Detect Cross-Browser Issues in JavaScript-based Web Applications based on Record/Replay

Guoquan Wu, MeiMei He
State key Lab, Institute of Software, Chinese Academy of Sciences
Email: {gqwu,hemeimei13}@otcaix.iscas.ac.cn

***Abstract*—With the advent of Web 2.0 application, and the increasing number of browsers and platforms on which the applications can be executed, cross-browser incompatibilities(XBIs) are becoming a serious problem for organizations to develop web-based software. Although some techniques and tools have been proposed to identify XBIs, they cannot assure the same execution as only explicit user activity is considered while ignoring "behind-the-scene" activity like the firing of timer and random number generation, and thus generate many false positives/negatives. To address this limitation, we developed X-Check, a novel cross-browser testing approach and tool, which combines crawling and record/replay technique, and supports automated XBIs detection with high accuracy. To improve the efficiency of XBI detection, X-Check also designed an incremental detection algorithm by only checking mutated and layout-changed DOM nodes. Our empirical evaluation shows that X-Check is effective and efficient, improves the state of the art, and can provide useful support to developers for diagnosis and (eventually) elimination of XBIs.**

I. Introduction

Highly interactive web applications that offer user experience and responsive of standard desktop applications are becoming increasingly popular these days. Applications such as Yahoo! Mail and Google Docs, now have enjoyed wide adoption and pervade all aspects of human activities. Unlike traditional web applications which perform the majority of their computation on the server, modern web applications have heavy client-side behavior footprints which need to be interpreted and executed in the browser.

Web applications are expected to behave consistently across all of the popular browsers and platforms. However, it is well known that different web browsers render web content somewhat differently. These inconsistencies lead to what we call cross-browser incompatibilities (XBIs) - differences in the way a web page looks and behaves in different environments[21].

Because of the increasing importance of XBIs, a number of tools and techniques have been proposed to address them. There are over 30 tools and services for cross-browser testing currently in the market. Such problem also gets the attention from the academia, which aims to propose automated techniques for XBIs detection. According to the work[21], XBIs can be summarized as three main types: behavior, structure and content.

- *Behavior XBI* involves the difference in the behavior of the individual functional components within a page. One such example would be a button that performs some action within one browser and a different action, or no action at all in another browser.
- *Structure XBI* refers to the difference in the layout of the page. For example, two buttons in the pages are arranged horizontally (left to right) in one browser, but vertically in another browser.
- *Content XBI* refers to the difference in the content of individual components of the web page. It can be further classified as text-content XBI and visual-content XBI. The former involves the difference in the text value of an element, whereas the latter refers to the difference in the visual aspect of a single element (e.g., page title has shadow in FF and no shadow in IE).

Existing work mainly focuses on designing specific technique for each class of XBIs. For example, to identify behavior XBIs, Mesbah et al.[17] adopted crawling technique to explore the web application under different browser environments and compare the extracted models for equivalence. Based on this work, Shauvik et al.[21] focused on detecting structure XBIs besides behavior XBIs and content XBIs. Alignment Graph is proposed to represent the relative-layout of the elements within a page, and then checked for equivalence to identify structure XBIs.

Although existing tools and techniques provided encouraging results, there are still some limitations. To detect XBIs, most work needs to first extract the state graph of the given application under different browser environments, and then detect behavior XBI by comparing state graphs, check structure/content XBI based on the collected page data (e.g., screenshot and layout information). However, crawling technique only considers user activities, while ignoring other sources of non-determinism (e.g.,timer, Ajax request) inside the browser, which can not assure the same execution when the application is crawled under different browser environments. Therefore, existing techniques may lead to some false positives, or miss some XBIs. The work done by Gao et.al [16] also illustrates a common set of factors (including time delay for Web application) will impact the determinism in the test outputs.

To address the limitation of existing techniques, this paper presents X-Check, a novel cross-browser testing technique and tool, which leverages crawling technique to automatically explore the state space of the web application (running in the reference platform), and at the same time captures various non-deterministic events. By replaying captured event traces in different environments (including reference and test platforms) and collecting page data during replay, X-Check supports to detect XBIs with very high accuracy. Record/replay technique

also provides a useful support for XBIs diagnosis, as the trace (which contains XBIs) can be deterministically replayed.

However, existing record/replay techniques usually assume that the application runs in the same environment (to reproduce a past execution faithfully). Captured events are sequentially replayed without needing to check whether the replayed event is the same one captured during record. Direct replaying captured log in a different platform may cause fragile or even wrong replay (the behavior diverges from the original's) as an error caused by incompatible JavaScript API invocation may affect the replay of the remaining events. To address this issue, X-Check introduces a novel cross-browser record/replay technique, which can identify the differences between the captured log and the replayed execution timely and reproduce a previous execution as faithfully as possible in a new environment.

Secondly, notice that for modern Web 2.0 application, Ajax technologies are widely used to communicate with the server asynchronously without reloading the whole page, which usually only incurs partial change of the page. By computing mutated and layout-changed DOM nodes resulting from replaying each event, X-Check designed a new incremental algorithm to detect XBIs, which improves the efficiency in XBIs detection by avoiding checking unchanged DOM nodes repeatedly.

The main contributions of this work are:

- We proposed X-Check, a novel XBIs detection approach by combination of crawling and record/replay technique, which not only supports to detect various XBIs automatically, but also improves the accuracy of XBIs detection. Specifically, by adapting existing record/replay technique, X-Check designed a new cross-browser record/replay technique, which can reproduce a previous execution as faithfully as possible under different browser environments by avoiding an error caused by incompatible API invocation to influence the replay of other events;
- We designed a new incremental XBIs detection algorithm. By only checking mutated and layout-changed DOM nodes resulting from each event, X-Check improves the efficiency in detecting XBIs;
- An implementation of our approach and a thorough empirical study whose results show that X-Check is effective and efficient in detecting XBIs, improves the state of the art, and can support developers in diagnosing and (eventually) eliminating XBIs;

II. Motivation Example

Modern web applications are largely event-driven. Their client-side execution is normally initiated in response to the following types of events: DOM event (e.g., click, key up), timing event, or asynchronous callback message from the server. In this section, we introduce a simple web application, referred to as Book Manager, which is developed for an organization to manage borrowed books.

Fig.1 shows the main page after user logins successfully. The navigation bar on the left shows the category of books and the tab widget on the right shows the detailed book

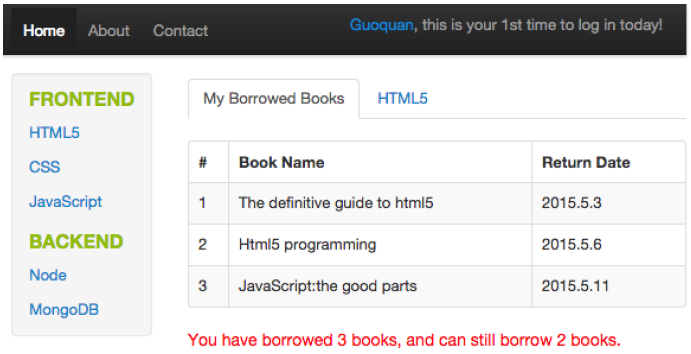

Fig. 1. Main Page in Chrome

information. When user clicks one item of the navigation bar, a new tab label will be added to the tab widget if it does not exist before. The first tab label lists the books that are borrowed by current user.

