 6. Dependency example for DOM manipulations

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

registered and accessed when the event handler is triggered. So, we can resolve this in *DOM dependency analysis*.

2) DOM dependency analysis

The DOM APIs are designed for manipulating web pages by the organization W3C, which is integrated to the browser environment but not the ECMAScript standard or JavaScript itself. Since the DOM tree cannot be simply treated as a general variable, our JavaScript dependency rules are insufficient to analyze these APIs. We summarize the problems as follows:

- Inconsistent ways to modify the DOM tree. The DOM tree can be modified by assigning statement or a native function call. Dependencies may be lost without knowing the semantics of DOM APIs. As shown in our motivation example, 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*.
- We need to scan the DOM tree to make sure whether there is dependency relationship between two operations on the DOM tree. In Fig. 6, after adding a node *div* to the root body in step 1, step 2 adds a new node *p* to the node *div*. Therefore, step 2 depends step 1. Step 3 depends on step 1 because the operated DOM element *div* must already exist, and step 3 depends on step 2 because the reading *innerHTML* operation on the node *div* will get the serialization string of the subtree.

To resolve these problems, we abstract the DOM manipulations and extend the JavaScript dependency analysis to propagate DOM-specific dependencies. We summarize the DOM APIs into eight instructions in Fig. 7. DOM instructions can read, add, remove, and replace a DOM element; modifies a subtree of a DOM element; read, write, and remove an attribute of a DOM element. Each DOM instruction *dop* associates with a JavaScript instruction *op*.

DOM dependency analysis builds the dependencies 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 II presents the dependency propagation rules of DOM dependency analysis. The first column shows DOM instructions. The second column shows the DOM dependency 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 II, 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 dependencies.

To trace DOM-specific dependencies, we introduce the function *SearchDOMDep* to search for DOM-specific dependencies. Algorithm 1 presents our searching algorithm for *SearchDOMDep*. The algorithm firstly 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 get access to *ele*. Next, if the operation *dop* is reading an attribute *attr* of *ele*, then current *dop* will depends on the operations 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

TABLE II. DOM DEPENDENCY 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 clear 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*.

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 accesses 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;
```

Algorithm 2. Slicing algorithm *DS*

```
Input: g (event dependency 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:    ei = q.deQueue();
5:    if ei is not in result
6:       result.add(ei);
    //adjacentNodes can find adjacent nodes of ei in graph g
7:       for each ej in adjacentNodes(g, ei)
8:           if !result.contains(ej)
9:                q.enQueue(ej);
```

(line 5). We have manually inspected the DOM APIs according to DOM3 specification [11] to decide the DOM instruction type and whether it is necessary to search the subtree of *ele*.

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

3) DDG and EDG

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

The Dynamic Dependency Graph *DDG*(*N*, *E*) consists of a set of nodes *N* and a set of directed edges *E*. Nodes *N* are JavaScript instructions or DOM instructions, and edges *E* are the directed dependencies among instruction nodes *N*. We can trace these dependencies by JavaScript dependency analysis or DOM dependency analysis.

We can build the Event Dependency 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 there is at least one instruction in event e_i that depends on an instruction in event e_j.

The example in Fig. 8 illustrates *DDG* and *EDG* including JavaScript dependency analysis and DOM dependency analysis. The original code is part of our motivation example, which has an *onAdd* event handler and an *onSave* event handler. The dashed boxes are executed events. The solid boxes are executed instructions and the arrows between them denote dependency relationship. The rectangles denote JavaScript instructions and the rounded rectangles denote DOM instructions. Each DOM instruction is associated with its manipulating DOM element and the dependencies are retrieved by searching the DOM tree. As Fig. 8 shows, the set property *innerHTML* of *popup* at line 2 (*op4*) depends on *op2* and *op3* (the literal string) according to the *DAttrWrite* rule by resolving *JSDep*. The operation *getElementById* (line 5) implies the reading of *id* attribute of *title* (*op11*). Thus, according to the *DAttrRead* rule, *op11* depends on *op2* (by resolving *JSDep*) and *op4* (by resolving *DOMDep*). A reading property *value* of *title* element at line 5 (*op12*) depends on *op11* (*JSDep*), *op4* and *op7* (*DOMDep*) according to the *DAttrRead* rule. The registering event handler *onSave* is treated as an attribute with a unique name (*op9*), while implicitly triggering the event, an access to this event handler is recorded (*op11*), and this access depends on the registering operation. *EDG* is built based on the *DDG*. For example, the event that triggers *onSave* handler depends on the event that triggers *onAdd*, because there are edges between them. As we can see, the resulted *DDG* of combined JavaScript dependency analysis and DOM dependency analysis consists 4 kinds of dependencies in the graph, JS node to DOM node, DOM node to JS node, JS node to JS node, and DOM node to DOM node. The result of JavaScript dependency analysis or DOM dependency analysis alone is only a subgraph of this *DDG* in Fig. 8.

