2018 IEEE 11th International Conference on Software Testing, Verification and Validation

Automatic Detection of Visibility Faults by Layout Changes in HTML5 Web Pages

Yeonhee Ryou Sukyoung Ryu
School of Computing, KAIST, South Korea
{yeonhee.ryou, sryu.cs}@kaist.ac.kr

***Abstract*—Modern HTML5 web pages (pages) often change various elements of their documents dynamically to provide rich functionality to users interactively. As users interact with a document via events, the combination of HTML, CSS, and JavaScript dynamically changes the document layout, which is the arrangement of the document elements visualized to the users. Web pages change their layouts not only to support user interaction but also to react to different screen sizes being used to run the pages. To support diverse devices with different screen sizes using a single web page document, developers use Responsive Web Design, which enables web page layouts to change when the sizes of the underlying devices change. While such dynamic features of web pages provide powerful experiences to users, they also make development of web pages more difficult. Even expert developers find it difficult to write HTML5 web pages correctly. In this paper, we first define the problem that functionalities of HTML5 web pages may become unusable due to layout changes, and propose a technique to detect the problem automatically. We show that our implementation detects such problems in real-world HTML5 web pages.**

I. Introduction

The HTML5 technology [1] enables web pages to dynamically change various elements of their documents, even for a single web page. The multiple languages used in HTML5 allow rich interaction between web pages and their users. The page content is specified in HTML, the page layout is managed by CSS, and the user interaction is handled by JavaScript. When a user touches a button of a dialogue page, the document may change its layout still remaining in the same page by using Document Object Model (DOM) APIs. DOM is a platform- and language-independent interface that enables programs to access and update the content, structure, and style of HTML documents dynamically [2]. Because JavaScript can run on any devices with a browser, it facilitates development of portable web pages. Thus, popular JavaScript libraries such as React [3] and AngularJS [4] support dynamic changes of documents incrementally and efficiently.

The layouts of HTML5 pages may change not only by their interaction with users but also by the screen sizes being used to run that pages [5]. A wide screen can show multiple columns of content while a small screen may show the content partially in a single column. As devices of various sizes such as smartphones and tablets are being widely used [6], *Responsive Web Design* (RWD) has become one of the most important features in web pages [7]. RWD enables the layout of a page to change as the screen size running the web page changes. In order to support customers with multiple devices of various sizes and to maintain one web page for various sizes of browsers, RWD aims to make web pages display appropriately across different screen sizes and on various devices. Thus, responsive web pages display their elements differently for different screen sizes. They change the sizes, shapes, and arrangements of their elements, or make some elements invisible on small-sized screens using DOM APIs [8].

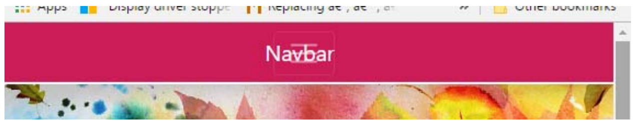


Fig. 1: Screenshot attached at the issue #22028 of Bootstrap. The `.navbar-brand` element containing the brand name Navbar overlaps the round rectangle button with 3 lines, which is the `.navbar-toggler` element.

However, such powerful and useful dynamic features of web pages make their development and management more complex and difficult. Because each element in a document may be dynamically rearranged during execution, developers should consider abundant cases. Moreover, since such dynamic transformation of individual elements and the whole structure of the document depend on the size, namely the *viewport width* of a browser, developers should manage each context of modification separately, which is tedious, labor-intensive, and error-prone.

In addition, a small mistake in dynamic visualization of document layout can hinder document elements from providing their functionality to users. For example, the issue #22028 of Bootstrap[1] discusses a problem that an element overlaps another element due to differences in the viewport width. As shown in Figure 1, when the viewport width is reduced to 575 pixels or less, the round rectangle button with 3 horizontal lines, which is a `.navbar-toggler` element, and the brand name Navbar on a `.navbar-brand` element overlap each other so that it is difficult to find the `.navbar-toggler` element and to read the brand name on `.navbar-brand`, Navbar. This issue has more than 10 related issues that reference each other, and a solution is proposed.