In the following, we will use this application as the motivating example to illustrate some issues that are not well addressed by existing techniques.

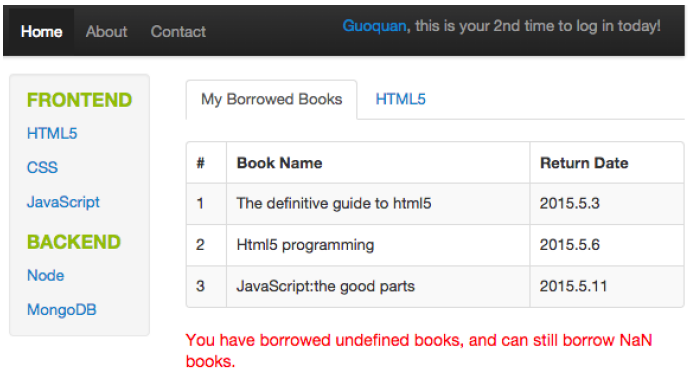

Fig. 2. Main Page in IE

Problem 1: Accuracy of XBI detection. Because of dynamic and non-deterministic behavior of JavaScript-based Web applications, even the same page will be rendered slight differently when it runs twice in the same browser. Existing XBIs detection techniques are prone to generate some false positives or false negatives as they only capture explicit user activities while ignoring other sources of non-determinism inside the browser (e.g., Ajax event, timing event).

Fig.2 shows the rendered main page when the application runs in the browser IE. Compared with Fig.1, it can be seen that there are two differences: one is login times and the other is the reminder about the number of books that user still can borrow. Fig.3 shows the sample JavaScript code related to reminder (code related to login times is not shown for space limitation). When `document` is ready, `setTimeout` is invoked (line 4) to register a callback function `showReminder`, which will be triggered 1000ms later to check whether the borrowed books have been shown in the page. If they already exist, function `reminder` will get the number of borrowed books by using property `childElementCount` to query the size of the list, compute the remaining books that the user still can borrow, and then show the reminder

information. After 1500ms, the reminder will disappear. As `childElementCount` property is not supported in IE, the query will return 'undefined', which results in the observed error shown in the bottom of Fig.2.

```
$(document).ready(
 ...
 setTimeout(remind,1000);
);

function remind(){
 var list=document.getElementById('booklist');
 if(list){
  var count=list.childElementCount;
  var borrowed='You borrowed '+count+' books'
      +', and can still borrow '+(maxNum-count)+' books';
  var text = document.createTextNode(borrowed);
  var reminder=document.getElementById('hint');
  reminder.appendChild(text);
  setTimeout(function(){$('#hint').fadeOut()},1500);
 }else{
  setTimeout(remind,1000);
 }}
}
```

Fig. 3. JavaScript Code

It's obvious that only reminder information is the real XBI. However, existing technique will identify login times as a content XBI as the number is different when the application runs in Chrome and IE, and thus has one false positive. XBI about the reminder information will be missed as timing event is ignored. Therefore, appearing and disappearing of the reminder as a result of the execution of the callback of timing event cannot be captured and lead to one false negative.

Problem 2: Duplicate XBI Detection. To identify content and structure XBIs, existing work extracts the whole DOM nodes and their layout information once page is detected to change. The limitation of this approach is that some nodes will be checked mutilple times even if they remain unchanged after an event is triggered. For modern JavaScript-based web applications, in order to improve its responsiveness and user experience, usually only parts of the page will be refreshed resulting from each event. For example, in Fig.1, when user clicks different items in the navigation bar, only tab widget will respond to this action. Navigation bar will not change during this time. If only changed DOM nodes are checked, we can improve the efficiency of XBIs detection greatly by avoiding checking unchanged DOM nodes repeatedly.

III. Approach

In this section, we describe our approach for addressing the issues mentioned in the motivation example. Fig.4 shows the overview of the proposed approach, which consists of the following main steps:

- To collect the event traces automatically which can be replayed in different platforms later, our technique first crawls the states of the web application running in the reference platform. Before exploration, recorder is plugged into the page and will capture various non-deterministic events when the application is crawled.
- Secondly, the collected traces are replayed remotely in the test and reference platforms through a remote agent. To reproduce the execution (captured from reference platform) as faithfully as possible in test platforms, X-Check proposed a novel cross-browser record/replay technique, which can identify the behaivor issue timely when the execution diverges from the original's. After event is replayed, data about current page, such as DOM nodes, layouts and screenshots are extracted.
- Based on the collected data, an incremental XBIs detection algorithm is designed to identify different types of XBIs by computing mutated and layout-changed DOM nodes between two events, and generates visual error reports to developers finally.

The following sections present further details of these phases:

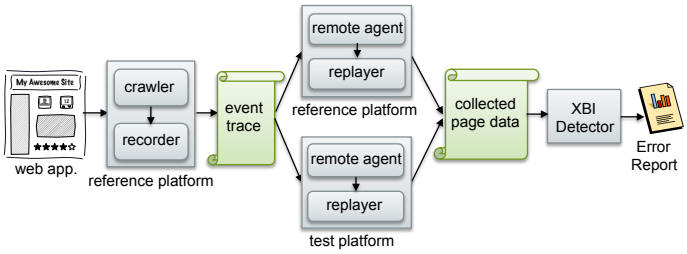

Fig. 4. Approach Overview

A. trace collection

To detect XBIs automatically, similar to existing techniques, X-Check first involves a crawler to explore the state space of the web application. The difference is that our technique only explores web application in the reference platform. During the exploration, recorder will be responsible to capture non-deterministic events. However, direct replaying captured events will fail as the crawler needs to backtrack to previous state (to explore new state) during the exploration and such 'backtrack' action cannot be captured by recorder.

Note that, to crawl Ajax-based Web applications (which usually cannot backtrack to previous state by clicking browser's *back* button), crawler will save an *event path* (starting from index page), through which a previously explored state can be arrivied. Therefore, our approach splits the captured logs into multiple sessions/traces. A new session starts each time the crawler returns to index page to explore new state.

B. cross-browser record/replay

Despite an abundance of research on record/replay systems, existing works usually assume that the captured logs will be replayed in the same platform to reproduce a previous execution faithfully. As the environment is the same, during replay, these techniques can ensure that the captured events will be replayed correctly. However, when the captured logs are replayed under a different browser environment, an error caused by incompatible JavaScript API (between two different browsers) may cause replay to diverge from the recorded execution. To identify such behavior deviation and prevent the error from affecting correct schedule of remaining events, X-Check designs a novel cross-browser record/replay technique.