Calculating the key events related to the error is a graph reaching problem. Algorithm 2 gives the slicing algorithm *DS*.

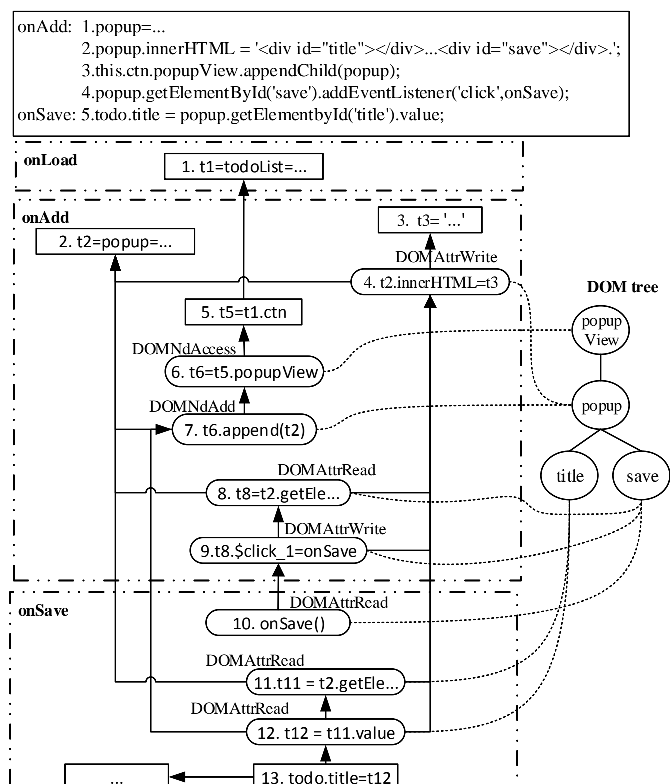


Fig. 8. Constructing DDG and EDG

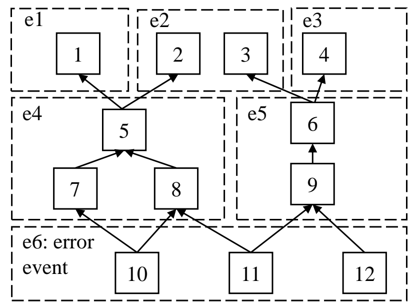


Fig. 9. An example for EDG

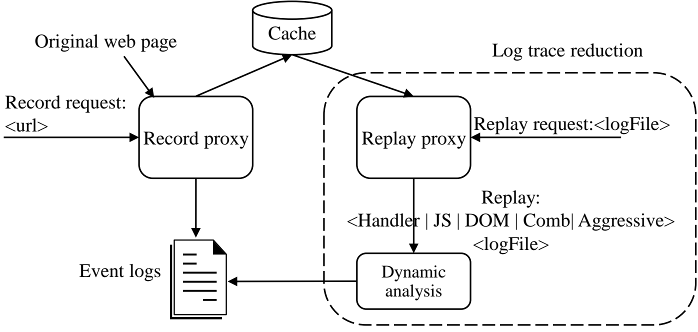


Fig. 10. The architecture of JSTrace

The algorithm is a breadth-first algorithm. 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 add it to the result set *result* (line 6), and add all the 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. As the example given in Fig. 9, we can calculate the key events as [e1, e2, e3, e4, e5, e6].