Nevertheless, few works have tried to address such layout faults. One approach is ReDeCheck [9], a tool that

[1]https://github.com/twbs/bootstrap/issues/22028

detects responsive layout failures (RLFs) in responsive web pages automatically by analyzing HTML documents and CSS rules. However, REDECHECK cannot detect dynamic layout changes due to JavaScript events, because it does not consider JavaScript, which bridges web pages and the event system in a browser. Another approach is to detect Cross-Browser Inconsistency (XBI) by tracking dynamic execution flows of web pages, but it is not applicable to solve the layout fault problem in HTML5 web pages either. Since XBI detection tools such as WebDiff [10] focus on the effects of diverse browser environments, they assume that each browser has a fixed viewport width, which is not suitable for responsive web pages. More specifically, the XBI tools assume that a DOM element has the same color, position, and size in every browser, which is frequently violated in responsive web pages.

In this paper, we identify new problems related to the complexity of dynamically changeable layouts, and propose a technique to automatically detect them in real-world HTML5 web pages. To help developers build correct HTML5 web pages, we first introduce a new kind of problems, *visibility faults*. Visibility faults have two cases: 1) dynamic reconstruction of a document leads to a different visibility of its elements, and 2) the different visibility of document elements leads to different (possibly wrong) behaviors. We propose *comparability*, which is the key concept in defining whether a DOM element has a visibility fault or not. Then, we present a technique that finds visibility faults automatically. We implement the technique, and show that the implementation detects genuine visibility faults from real-world HTML5 web pages.

The main contributions of this paper include the following:

1) We identified a new kind of problems that stem from complex and difficult development of HTML5 web pages. The problems may break the visibility of web pages, and they may even prohibit users from accessing some elements of the web pages.
2) We developed a tool that detects the problems automatically and showed that the tool reports genuine problems correctly from real-world HTML5 web pages, efficently.
3) We found that widely-used HTML5 web pages contain various visibility faults, which may hinder their usability.

In the rest of the paper, we first explain the problem with a concrete example (Section II). Then, we formally define the visibility fault problem and comparability (Section III), and an automatic detection mechanism of the visibility faults (Section IV). We evaluate the proposed mechanism using real-world web pages that contain visibility faults in two respects: whether the tool detects the faults precisely and efficiently, and whether the tool is comparable to REDECHECK (Section V). We discuss related work (Section VI) and conclude (Section VII).

II. MOTIVATION

In this section, we present an example that illustrates visibility faults stemming from the complexity of dynamically changeable elements.

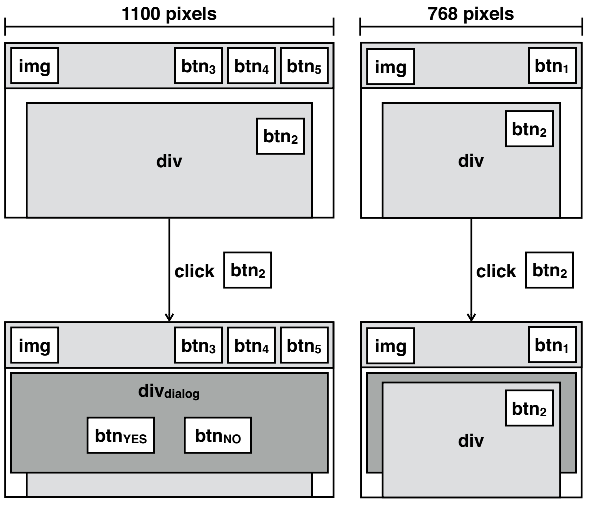
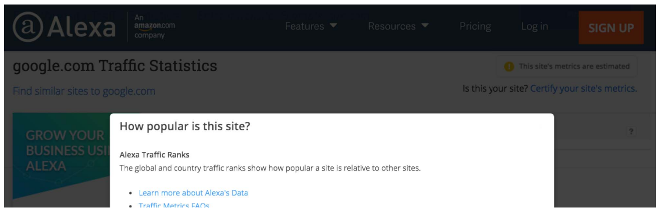