In the following, we first introduce how various sources of non-determinism are recorded/replayed by leverage the technique proposed in Mugshot[19], and then discuss the problem when captured logs are directly replayed in different platforms. We also give our approach to adapt existing record/replay technique to support cross-browser record/replay.

Record/Replay Non-deterministic Events. The sources of non-determinism in client-side JavaScript application mainly include the following four types:

DOM events. DOM events (e.g., mouse events, keyboard events) allow JavaScript to register different event handlers on nodes inside DOM tree. To capture DOM events, recorder mainly leverages DOM level 2 model [3], which defines a three-phase event dispatch model (capturing-target-bubbling). In this model, capturing handler registered in ancestor node will be executed before any handlers registered on child's node. As `window` object is the highest ancestor in the DOM event hierarchy, recorder registers a capturing handler on `window` object for each type of DOM events to log the event. For IE with version $<$9.0, as the event model does not support capturing phase, event is logged by registering bubbling handler on `window`. To avoid an event to be cancelled before it reaches `window`-level handler, the recorder overrides `cancelBubble` setter method of class prototype `Event` object to log the event before its cancellation. Replaying DOM events is simple, and replayer just needs to locate the target element, and then dispatch a synthetic event.

Timing events. JavaScript `setTimeout`/`setInterval` allows the application to register a callback function, which will be invoked once (for `setTimeout`) or repeatedly (for `setInterval`). Recorder can capture timing event by wrapping `setTimeout` and `setInterval` function. The wrapped registration functions take an application provided callback, wrap it with the logging code, and then register wrapped callback with native timer. Each callback is also assigned with a unique id, which will be logged to indicate this callback is executed when the timer is triggered later.

To replay timing events, `setTimeout`/`setInterval` are overridden to replace the native one. The overridden versions tag each application-provided callback with an ID and add the callback to a global function cache. This cache is built in the same order that IDs were assigned at logging time. Therefore, when timing event is scheduled, replayer simply retrieves appropriate function from the cache and executes it.

Ajax events. Ajax request, represented as `XMLHttpRequest` (*XHR*) object in the browser, is used to exchange data asynchronously with the server. To capture an Ajax event, recorder interposes on object's `send` method to wrap the application handler with the logging code to log current state of the request. To log response, recorder uses `setter` and `getter` to interpose on assignments to `responseXML`/`responseText` properties. To replay Ajax event, *XHR* object is overriden, which first updates the state and response with the log data before invoking the callback.

Non-deterministic function calls. Applications call `Date()` to get current time and `Math.random()` to generate a random number. To capture these two non-deterministic function calls, recorder wraps the original constructor to log the result. For the replay, `Date()` and `random()` constructors will be overriden to get results directly from the log.

Besides above non-deterministic events, X-Check also supports to record/replay web applications with multiple web pages (which is not supported in Mugshot). Each time the crawler starts to explore states from index page, recorder starts a new session by assigning it a unique session id and treat all the operations in the session as a complete transaction. For each crawled page, a sequence id (initialized with 0) is used to represent this page. All the events recorded in current page are matched to this sequence id. Session id and sequence id are stored persistently into `sessionStorage` to keep a complete transaction. X-Check also utilizes `beforeunload` event, which is called just before the browser navigates away from the current page, to increase sequence id, and then capture events occurred in the new page. To support replay, `beforeunload` event is triggered to increase the sequence id before the application navigates to a new page, and event logs that corresponds to this new page will be retrieved to continue to replay.

However, record/replay technique introduced above cannot ensure that the captured events will be correctly scheduled when the application runs under a different browser during replay. Consider the segment of JavaScript codes shown in Fig.5: $elem0$ and $elem1$ are registered with two click events respectively. After each element is clicked, a timer will be registered through `setTimeout` function. When timing event is triggered, the callback will call `Math.random()` to generate a random number. Fig.6 shows two possible captured logs when the above code segment runs in Chrome. Firstly, $elem0$ is clicked and a timer (callback id is 0) is registered. If $elem1$ is clicked before the timing event is triggered, captured log will be shown as trace 1. If $elem1$ is clicked after the timing event was triggered, captured log will be shown as trace 2.

It is obvious that in Chrome, both captured logs will be replayed successfully. However, when the application runs in IE during replay, as `firstElementChild` API is not supported in IE, el is null and an exception will be thrown in line 4 when get its inner HTML, which causes the remaining events can not be correctly scheduled. For trace 1, when the first timing event is scheduled, as it is not registered beforehand, it will execute the callback $fn1$ registered by the second timer, which then consumes the wrong random number (generated by $fn0$ during record). When the second timing event is replayed, it cannot find the corresponding callback as the logged id is different from the one generated during replay. For trace 2, when first timing event is replayed, the replayer cannot find the corresponding callback $fn0$, as it is not registered when an exception is thrown. When the second timing event is replayed, replayer still cannot find the callback $fn1$, as logged id is different from the one (now callback id of fn is 0) assigned during replay. Furthermore, both random numbers are not consumed, which may be wrongly used if remaining events also call `random` function.

From this simple example, it can be seen that existing record/replay technique can not ensure captured log will be correctly replayed when the platform changes during replay. The callback may be wrongly scheduled, which ultimately results in many false positives during XBIs detection as problematic page data is collected.

```
elem0.addEventListener('click',function(){
  ...
  var el = elem0.firstElementChild; //incompatible api;
  var val = el.innerHTML; //exception is thrown in IE;
  setTimeout(function fn0(){ //callback fn0
    coordinate = Math.random()*200+200;
    el.removed(val);},200);
    ...
});

elem1.addEventListener('click',function(){
  setTimeout(function fn1(){ //callback fn1
    coordinate = Math.random()*2000+2000;
    ...
  },2000);
});
```

Fig. 5. JavaScript Code

```
/************event trace 1************/
events:[
  {type:'click',target:xpath(elem0)},
  {type:'click', target:xpath(elem1)},
  {type:'setTimeout', id:0}, //fn0
  {type:'setTimeout', id:1}, //fn1
  ...
],
random:[
   ...,
   0.8898410063702613,  //generated by fn0
   0.4728045095689595,  //generated by fn1
   ...,
]
/************event trace 2************/
events:[
  {type:'click',target:xpath(elem0)},
  {type:'setTimeout', id:0}, //generated by fn0
  {type:'click', target:xpath(elem1)},
  {type:'setTimeout', id:1}, //generated by fn1
  ...
],
random:[
  ...,
  0.38825905695557594,  //generated by fn0
  0.11336620501242578,  //generated by fn1
  ...,
]
```

Fig. 6. Captured log

Cross-Browser Record/Replay. To detect behavior deviation and restrict the influence of incompatible API invocation on (replaying) other events, by adapting existing record/replay technique, X-Check proposed a novel cross-browser record/replay. In the new recorder, non-deterministic events (e.g., DOM events, timing events, Ajax events) will still be logged sequentially. The difference is that it will maintain a *local object* for each captured event, which saves the callback information of registered internal events and the result of non-deterministic function calls. During replay, for each internal event, before it is scheduled to replay, the replayer will determine whether correct callback is invoked by querying *local object* of the event (which registers this callback). For non-deterministic function call, replayer will return the result directly from the *local object* of current replayed event.