C. Replayable

We classify symptoms of web application errors into the following three cases. 1) *The rendering errors*, such as missing UI components. The symptoms of 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.* This are often functional anomalies such as the inoperative UI components caused by program logic error or the wrongly computed result. 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.

IV. IMPLEMENTATION

Our implementation of the record-replay system is similar to Mugshot [3] that provides the essential functionality to trace events and replay the event trace. We extend the replay phase to support event trace reduction as shown in the dashed box in Fig. 10. Our event trace reduction approach has taken advantage of shadow execution and instruments JavaScript code using Jalangi [6] to capture instructions and support annotated value. Our implementation is entirely in JavaScript which is transparent to users and easy to deploy.

Architecture. At the record phase, the *record proxy* retrieves the original web page and instruments it to record event trace. The event trace is periodically updated to the server and stored in a log file. It supports manually submitting of the error by users or automatically submitting when an exception is thrown. The *Cache* is used to store web pages and nondeterministic data, which makes sure that the replaying phase will get the same data at the recording phase. The *replay proxy* is designed to replay a given event trace. *Dynamic analysis* module can perform event trace reduction at several levels by separately applying untriggered event handler analysis, JavaScript dependency analysis, combination analysis (combination of JavaScript and DOM dependency analysis), and aggressive analysis (section VI) accordingly.

Instrumentation. The instrumentation is patched to the client side code, thus the dynamically generated code can be instrumented as well, such as *eval*, *setTimeout* and *setTimeinterval* that can run or schedule a task to execute a piece of string code. All the JavaScript code in libraries is instrumented. All the concerned JavaScript instructions and DOM instructions are intercepted by the instrumentation.

JavaScript dependency analysis. To trace data dependencies dynamically, we incorporate the idea of shadow execution [6] 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

TABLE IV. EVALUATED ERRORS

Apps	BugId	IssueID	#All	#Handler/R	#JS/R	#Comb_L1/R	#Comb_L2/R	#Comb_L3/R	#Expected	Rate
Chart.js	1	503	1139	351/Y	52/N	342/Y	79/Y	74/Y	6	94.0%
	2	920	1168	770/Y	139/Y	757/Y	139/Y	139/Y	6	88.60%
HandsonTable	3	1366	403	345/Y	29/Y	344/Y	297/Y	29/Y	5	94.0%
	4	638	694	668/Y	28/Y	388/Y	31/Y	28/Y	3	96.4%
	5	2231	606	462/Y	34/Y	67/Y	51/Y	34/Y	6	95.0%
JPushMenu	6	1	342	6/Y	2/Y	6/Y	6/Y	2/Y	2	100.0%
TodoList	7		1410	851/Y	21/N	53/Y	43/Y	24/Y	3	98.5%
FullPage	8	146	398	39/Y	30/Y	39/Y	36/Y	30/Y	3	93.2%
Editor.md	9	18	1023	791/Y	9/Y	97/Y	72/Y	9/Y	2	99.3%
My-mind	10	12	1454	351/Y	8/Y	40/Y	22/Y	8/Y	6	99.9%

TABLE III. REAL-WORLD APPLICATIONS

Apps	Description	JS size	Popularity
Chart.js [12]	basic charts	105K	14803
Handsontable [13]	Excel-like data grid editor	4.7M	4989
TodoList [9]	Offline calendar	312K	19
JPushMenu [14]	A menu library	1.5M	134
FullPage [15]	To create full screen scrolling websites	882K	9518
Editor.md [16]	A markdown editor	257K	530
My-mind [17]	Online mindmapping software	223K	1449

each value taking advantage of 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 is modified, the shadow value is updated with the id of the operating instruction.

DOM dependency analysis. We regard a JavaScript instruction as a DOM instruction if the operating object or returned value is DOM element. We manually inspect the APIs to decide the information that are used by *SearchDOMDep.*

Limitations. Arrays are treated as an integrity to avoid missing dependencies. However, this may cause unexpected event dependencies. The functions *setTimeinterval* and *setTimeout* that periodically execute a scheduled task may cause iterative event dependencies, and greatly lower the reduction rate of our approach.