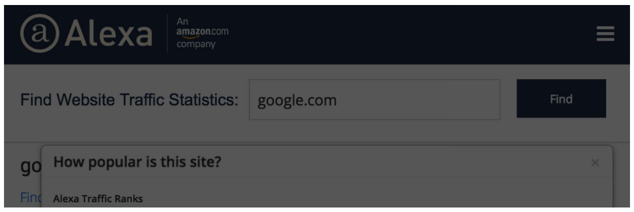
Fig. 2: A simple example HTML5 web page loaded by two different viewport widths, 1100 pixels and 768 pixels.

Figure 2 demonstrates how an HTML5 web page can have diverse layouts depending on viewport widths and dynamic execution of event handlers. The two layouts on top of Figure 2 show the effects of viewport widths. With the viewport width 1100 pixels, the web page layout displays multiple button elements on a navigation bar, which are $\textbf{btn}_3$, $\textbf{btn}_4$, and $\textbf{btn}_5$. On the contrary, with the viewport width 768 pixels, the buttons are not displayed, but $\textbf{btn}_1$ is displayed instead. Since both layouts have the same button $\textbf{btn}_2$, we expect that the event handler on $\textbf{btn}_2$ would have the same functionality on both layouts. The two layouts on bottom of Figure 2 are the results of dynamic changes triggered by the click event on $\textbf{btn}_2$. The execution of the event handler dynamically creates the $\textbf{div}_{\text{dialog}}$ element on both layouts, which provide two buttons to users of the web page, $\textbf{btn}_{\text{YES}}$ and $\textbf{btn}_{\text{NO}}$. However, with the viewport width 768 pixels, $\textbf{div}_{\text{dialog}}$ is created below the existing **div** element, which covers the contents of $\textbf{div}_{\text{dialog}}$. Because $\textbf{btn}_{\text{YES}}$ and $\textbf{btn}_{\text{NO}}$ on the dialog are covered by **div**, the dialog cannot work correctly.

The example in Figure 2 comes from a layout fault in a real-world web page, Alexa [11] shown in Figure 3. Alexa is a traffic analysis service of `amazon.com`, and its web page uses HTML5 techniques to support devices of various sizes and dynamic execution of user events. Figure 3 presents a dialog created on the statistic page for the `google.com` site with different viewport widths by Alexa. In Figure 3(a), the dialog (partially shown in the figure) includes short description about the page contents and several links to more detailed explanation, and a black translucent background is created together between the dialog and main contents of the page highlighting the dialog. While both layouts in Figure 3 are generated by the same event execution, in Figure 3(b), the black translucent background is created above the dialog, and covers the dialog as well as the main contents of the page.


Alexa
Features
Resources
Pricing
Log in
SIGN UP
google.com Traffic Statistics
Find similar sites to google.com
This site's metrics are estimated
Is this your site? Certify your site's metrics.
GROW YOUR BUSINESS USI ALEXA
How popular is this site?
Alexa Traffic Ranks
The global and country traffic ranks show how popular a site is relative to other sites.
Learn more about Alexa's Data
Traffic Metrics FAQs

(a) Help dialog with the viewport width 1100 pixels



(b) Help dialog with the viewport width 768 pixels

Fig. 3: Help dialogs of the sections How popular is this site? with browsers of the viewport widths 1100 pixels and 768 pixels

The weird behavior with the viewport width 768 pixels is mainly due to the complex semantics of dynamically rearranging DOM elements. When DOM elements are displayed at a browser, a set of rules described in the W3 specification [12] determines the *back-to-front order* among DOM elements, where "front" elements are above "back" elements. However, there exist various DOM element properties such as `transform` [13], `isolation` [14], and `opacity` [15], and their corresponding rules that can change the back-to-front order relation. In Figure 3(a), the dialog element is determined to be in front of the background element with the viewport width 1100 pixels, but, in Figure 3(b), they are in a reverse order. The only reason caused the difference is the value of the `transform` property of a DOM element in the pages, which is used to adapt RWD to the Alexa page. Thus, dynamic layout changes may change the back-to-front order among DOM elements, and the changed back-to-front order may limit the functionality of DOM elements.