Recorder. The new recorder will first assign a unique id for each captured event. For timing event and Ajax event, different from the existing techniques which assign a global unique id for corresponding callback, the recorder will use the combination of current captured event id and a local id (which starts from index 0) to represent the callback. The interface of the wrapped callback of `setTimeout`/`setInterval` and *XHR* constructor are modified to receive event id (which registers the callback) and local id, which will be logged when these events are triggered.

In the *local object* of each event, the recorder will maintain a list for each type of internal browser events and non-deterministic functions. Currenlty, the local object contains four lists (see Fig.7): *timer*, *xhr*, *random* and *date*. For timing event, the callback body is saved (by invoking toString() method) along with local id. For Ajax event, when its constructor is invoked, besides generating a local id for this object, recorder also logs the request. When a response is returned, it will be saved into *xhr* list according to event id and local id. For `random` and `date` call, the results are directly saved into the *local list* maintained for each event (instead of a global list which existing record/replay techniques usually do).

Fig.7 shows the maintained *local object* list for two click events after it is serialized into the log, where key *id0* and *id1* represent the event id, registered callback of timer event and random values are saved locally into the *random* list and *timer* list.

```
var localObjList:= {
  id0:{
   timer:[{id:0, callback:''function fn0(){...}''}],
   xhr:[],
   random:[0.8898410063702613],
   date:[]
  },
  id1:{
   timer:[{id:0, callback:''function fn1(){...}''}],
   xhr:[],
   random:[0.4728045095689595],
   date:[]
}}
```

Fig. 7. local objects maintained for two click events by Recorder

```
var localObjList:= {
  id0:{
   timer:[{id:0, callback:fn0}],
   xhr:[]
  },
  id1:{
   timer:[{id:0, callback:fn1}],
   xhr:[]
}}
```

Fig. 8. local objects maintained for two click events by Replayer

Cross-browser replayer. To prevent an error caused by incompatible API invocation from affecting the replay of remaining events, the replayer needs to ensure correct callback of the event is invoked. To do this, for each replayed event, replayer also maintains a *local object* which saves the callback of timing event and request of *XHR*. Fig.8 shows the *local object* list for two click events (captured by recorder in Fig.7) maintained by replayer

For timing event, before it is replayed, replayer will check whether the callback (by querying its maintained *local object*)

is the same as the one captured by the recorder according to event id and local id. For comparison, the callback maintained by replayer (e.g., fn0) needs to invoke the toString() method firstly. If two callbacks are the same, replayer can determine this event is correctly scheduled and corresponding callback is executed. Otherwise, the replayer knows that the callback is wrongly scheduled, and this event will be annotated to have a behavior XBI. At the same time, replayer will query whether the callback can be found in other positions from its maintained timer list (stored in the *local object*). If found, the registered callback will also be executed. Otherwise, current event is skipped.

For Ajax event, before returning the response, replayer first checks whether the request is the same as the one saved by recorder according to event id (which invokes *XHR* constructor) and local id. If the request is the same, the response is returned from local *xhr* list. Otherwise, this event is tagged to have a behavior XBI. Replayer also tries to determine whether the request could be found in other positions of local *xhr* list saved by recorder, and return the response if same request is found. Otherwise, current Ajax event was skipped by ignoring corresponding application handler.

For non-deterministic function, such as `random` and `date`, when the constructor is invoked, the overriden one will consume values from the local *random/date* list saved by the recorder. It is possible that saved results are all consumed by previous `random`/`date` invocation. In this case, the replayer will call native `random`/`date` to generate new value, and current replayed event is tagged to have a behavior XBI.

After an event is replayed, it is possible that logged random/date values by the recorder are not completely consumed by the replayer, or registered timing events and *XHR* requests are not the same during record and replay. If any case occurs, a behavior XBI is identified.

X-Check does not maintain the registered DOM events in the *local object* of each captured event, as target element needs to be found firstly before invoking `dispatchEvent` to trigger an event during replay. If target cannot be located (using XPath), a behavior XBI will be identified timely.

Page Data Collection. During replay, after each event is replayed, DOM tree will also be traversed to collect page data, such as the attributes, the coordinate of each DOM node. The screenshot of current page is also taken (for image XBIs detection and visual XBIs report). However, if X-Check collects the page data for each replayed event (especially for timing event, user input event), replay will become very slow as page data need to be collected for each event even these events only incur subtle changes on the page.

To make replay more efficiently, during replay, X-Check makes the following two optimizations: 1) for continuous user input events, replayer will skip page data collection and continue to replay until next event is not user input event. Before replaying next event, the page data will be collected; 2) for continuous timing events, the replayer will collect the page data at fixed intervals.

C. Incremental XBI detection

Algorithm 1: compute mutated and layout-changed nodes

Input: map_i,map_j: DOM nodes of event e_i and e_j, j>i
Output: S: mutated nodes; F: layout-changed nodes

begin
  $S.added \leftarrow \emptyset$;$S.removed \leftarrow \emptyset$; ... ;$F \leftarrow \emptyset$;
  $(r_j, r_i) \leftarrow getRoot(map_j, map_i)$;
  if $checkCoords(r_j, r_i)$ **then**
    $insert(F, r_j)$;
  $insert(r_j, WORKLIST)$;
  while *WORKLIST is not empty* **do**
    $n \leftarrow extract(WORKLIST)$;
    foreach $c \in node.children$ **do**
      if $checkCoord(c, map_i.get(c.refId))$ **then**
        $insert(c, F)$;
      $update(S, computeMutate(c, map_i, map_j))$;
      $insert(c, WORKLIST)$;
  $computeRmv(S, map_i)$);
  return (S, F);

Page Data Preprocessing. For modern interactive web applications, most events only lead to partial change to the whole page. If only changed nodes are checked for each event, we can improve the efficiency of XBIs detection by avoiding checking those changeless nodes repeatedly. X-Check mainly considers two types of changed nodes: mutated nodes and layout-changed nodes. When an event is triggered, DOM tree will be modified (e.g., add a node) and those modified nodes are called mutated nodes. To detect behavior XBIs, X-Check also checks whether the same event generates same DOM mutation on different browsers. Layout-changed nodes represent those whose coordinate change after an event is triggered, and they are mainly used to detect structure XBIs.

It should be noted that mutated nodes only tell us the change of DOM tree. It is not enough to just consider mutated nodes when detecting structure XBIs, as these nodes also incur in layout change of adjacent DOM nodes. For example, in Fig.1, when a new book is added to table (mutated nodes include 1 <tr> and 3 <td> elements), it will affect the layout of parent table elements (i.e., the height of table increases). X-Check needs to check all coordinate-changed DOM nodes to avoid missing some structure XBIs.