V. EVALUATION

We evaluated JSTrace on 10 real-world errors from 7 different open source web applications in GitHub. In order to select the representative applications, we first used the condition “language:JavaScript tag:bug comment:>2” to filter the applications that are written in JavaScript and the issues are marked as bugs with multiple comments. We used the keywords *repro*, *sample* or *bug* to filter the errors that may have reproducing steps. From the result, we made further study on the errors, and made sure that they can be manually reproduced and has certain difficulty to diagnose (with multiple steps to reproduce). Finally, we classify them to different categories and bias the applications that are more popular, weighed by the number of stargazers in GitHub. We have selected 10 real-world errors form all the filtered issues.

Table III provides an overview of our evaluated web applications. These web applications are designed for different purpose and functionality. For example chart.js [12] is a drawing library that uses canvas, Handsontable [13] is an excel-like application, and TodoList is an offline HTML5 application that works like a calendar. These web applications are very complicated, and all of them use many JavaScript. For example, Handsontable uses 4.7M JavaScript code.

In the evaluation, we measure reproducibility and efficiency of our tool. Specifically, we investigate the following research questions:

RQ1: Can the reduced *event trace faithfully reproduce the errors?*

RQ2: Can our tool efficiently reduce the event trace?

RQ3: Is the performance acceptable?

As far as we know, there is no similar work that aims to reduce event trace for web applications. Thus, we have not compared our work to others.

A. RQ1: Reproducibility

To address the first research question, we evaluated whether the reduced trace can faithfully reproduce the errors.

Table IV shows the result on 10 errors. The column *issueID* shows the issue id in GitHub. The column *ALL* shows the number of events in the event trace. The columns *Handler*, *JS* show the reduced result for untriggered event analysis, and JavaScript dependency analysis. *Comb_L** is the combination of JavaScript dependency analysis and DOM dependency analysis but with different granularity of DOM analysis. *Comb_L1* simply treats the DOM tree as an integrity variable, *Comb_L2* further manually identifies whether the operation is writing or reading and then simply build DOM dependencies by the read-after-write rule. While *Comb_L3* uses our fine-grained DOM analysis (section III.2). The *R* flag with value *Y* or *N* represents whether the reduced trace can successfully reproduce the errors or not.

The untriggered event analysis may significantly reduce event trace. However, the remained event trace is still quite long to diagnose. For example, the bug 2 and bug 9 remove only less than a half irrelevant events, but the expected event number is no more than 6.

JS analysis and *Comb* analysis are applied to further reduce the event trace. However the result of *JS* analysis may not be reproducible. For *JS* analysis, 2 out of 10 errors failed to replay. The *Comb_L** performs best with respect to reproducibility, all

```
<a href="#" id="1" class="selected">
    <span class="icon">
        <img alt="" src="images/radio-sel.png">
    </span>
    <span class="text afternoon">Afternoon</span>
</a>
```

Fig. 11. HTML code snippet for TodoList

the errors are successfully replayed. The result of fine-grained *Comb_L3* analysis is closer to the expected event traces.

The reason why the reduced event trace of *JS* analysis failed to replay is the missing DOM dependencies. The failing number is not high because our evaluated applications are highly-cohesive, and the applications have much shared data and JS dependencies are very common. This result also explains why the *DOM analysis* is necessary. Bug 1 is a division by zero error triggered when the user add data to a line chart right after clearing the chart. The *JS* analysis failed to trace the DOM dependency while executing *add data*. Fig. 11 shows how a DOM dependency is missing. It uses an *img* to render a checkbox and its value is represented by the *class* attribute of its ancestor DOM element. That means that reading the value will search for modification of its ancestor elements. Without our *DOM analysis*, the dependency cannot be established.