In order to detect the layout fault in Figure 2, we need to identify three facts. First, we have to identify which pairs of layouts should have consistent functionalities. For example, the two layouts on bottom of Figure 2 should have consistent functionalities. Also, in such layouts, we have to identify DOM elements that we should compare. For example, we should compare $\textbf{btn}_{\text{YES}}$ and $\textbf{btn}_{\text{NO}}$ on both sides, but we should not compare $\textbf{btn}_3$, $\textbf{btn}_4$, $\textbf{btn}_5$, and $\textbf{btn}_1$ since they intentionally exist on only one side. Finally, we have to identify the visual status of the DOM elements that we compare. For example, we have to identify that we should compare whether $\textbf{btn}_{\text{YES}}$ and $\textbf{btn}_{\text{NO}}$ are covered by other elements or not.

In the next sections, we formally define layout faults and an algorithm that detects the faults automatically.

III. Event Object Visibility Faults

We define necessary terminology and a formal definition of the problems due to layout changes in HTML5 pages.

A. Event Objects

Throughout the paper, we collectively call HTML documents and their linked or embedded CSS style sheets and JavaScript code *web pages* or simply *pages*. Web pages may change their layouts dynamically by executing JavaScript code, typically via user *events*. When a user triggers an event like a mouse click or an orientation change, the registered JavaScript function to the event, if any, is executed. Among various kinds of events, we consider only 26 types of UI Events defined in the W3C specification [16] such as `load`, `blur`, and `click` to focus on events that handle user interaction rather than timeout and device-specific events.

Definition 1: An *event* is a pair of an event type and its target DOM element. An event type is one of 26 UI Events and every DOM element is classified into 102 *tag* types [17].

$$e \in \mathit{Event} = \mathit{EventType} \times \mathit{DOMElement}$$

Web pages can handle events by registering JavaScript functions called *event handlers* and executing them when their corresponding events are triggered. An event handler may be registered to a DOM element by following ways:

1) The DOM API function `addEventListener` registers a given event handler to a given DOM element.
2) DOM elements of some tag types have event attributes such as `onload`, `onclick`, and `onresize`. The value of an event attribute is registered as an event handler of the corresponding DOM element.
3) DOM elements of some tag types have implicitly registered event handlers for specific event types. For example, if a DOM element of the tag `a` has the field `href` of a URI value, triggering an event `click` targeted at the element changes the current page to the value of its `href` field.

Definition 2: An *event object* is a DOM element, which has one or more registered event handlers.

In Figure 2, we assumed that **img** and **div** are not event objects because they have no event handlers, while **btn** objects are event objects because they have the `button` tag type.

B. Visibility of Event Objects

DOM elements in web pages have diverse properties that determine their layouts in the pages such as positions and sizes. The W3C specification [2] describes complex rules that calculate the properties of DOM elements and visualize the elements on browsers. For example, if a DOM element has the field `display` of value `"none"`, the element does not have any size or position value, which makes browsers hide the element from users. In addition, the back-to-front order determines how DOM elements are stacked in pages.

When a user accesses a page, how DOM elements are displayed may affect how event objects execute on the page. As a special case of DOM elements, event objects may be invisible to users, and their event handlers may not be executable depending on their properties. If an event object has the field `opacity` of value `0`, even though it has its size and position values and exists in a web page, it is not revealed to users. Thus, while such opaque objects are invisible to users, users can execute event handlers registered to opaque event objects. On the other hand, if an event object is fully covered by another DOM element by the back-to-front semantics, the event object is invisible to users even though it exists in a web page. However, unlike opaque objects, users cannot execute event handlers registered to fully covered event objects.