X-Check computes mutated and layout-changed nodes by comparing two sequentially collected page data. To do this, when traversing DOM tree, replayer will assign a unique reference id (which is inserted into the DOM nodes as an un-enumerable attribute) to each visited DOM node. By traversing two DOM trees and compare nodes based on assigned reference id, X-Check can find mutated and layout-changed nodes very easily. Currently, it classifies mutated nodes as six types: *added*, *removed*, *reparented*, *reordered*, *attributeChanged* and *textChanged*, which are the same as the

classification used in mutation summary[6]. Here, we did not use mutation summary to get DOM mutation directly, as it relies on mutation observer[5], which is a new browser feature and only supported in latest version of some browsers (e.g., IE only supports this feature from version 11).

Algorithm 1 shows how to compute mutated and layout-changed DOM nodes based on breadth-first traversing technique. map_i and map_j store collected DOM nodes for event e_i and e_j respectively, in which the key is reference id and the value stores the node information. From the node, X-Check can also get the reference id of its parent and children. To compute mutated nodes, for each visited node c (line 10), algorithm first invokes $computeMutate$(line 12) to determine whether it is an *added*,*reparented*,*reordered*,*attributeChanged* or *textChanged* nodes. For example, if c does not appear in map_i, it is considered as an *added* node. To compute removed nodes, when c is found, it will be removed from map_i. After all nodes in map_j are visited, the nodes remaining in map_i belong to *removed* (function $computeRmv$)(line 14). Note that, as X-Check detects text-content XBIs separately, *added*/*removed* node sets in S do not contain text node. All mutated (including *added*/*removed*) text nodes are inserted into *textChanged* set. To distinguish removed text nodes with others, the content is set to special value -1.

Computing layout-changed nodes is simple. To do this, the algorithm first get the root nodes (r_i, r_j) (line 3) of the two DOM trees (represented by map_i, map_j). If the coordinates change (line 4), node r_j will be added F (line 5). During the traversal of map_j, if X-Check finds the coordinate of visited node changes (function $checkCoord$, line 10), it will add this node (represented by c) to F (line 11).

Overall Detection Algorithm. To detect XBIs, captured traces will be replayed in reference and test browsers respectively. During replay, if the schedule of an event in test browser diverges from the original's, a behavior XBI is identified.

Algorithm 2 presents our overall approach for XBIs detection. Its input is the captured trace $trace$, URL of the web application url, reference browser Br_0 and test browser Br_1. Its output is a list of XBIs. The first step is to replay each captured trace in two browsers. After replay, trace rtr_0 and rtr_1 will be returned, which stores the collected page data for replayed events. Each event has also an attribute to denote whether a behavior issue is identified during replay. For the multi-page web application, if the last event of the page being replayed cannot be fired successfully (which means failing to forward to next page), replayer will stops to replay following events.

Then algorithm traverses the replayed traces (rtr_0,rtr_1). For each pair of event (e_i^0, e_i^1), function $diffBehavior$ (line 4) will check whether they have behavior XBI B^e (detected by cross-browser replayer). If a behaivor XBI is detected, layout and content XBIs detection will be skipped to check the following events (line 6). To prevent duplicate XBIs detection, algorithm will first invoke $detectDOMTree$ (line 7) to check whether the DOM trees of the event (e_i^0, e_i^1) have been processed. To facilitate checking, each pair of DOM trees are hashed and compared with the saved hash values of DOM trees (which have been checked). The event (e_i^0, e_i^1) will be skipped if the corresponding DOM trees are processed. Function $getMutatedNodes$ (line 8) will be invoked to return mutated DOM nodes set (S_i^0, S_i^1). If S_i^0 and S_i^1 are both null, algoritm will continue to check the following events (line 10). If one set is null, and the other is not null, a behavior issue is identified (line 12). If S_i^0 and S_i^1 are both not null, $matchMutate$ is invoked to match mutated nodes (line 13). Note that, as X-Check only collect page data after page is loaded, *added* set will save the whole DOM tree if e_i^0 and e_i^1 are load events. Function $matchMutate$ will first match nodes stored in *added*, and then update match list map. After that, it checks *added*,*removed*, *reparented*, *reordered* and *attributeChanged* set to determine whether the same event leads to same DOM mutation. If not, a behavior XBI is identified (which is saved into B_i^n).

Algorithm 2: X-Check: XBI Detection Algorithm

Input: Br_0:ref. browser; Br_1: test browser
url:URL of target web application
$trace$: capture log
Output: X: List of XBIs

```
1  begin
2      (rtr_0, rtr_1) ← replay(Br_0, Br_1, url, trace);
3      foreach (e_i^0, e_i^1) ∈ (rtr_0, rtr_1) do
4          B_i^e ← diffBehavior(e_i^0, e_i^1);
5          if B_i^e is not null then
6              addErrors(B_i^e, X); continue;
7          if not detectDOMTree(e_i^0, e_i^1) then
8              (S_i^0, S_i^1) ← getMutatedNodes(e_i^0, e_i^1);
9              if S_i^0 and S_i^1 are both null then
10                 continue;
11             if S_i^0 or S_i^1 is null then
12                 addErrors(B_i^e, X); continue;
13             B_i^n ← matchMutate(match, S_i^0, S_i^1);
14             (F_i^0, F_i^1) ← getLayoutChgNodes(e_i^0, e_i^1);
15             (map_i^0, map_i^1) ← getDOMNodes(e_i^0, e_i^1);
16             (A_i^0, A_i^1) ← constructAG(map_i^0, map_i^1);
17             L_i^r ← detectLayout(A_i^0, A_i^1, F_i^0, F_i^1, match);
18             C_i^t ← detectText(S_i^0, S_i^1, match);
19             C_i^v ← detectImage(F_i^0, F_i^1, match);
20             addErrors(B_i^n, L_i^r, C_i^t, C_i^v, X);
21     return X;
```

Function $matchMutate$ matches nodes based on the following observation: DOM tree of the same page in different browsers has similar structure. Therefore, for each node (to be matched), unlike X-PERT which finds matched node in another DOM tree globally, it first tries to match node in the same layer. For the matched nodes, their children will continue to be visited to find matched nodes. For locally unmatched nodes, their children will be skipped. After traversal, each locally unmatched node will be matched globally. If matched

node can be found, their children will continually be matched locally. Otherwise, this node is denoted as unmatched node. The matching process continues until all locally unmatched nodes in the reference browser are visited.

To detect structure XBIs, the algorithm first invokes *getDOMNodes* to return traversed DOM tree (map_i^0,map_i^1), and then invoke function $constructAG$ to construct Alignment Graph (AG)(the construction process is the same as X-PERT). Function $getLayoutChgNodes$ will return layout-changed DOM nodes. Based on constructed AGs (A_i^0,A_i^1), algorithm will invoke $diffLayout$ to detect structure XBIs by only considering (F_i^0,F_i^1) . To detect text XBIs, $detectText$ just compares *textChanged* node set retrieved from mutated nodes (S_i^0,S_i^1). For layout-changed nodes which have neither structure nor text XBIs, $detectImage$ is invoked further to detect image XBIs. Similar to visual-content detection in X-PERT, it takes the images of two corresponding elements and compares their color histograms using χ^2 distance.