B. RQ2: Efficiency

We use reduction rate to evaluate the efficiency. Table IV presents the result. The column *Expected* is the number of the actual key events to reproduce the error. The column *Rate* is the reduction rate that calculated by (*#ALL-#Comb_L3*) / (*#ALL-#Expected*).

In Table IV, the average reduction rate is 96%. Untriggered event analysis may significantly reduce event trace such as bugs 1, 6, 8 and 10 that have sparsely registered handlers and rarely use event delegation programming pattern, but the other bugs only remove few irrelevant events. *Comb_L3* (with reduction rate of 87% relative to the untriggered event analysis) can remove more irrelevant events compared to the coarse-grained *Comb_L1* analysis (40%) and *Comb_L2* (65%). This result proves the necessary of our fine-grained DOM analysis.

However, the result of fine-grained analysis contains irrelevant events. We dig into the errors and find that the unremoved irrelevant events are mostly caused by: 1) *Array operation*, since we treat array as an integrity object. The application char.js and HandsonTable suffer from this problem. 2) *Calls to setTimeout and setTimeInterval*. Such calls are often used to periodically check and modify a shared data or showing animations which result in large amount of dependencies and lower the reduction rate. The application char.js and fullPage suffer from this problem. 3) *Redundant data dependencies*. For example, an event reads the value of variable *guid* set by the handler of a previous event e_i and adds it by 1. There is a JavaScript dependency between them. However, the error may not care about the exact value of *guid* and the user cannot feel the existence of it. Thus, the event e_i is not expected to appear in the result from the user side. Almost all of the applications suffer from this problem. We try to solve this issue with the aggressive event reduction in section VI.

TABLE V. MEMORY OVERHEAD

BugId	Original (MB)	Comb_L3 analysis (MB)	Overhead (X)
1	4.5	12.2	2.71
2	4.7	30.5	6.49
3	10.3	64.1	6.22
4	7.5	74.4	9.92
5	23.4	45.3	1.94
6	12.5	21.3	1.70
7	12.4	161	12.98
8	4.8	64	13.33
9	11.2	144	12.90
10	11.6	16.3	1.41

TABLE VI. AGGRESSIVE EVENT REDUCTION

App	BugId	#Comb_L3/R	#Aggressive/R	#Expected
Chart.js	1	74/Y	4/N	6
	2	139/Y	28/Y	6
HandsonTable	3	29/Y	24/N	5
	4	28/Y	9/N	3
	5	34/Y	13/N	6
JPushMenu	6	2/Y	2/Y	2
Todolist	7	24/Y	5/Y	3
FullPage	8	30/Y	21/Y	3
Editor.md	9	9/Y	2/Y	2
My-mind	10	8/Y	7/Y	6

C. RQ3: Performance

The performance of the record and replay (without analyzing) is the same as Mugshot [3]. Since our analyzing is performed offline on the server side, we assume that the slowdown of analyzing is acceptable. In this section, we evaluated the memory usage.

We have evaluated the memory usage of the 10 errors on Google Chrome and taken a heap snapshot on the profiles tab of the developer tool when the execution is ended. We use 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 V compares the memory usage of the original and the *Comb_L3* analysis. 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.

D. Discussion

A threat to our evaluation is that the 10 errors come from only 7 applications. However, the applications considered are randomly selected from real-world open source projects. 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. Finally, the applications are developed for different purposes.

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 to replay an event trace that 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 [3]. Therefore, from the point of reproducibility, there is no need to trace events across pages. Each execution of one page will generate one event trace.

VI. CASE STUDY FOR AGGRESSIVE EVENT REDUCTION

As we have mentioned in our evaluation, almost all the applications have dependencies that cannot be removed. In this section, we setup a study to make further event reduction.

Our insight is that developers may not concern all the instructions triggered by an erroneous event, but a subset. If we only trace from a small part of instructions, we may gain a slimmer event trace. We call this as an a*ggressive event analysis*. To calculate this slice, we first give the expected piece of code to replay, and use the execution of this code as the reproducible criterion. Then we trace from the variables that are involved in the given piece of code and find the event set that directly affect these variables, this event set is called *event cut*. Finally we use this event cut to run our *DS* algorithm and get the final event trace. For the example shown in Fig. 9, according to the *EDG*, we can get the final event set [e1, e2, e3, e4, e5, e6]. If the error is triggered by the execution marked as 12 and we only concern the reproducing of that execution, then we do not need to keep track on the execution of 10 and 11. Based on the *EDG*, we can get the final event set as [e2, e3, e5, e6].