IV. Tool Implementation

We implemented our approach in a tool called X-Check, which is implemented in Java (execept record/replay library, which is implemented in standard JavaScript language). It uses Crawljax [18] to explore the Web application. A proxy called WebScarab [8] is configured to inspect http response destined for the client. When the response type is html, it is injected with our developed record/replay library. Another function of the proxy is to cache the response (e.g., html/js files), which will be returned to the client during replay.

To replay captured traces in reference and test browsers, a remote agent needs to be installed in the reference and test platforms. It is implemented as a daemon program, which will receive a trace list (to be replayed) after crawling and then starts the specified browser to replay. Remote agent also integrates Selenium WebDriver [7], which supports to take screenshot in almost all popular browsers, such as IE, Firefox, Chrome and Safari. After *load* event is replayed, remote agent will utilize *executeScript* method that Selenium WebDriver provides to execute $replay\_event()$ method of the replayer to replay event, and then invoke *getScreenshotAs* to save the image of current page.

To detect content XBIs in the third phase, X-Check compares the images using the implementation of the χ^2 metric in the OpenCV[10] computer vision toolkit, and performs textual comparison using string operations defined in the Apache Commons Lang library (http://commons.apache.org/proper/commons-lang/).

V. Evaluation

To assess the effectiveness and efficiency of our technique to detect XBIs, we used X-Check to conduct a thorough empirical evaluation on a suite of Web applications. In our evaluation, we investigated the following research questions:

- **RQ1:** What is the performance overhead of X-Check while recording/replaying non-deterministic events?
- **RQ2:** How efficient is incremental XBI detection algorithm in reducing duplicate XBI detection?
- **RQ3:** How does X-Check's ability to identify XBIs compare to that of a state-of-the-art technique?

In the rest of this section, we present the subject programs, the experimental protocol and our results.

A. Subject Applications

Table I shows ten subjects used in our evaluation, in which the first four are single-page Web games developed using HTML5, JavaScript and CSS3 technologies. *Annex* is a Reversi game. *Countbeads* teaches young kids counting number. *Rabbit* is a fun grid game where the rabbit finds and eats the carrots dodging the foxes. *Mancala* is a strategy game to capture more marbles than the opponent. The rests are multi-page Web applications. *Organizer* is an open source personal organizer and task management application. *Roundcube* is a web-based email client. *OpenCart* is an open source solution for e-commerce. *DivineLife* is an application about spiritual life. *CUPES* and *ISCAS* are two Chinese websites of colleage and institute, and the contents are updated periodically.

Along with the name, URL, and type of each subject, the table also reports minimum, maximum, and average number of DOM nodes analyzed per web page. This latter information provides an indication of the complexity of the individual pages. The main criterion for picking these subjects was the presence of known XBIs found manually in the study or false positives/negatives by only considering user interaction events.

B. Experiment Protocol

To investigate RQ1, we examined the overall performance overhead for the implementation of cross-browser record/replay. As chosen applications are interactive, for each of them, we developed an automated Selenium test case to simulate user interaction with the application. This study ran in the platform Chrome browser (v41.0) and Mac OSX 10.10.2 laptop with a 2.4 GHz Intel Core i7 processor and 8 GB of RAM. In order to estimate record/replay overhead, we compared the running time between original or "baseline" application and instrumented one when the latter is recorded and replayed, respectively. Recording time includes event logging and *local object* maintenance, and replaying time includes B_i^e detection and page data collection besides logging each replayed event.

To investigate question RQ2 and RQ3, we compare our tool with X-PERT, which represents the state of the art in detecting XBIs. In the experiment, X-Check was configured to run four browsers: Internet Explorer (v11.0, IE), Chrome (v41.0, CH), Firefox (v38.0, FF) and Safari (v8.0, SF). The former three browsers run in 32-bit Window 7 and the last runs in 64-bit Mac OS X 10.10.2. For each application, one browser is chosen as reference browser, where the behavior and appearance of the application are correct. The last two columns in Table I show the reference and test browsers for each application. To get event traces which can be replayed in different platforms, crawler is configured to explore ten subjects (running in reference platform), and at the same time

TABLE I
DETAILS OF THE SUBJECTS USED IN OUR EMPIRICAL EVALUATION.

Subject	URL	Type	DOM Nodes (per page)			Ref. Browser	Test Browser
			min	max	average		
Annex	https://01.org/html5webapps/online/annex/	Game	223	489	392	SF	IE, FF, CH
CountBeads	https://01.org/html5webapps/online/countbeads	Game	329	332	332		
Mancala	https://01.org/html5webapps/online/mancala	Game	357	436	430		
Rabbit	https://01.org/html5webapps/online/run-rabbit-run	Game	266	432	427		
Organizer	http://localhost:9090/theorganizer	Productivity	108	978	472	IE	CH, FF
Roundcube	http://localhost:8082/roundcube	Productivity	66	1857	600	CH	IE, FF
OpenCart	http://demo.opencart.com/	business	322	427	361		
DivineLife	http://sivanandaonline.org	Spiritual	334	1565	670	FF	IE, CH
CUPES	http://www.cupes.edu.cn/cenep/cupes	Information	134	674	345	CH	IE, FF
ISCAS	http://www.iscas.ac.cn/	Information	206	2441	973	FF	IE, CH

the recorder is plugged to capture various non-deterministic events. To detect more states which are missed by the crawler, we also recruited 2 students. When they interact with the applications, the event traces are collected.

To evaluate the proposed incremental XBIs detection technique, we focus on the performance of detecting structure XBIs in the experiment. To detect structure XBIs in X-PERT, for each pair of pages from different browsers, the DOM nodes are firstly matched, and then AGs are constructed and compared based on coordinate information. For ease of comparison, we modify X-Check to implement the structure XBIs detector proposed in X-PERT. After replay, the original detector and the improved one proposed in X-Check are invoked respectively to identify structure XBIs.

To perform a fair comparison with X-PERT for RQ3, we simulate X-PERT by firstly modifying the recorder to only capture DOM events during the crawling, which are then replayed remotely like X-Check. After each event is replayed, page data is collected and used to detect XBIs.

C. Results

To answer RQ1, we measured the average performance for record/replay by running the test cases 5 times. Table II presents the average performance overhead for our record/replay implementation when running on 10 subjects, along with the number of captured events and the log size. It can be seen that the average recording overhead was no higher than 1.14X, with a minimum 1.01X of and maximum of 1.87X. For the replay, we observe an average 2.75X slowdown, with a minimum of 1.56X and maximum of 4.95X. The results of this case study suggest that X-Check introduces low overhead for event capture, and can be used for online event trace collection in JavaScript-based Web applications. Compared to recording overhead, replaying overhead is higher, as X-Check needs to traverse DOM tree and take screenshot during this phase.

To answer RQ2, Table V summarizes and compares the performance of X-Check and X-PERT in structure XBIs detection. The table shows, for each subject, the detection time of the two tools, the collected log size, the number of states, and the test browser used for each subject. The experiment was repeated 3 times, and average detection time is computed. As the table shows, X-Check outperformed X-PERT in terms of the efficiency of XBIs detection with a considerable margin. For *divinelife*, for instance, X-Check only takes about 2s to detect structure XBIs, comparing to about 476s using X-PERT. Note that, although collected log size is similar for *CUPES*, X-PERT only takes about 86s to detect structure XBIs. The reason for such varying detection time is that for *divinelife*, the same page has more differences (in DOM tree structure) when rendered in reference and test browsers. Therefore, the global DOM nodes matching algorithm used in X-PERT spent more time to match nodes.

To answer RQ3, Table IV presents a detailed view of X-Check's results when running selected subjects. The table shows, for each subject, the true and false positives reported by X-Check for each type of XBIs that X-Check identified in different platforms. As the results show, X-Check reported 1400 true XBIs and 32 false positives for our selected subjects. The detected issues included all three types of XBIs, with a prevalence of content XBIs (791), followed by structure (570) and behavior (32) XBIs.

Table V summarizes and compares the results of X-Check and X-PERT. The table shows, for each subject, its name, the number of XBIs found by manual analysis (XBI), and the results of the two tools in terms of true positives (TP), false positives (FP). It can be seen that X-Check misses some XBIs, which are mainly image XBIs, as some subtle differences between two images cannot be identified by adopted image difference algorithm. However, as the table shows, X-Check improves X-PERT in terms of both precision and recall for all of the subjects considered. For subject *roundcube*, for instance, X-Check produced no false positives, as compared to $>$1000 false positives produced by X-PERT. The reason is that the crawler changes the template of the application when exploring the states. When captured trace is replayed in different platforms, each page will have different appearance. Therefore, X-PERT reports a lot of structure XBIs and image XBIs. For *rabbit* and *mancala*, as these two applications use a lot of random number and timing events, their behavior and appearance will be different greatly when running in different platforms, and thus X-PERT generates many false positives. Based on these results, we can conclude that, for the subjects considered, X-Check is indeed effective in finding XBIs and does improve over the state of the art.

TABLE II
RECORD/REPLAY OVERHEAD OF X-CHECK

Subject	Events	Log Size (KB)	Record Overhead	Replay Overhead
Annex	323	485	1.01X	4.95X
CountBeads	721	651	1.05X	1.73X
Mancala	83	84.1	1.11X	2.09X
Rabbit	246	256	1.02X	3.13X
Organizer	156	329	1.14X	2.83X
Roundcube	162	455	1.04X	2.31X
OpenCart	288	217	1.02X	2.51X
DivineLife	47	83.6	1.87X	2.49X
CUPES	85	185	1.03X	3.85X
ISCAS	613	284	1.06X	1.56X

TABLE III
X-CHECK'S RESULTS COMPARED TO X-PERT

Subject	XBI	X-CHECK				X-PERT			
		TP	FP	Prec.	Recall	TP	FP	Prec.	Recall
annex	15	13	0	100%	86%	10	0	100%	66%
countbead	19	17	0	100%	89%	5	100	4%	21%
mancala	96	94	0	100%	97%	74	102	42%	77%
rabbit	485	483	8	98%	99%	474	125	79%	97%
organizer	8	8	0	100%	100%	8	0	100%	100%
roundcube	21	19	7	73%	90%	19	>1000	-	90%
opencart	0	0	0	100%	100%	0	93	0	100%
divinelife	694	694	2	99%	100%	694	2	99%	100%
cupes	53	52	0	100%	98%	52	38	57%	98%
iscas	20	20	2	90%	100%	20	143	12%	100%

TABLE IV
X-CHECK DETAILED RESULTS

Subject	BEHAV.		STRUCT.		CONTENT				Total		Browser
					TEXT		IMAGE				
	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	
annex	3	0	1	0	0	0	3	0	7	0	IE
	1	0	1	0	0	0	3	0	5	0	FF
	1	0	0	0	0	0	0	0	1	0	CH
countbead	1	0	0	0	0	0	5	0	6	0	IE
	1	0	3	0	3	0	3	0	10	0	FF
	1	0	0	0	0	0	0	0	1	0	CH
mancala	1	0	16	0	0	0	31	0	48	0	IE
	1	0	14	0	1	0	29	0	45	0	FF
	1	0	0	0	0	0	0	0	1	0	CH
rabbit	1	0	424	8	0	0	32	0	457	8	IE
	17	0	2	0	3	0	3	0	25	0	FF
	1	0	0	0	0	0	0	0	1	0	CH
organizer	1	0	1	0	0	0	0	0	2	0	CH
	1	0	5	0	0	0	0	0	6	0	FF
roundcube	0	0	7	0	0	0	4	7	11	7	IE
	0	0	8	0	0	0	0	0	8	0	FF
OpenCart	0	0	0	0	0	0	0	1	0	0	IE
	0	0	0	0	0	0	0	0	0	0	FF
DivineLife	0	0	8	0	0	0	678	1	686	1	IE
	0	0	8	0	0	0	0	1	8	1	CH
CUPES	0	0	21	0	0	0	0	0	21	0	IE
	0	0	31	0	0	0	0	0	31	0	FF
ISCAS	0	0	8	9	0	0	0	0	8	9	IE
	0	0	12	6	0	0	0	0	12	6	CH
Total	32	0	570	23	7	0	791	9	1400	32	-

TABLE V
STRUCTURE XBI DETECTION TIME OF X-CHECK COMPARED TO X-PERT

Subject	STRUCT.(ms)		Log(MB)	States	Browser
	X-Check	X-PERT			
Annex	534	7994	19.2	192	FF
Countbeads	237	3101	3.38	49	FF
Mancala	464	1179	2.87	51	FF
Rabbit	763	10457	6.88	110	FF
Organizer	751	5506	5.21	93	CH
Roundcube	3391	22289	9.98	73	FF
OpenCart	769	6311	11.1	60	FF
DivineLife	2042	476708	31.9	48	CH
CUPES	917	86851	30.5	80	FF
ISCAS	1129	6905	12	15	CH

VI. THREATS TO VALIDITY

As with most empirical studies, there are some threats to the validity of our results. In terms of external validity, in particular, our results might not generalize to other web applications and XBIs. To minimize this threat, we selected real-world web applications and applications used in previous studies. Although we only evaluated four types of browsers, as we avoid to use browser-specific logic (e.g., we didn't use mutation observer to get mutated DOM nodes, as it is only supported in IE from version 11), our technique is expected to work similarly on other browsers.

Threats to construct validity might be due to implementation errors in X-Check especially with respect to coordinate remote agent (a java client) with replay library (implemented in JavaScript and plugged into web application) to support collect page data after replaying an event. We mitigated this threat through manual inspection of replay results to prevent screenshot is not consistent with extracted DOM data.