This approach is similar to EFF's *meta slicing* [7], EFF has proved its correctness. However, in the case of web applications, events often refer to user interaction events that are associated with multiple event handlers. Thus, the result of our *aggressive event reduction analysis* cannot guarantee the reproducibility, but is enough to diagnose the expected code.

Experiment. We perform the aggressive event reduction analysis for all the 10 errors. We compare with *Comb_L3 analysis*, since the *Comb_L3 analysis* has high reproducibility but with many irrelevant events.

Table VI shows the result of aggressive event reduction analysis. Comparing to *Comb_L3 analysis*, the reduction result is closer to the expected result, but 4 errors cannot be reproduced. This aggressive event reduction analysis requires that the user is familiar with the source code. Thus, this analysis is more helpful for developers to diagnose an error.

VII. RELATED WORK

Record and replay in web applications. Mugshot [3] is a high performance record-replay system that captures all events in JavaScript applications and all the nondeterministic information such as AJAX request, random calls and timers to make sure the replaying phase loads the same data. DoDOM [19] records user interaction events, then the web application can be repeatedly executed under the captured event sequence. WaRR [20] records user interactions with a web application and uses the recorded interaction trace to perform high-fidelity replay of the web application. Ripley [21] replicates execution of the client-side JavaScript application on the server replica to automatically preserve the integrity of a distributed computation. Reducing event trace are out of their scope. Our basic record-replay approach applies the same technique as Mugshot [3].

Dynamic slicing. Dynamic slicing [22][23][24] 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 dependency graph [7][25]. Our work differs from their work for DOM-specific challenges and in the way the graph is constructed. We combine the dynamic slicing technique and shadow execution technique to trace JavaScript programs. Thus, we can avoid resolving all the complex dynamic features of JavaScript.

Event-based application debugging and testing. CLEMATIS [5] records all the event-based interactions and execution context information such as DOM events, timeouts and function traces, to analyze the dynamic behavior of web applications. EFF [7] combines dynamic slicing technique and checkpoint technique 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 dependencies is not suitable for web applications. Josip's work [26][27] extract client-side web application code for the purpose of program understanding, debugging and feature extraction. They use the expected behavior as slicing criteria and perform dynamic program slicing to identify the related CSS, HTML, and JavaScript source. However, their work does not care which event the execution is performed, their approach of capturing dependencies is inefficient to resolve our challenges. They use parent-child relation to form structural dependency edges between DOM nodes and use the parsed AST to build dependencies between JavaScript statements. AppDoctor [28] 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 rules to build and propagate dependencies. The work [29] aims to reduce event traces using delta debugging technique that treats the operations as black boxes. The approach relies on trial and error to decide which inputs to discard, not the data dependency analysis.

VIII. CONCLUSION

In this paper, we propose a tool JSTrace to identify the key events related to a web application error. Given the expected symptoms or code snippet that we should reproduce, we precisely trace the dependencies between JavaScript instructions and DOM instructions, and develop a novel dynamic slicing to filter out irrelevant events. The evaluation on real-world web application errors shows that JSTrace can greatly reduce the event trace, and achieve 100% reproducibility.

IX. ACKNOWLEDGMENTS

The work was supported in part by National Natural Science Foundation (61173005) of China.

REFERENCES

[1] S. Thummalapenta, D. Pranavadatta, S. Sinha, S. Chandra, S. Gnanasundaram, D. D Nagaraj and S. Sathishkumar, “Efficient and Change-resilient Test Automation: An Industrial Case Study,” in *Proceedings of the International Conference on Software Engineering (ICSE), 2013*, pp. 1002–1011.