VII. LIMITATION

Although our technique achieves better results than previous work, it still has certain limitations. We discuss these in the following while noting that many of these can be mitigated by further research in this area.

As X-Check does not support some new HTML5 events (e.g., drag and drop) and Web API (e.g., websocket, web worker), it may still produce some false positives/negatives if the application uses these new features. We plan to support more non-deterministic events in the future work. The proposed cross-browser replayer can also be extended to support more internal browser events. Another source of some false positives comes from CSS3 animation, which can change element's coordinate without executing JavaScript code. To avoid reporting such XBIs, one solution is to extract DOM nodes before/after replaying an event. By comparing element's coordinate collected after replaying an event with the one collected before replaying next event, it can be inferred that layout change is caused by CSS3 animation rule.

For complex web applications, crawler may only explore parts of the whole state space, which results in missing some XBIs if the state (which contains XBIs) can not be explored. Based on the proposed record/replay technique, it is also possible to collect more real traces from users after the application is deployed into the field, and then replay them in different platforms to detect XBIs.

VIII. Related Work

Cross Browser Testing. As the relevance of XBIs, a lot of techniques and tools have been proposed to address this limitation. Early endeavor includes Eaton and Memon[15], who proposed a technique to identify problematic HTML tags based on manual classification of good and faulty pages. Tamm[23] presents a tool to find layout issues on a page based on visual and DOM information. Both techniques test a web application within a single web browser, and not cross-browser techniques.

The research work most closely related to ours is crawl-and-compare based technique for detecting XBIs, such as CrossT[17], WebDiff[13], CrossCheck[12],and X-PERT[21].These techniques generally consists of two steps. Firstly, behavior capture step automatically crawls and captures the behavior of web application in different browsers. Captured behavior includes screenshots and layout data of individual pages, as well as models of user-actions and inter-page navigation. Then, comparison step automatically compares the captured behavior to identify XBIs. WebMate[14] adopts similar technique, with the focus on improving the coverage and automation of the crawling.

Browsera[1], MogoTest[4], and Browserbite[2] are the first industrial offerings in this category. They use a combination of some limited automated crawling and layout comparison. In our experience with these tools, these tools generated a lot of false positives for some web applications, as they did not capture non-deterministic events insider the browser.

Record and Replay. Recently, there has been a lot of work on record/replay JavaScript-based web application. Most of these work aims to failure analysis and performance evaluation. In terms of the implementation, they can be divided into two kinds: browser-agnostic (e.g., Mugshot[19], Jalangi[22], JSBench[20]), and browser-specific (e.g.,Timelaspe[11], WaRR[9]).

Mugshot[19] is a record/replay tool for JavaScript-based Web application which can capture/replay various non-deterministic events in client-side JavaScript program. The record/replay component is implemented entirely in standard JavaScript, providing event capture on unmodified browsers. Jalangi[22] is a selective record/replay framework for JavaScript application, which enables recording and replaying a user-selected part of the program. One limitation of Jalangi is the high performance overhead, which makes it not appropriate to be an *always-on* record/replay tool for deployed JavaScript applications. JSBench[20] supports to create JavaScript benchmarks to evaluate the performance of JavaScript implementation using record/replay technique. It acts as a proxy between the browser and the server, rewrites JavaScript code on the fly to add instrumentation, and finally creates an executable stand-alone application that can be replayed under different browser environments.

Different from the above work, Timelapse[11] captures non-deterministic events by instrumenting the API of web interpreters. Currently, it is built into Safari browser as a part of web inspector. WaRR[9] is a tool for high-fidelity web application record/replay. The recorder is embedded into the WebKit-based browser and captures various DOM events (e.g.,click, keystroke), and the replayer uses a developer-specific web browser to simulate user interactions.

IX. Conclusion

Because of the richness and diversity of today's web platforms, XBIs are becoming a severe issue for organizations to develop web-based software. Existing techniques are prone to generate both false positives and false negatives as they can not assure the same execution when the web application runs in different web platforms. To address this issue, in this paper, we proposed a novel cross-browser testing approach by leverage existing record/replay technique, which can reproduce a previous execution under different user environments as faithfully as possible. Our result shows that X-Check is effective in detecting XBIs, and outperforms the state-of-the-art technique. In the future work, we plan to investigate techniques that can automatically eliminate XBIs through browser-specific web page repairs.

X. Acknowlegements

The part of record/replay work is done with Alex Orso and his student Jie and Juyuan when the first author visits Geogia Tech as a visiting scholar. Thank Shauvik for his input and several fruitful discussions.

References

[1] Browsera.http://www.browsera.com/.
[2] Browserbite. http://app.browserbite.com/.
[3] Document object model level 2 core specification. http://www.w3.org/tr/dom-level-2-core/.
[4] Mogotest. http://mogotest.com/.
[5] Mutation observer.http://www.w3.org/tr/dom/mutation-observers.
[6] Mutation summary.https://github.com/rafaelw/mutation-summary.
[7] Webdriver. http://www.w3.org/tr/2013/wd-webdriver-20130117/.
[8] Webscarab. https://www.owasp.org/index.php/category:owasp_webscarab_project.
[9] S. Andrica and G. Candea. Warr: A tool for high-fidelity web application record and replay. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 403–410. IEEE, 2011.
[10] G. Bradski and A. Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008.
[11] 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'13, pages 473–484, New York, NY, USA, 2013. ACM.
[12] S. R. Choudhary, M. R. Prasad, and A. Orso. Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 171–180. IEEE, 2012.
[13] S. R. Choudhary, H. Versee, and A. Orso. Webdiff: Automated identification of cross-browser issues in web applications. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.
[14] V. Dallmeier, M. Burger, T. Orth, and A. Zeller. Webmate: A tool for testing web 2.0 applications. In *Proceedings of the Workshop on JavaScript Tools*, JSTools '12, pages 11–15, New York, NY, USA, 2012. ACM.
[15] C. Eaton and A. M. Memon. An empirical approach to evaluating web application compliance across diverse client platform configurations. *Int. J. Web Eng. Technol.*, 3(3):227–253, Jan. 2007.

[16] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang. Making system user interactive tests repeatable: When and what should we control? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 55–65, Piscataway, NJ, USA, 2015. IEEE Press.

[17] A. Mesbah and M. R. Prasad. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 561–570. ACM, 2011.

[18] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3, 2012.

[19] 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'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[20] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of javascript benchmarks. *ACM SIGPLAN Notices*, 46(10):677–694, 2011.

[21] S. Roy Choudhary, M. R. Prasad, and A. Orso. X-pert: Accurate identification of cross-browser issues in web applications. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 702–711. IEEE Press, 2013.

[22] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498, New York, NY, USA, 2013. ACM.

[23] M. Tamm. Fighting layout bugs, http://code.google.com/p/fightinglayoutbugs/, october 2009.