[2] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, An Empirical Study of Client-Side JavaScript Bugs. in *Proceedings of ACM / IEEE International Symposium on Empirical Software Engineering and Measurement* (*ESEM*), 2013, pp.55-64

[3] J. Mickens, J. Elson, and J. Howell, “Mugshot : Deterministic Capture and Replay for JavaScript Applications,” in *Proceedings of the 7th USENIX conference on Networked systems design and implementation (NSDI)*, 2010, pp. 159-174.

[4] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst, “Interactive Record/replay for Web Application Debugging,” in *Proceedings of the 26th annual ACM symposium on User interface software and technology (UIST), 2013*, pp. 473–484.

[5] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman, “Understanding JavaScript Event-Based Interactions,” in *Proceedings of the ACM/IEEE International Conference on Software Engineering* (*ICSE*), 2014, pp. 367–377.

[6] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, “Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript,” in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*FSE*), 2013, pp. 488-498.

[7] X. Zhang, S. Tallam, and R. Gupta, “Dynamic Slicing Long Running Programs Through Execution Fast Forwarding,” *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*FSE*), 2006, pp. 81–91.

[8] “JavaScript.” [Online]. Available: http://en.wikipedia.org/wiki/JavaScript.

[9] “TodoList.” [Online]. Available: https://github.com/01org/webapps-todo-list.

[10] “Document Object Model (DOM) Level 3 Events Specification.” [Online]. Available: http://www.w3.org/TR/2011/WD-DOM-Level-3-Events-20110531/.

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

[12] “Chart.js.” [Online]. Available: https://github.com/nnnick/Chart.js.

[13] “handsontable.” [Online]. Available: https://github.com/handsontable/handsontable.

[14] “jPushMenu.” [Online]. Available: https://github.com/takien/jPushMenu.

[15] “fullPage.js.” [Online]. Available: https://github.com/alvarotrigo/fullPage.js.

[16] “editor.md.” [Online]. Available: https://github.com/pandao/editor.md.

[17] “my-mind.” [Online]. Available: https://github.com/ondras/my-mind.

[18] G. Li, E. Andreasen, and I. Ghosh, “SymJS: Automatic Symbolic Testing of JavaScript Web Applications,” in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2014,* pp. 449–459.

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

[20] S. Andrica and G. Candea, “WaRR: A Tool for High-Fidelity Web Application Record and Replay,” in *IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, 2011, pp. 403–410.

[21] K. Vikram, A. Prateek, and B. Livshits, “Ripley: Automatically Securing Web 2.0 Applications Through Replicated Execution,” in *Proceedings of the 16th ACM Conference on Computer and Communications Security* (*CCS*), 2009, pp. 173–186.

[22] M. Weiser, Program Slicing. in *Proceedings of the 5th International Conference on Software Engineering (ICSE)*,1981, pp.439-449.

[23] H. Agrawal and J. R. Horgan, “Dynamic Program Slicing,” *ACM SIGPLAN Notices*, vol. 25, no. 6, pp. 246–256, 1990.

[24] a. De Lucia, “Program slicing: Methods and Applications,” in *Proceedings of First IEEE International Workshop on Source Code Analysis and Manipulation*, 2001, pp. 142-149

[25] K. H. Lee, Y. Zheng, N. Sumner, and X. Zhang, “Toward Generating Reducible Replay Logs,” in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, USA, 2011, pp. 246–257.

[26] J. Maras, J. Carlson, and I. Crnkovi, “Extracting Client-side Web Application Code,” in *Proceedings of the 21st International Conference on World Wide Web* (*WWW*), 2012, pp. 819–828.

[27] J. Maras, M. Stula, J. Carlson, and I. Crnkovic, “Identifying Code of Individual Features in Client-Side Web Applications,” *IEEE Trans. Softw. Eng.*, vol. 39, no. 12, pp. 1680–1697, Dec. 2013.

[28] G. Hu, X. Yuan, Y. Tang, and J. Yang, “Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor,” in *Proceedings of the Ninth European Conference on Computer Systems* (*EuroSys*), 2014, pp. 18:1–18:15.

[29] 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 the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2015, pp. 333–344.