Thus, we categorize event objects according to whether they are revealed to users, whether their event handlers are executable, and whether they are laid out within `document.body`, which represents the main content of the document [18].

Definition 3: The *visibility* of an event object is one of the following:

- *absent* if the object has the field `display` of value `"none"` or the field `visibility` of value `"hidden"`. An absent event object is intentionally invisible to users and users cannot execute the event handlers registered to it. In Figure 2, $\textbf{btn}_3$ is absent in the top-right layout.
- *full-cover* if the object is *fully* covered by other objects. A full-cover event object is invisible to users and users cannot execute its event handlers. In the bottom-right layout of Figure 2, $\textbf{btn}_{\text{YES}}$ is full-cover since it is fully covered by the **div** element.
- *partial-cover* if the object is *partially* covered by other objects. A partial-cover event object is partially revealed to users and users can execute its event handlers.
- *off-screen* if the object is not covered by other objects and some part of the object is outside `document.body`. Users can see off-screen objects and execute their event handlers if they can scroll the screen beyond `document.body`.
- *normal* otherwise. A normal event object is present on top of other elements and its event handler is executable.

Note that absent event objects do not occupy any area in web pages, thus have no impacts on the visibility of other elements. For presentation brevity, we ignore absent event objects in web pages throughout this paper.

C. UI State Graphs

Now, we define the state of a web page, transitions between them, and graphs representing transitions. An event object may change its visibility dynamically due to dynamic changes of DOM elements, and DOM elements may change dynamically by changes of viewport widths of browsers, or by execution of event handlers. To focus on impacts of event handlers on the visibility changes of event objects, we assume that we use a single browser of a specific viewport width for now.

We represent the state of a web page as a set of event objects with their visibility.

Definition 4: A *UI state* is a set of event objects. We denote an event object and its visibility as a pair.

$$\sigma \in \textit{UIState} = \mathcal{P}(\textit{EventObject} \times \textit{Visibility})$$

For example, the bottom-right layout in Figure 2 can be represented as follow: $\{\langle \textbf{btn}_1, \text{normal}\rangle, \langle \textbf{btn}_2, \text{normal}\rangle, \langle \textbf{btn}_{\text{YES}}, \text{full-cover}\rangle, \langle \textbf{btn}_{\text{NO}}, \text{full-cover}\rangle\}$.

Execution of event handlers may change the visibility of event objects. In Figure 2, if a user triggers a `click` event on $\textbf{btn}_2$ in the top-left layout, $\textbf{div}_{\text{dialog}}$ rises above other elements changing its `display` property from `none` to `block`. Then, the children of $\textbf{div}_{\text{dialog}}$, such as $\textbf{div}_{\text{YES}}$ and $\textbf{div}_{\text{NO}}$, change their visibility from absent to normal. We call an event handler execution a *transition*. Note that an event may have multiple event handlers registered, and one event handler may make various changes. In this paper, we do not trace all such changes but focus on only the results of changes, a UI state.

Definition 5: A *transition* denotes that an event changes a UI state to another UI state by executing its event handler.

$$\langle \sigma, e, \sigma' \rangle \in \textit{Transition} = \textit{UIState} \times \textit{Event} \times \textit{UIState}$$

In Figure 2, the bottom-left layout is generated by the `click` event of $\textbf{btn}_2$ at the top-left layout. If σ and σ' are the top-left and bottom-left layouts, respectively, then there exists a transition $\langle \sigma, (\textbf{click}, \textbf{btn}_2), \sigma' \rangle$.

Now that we have defined UI states and their transitions, we define *UI state graphs* with UI states as vertices and transitions as directed edges for a given web page and a browser. The UI state right after loading the page is a unique *initial UI state*. Both layouts on top of Figure 2 are initial UI states.

Definition 6: A *UI state graph* is a directed graph with a set of UI states and a set of transitions between them.

$$G \in \textit{UIStateGraph} = (V,\ E)$$

where $V = \mathcal{P}(\textit{UIState})$ and $E = \mathcal{P}(\textit{Transition})$.

Definition 7: For two UI states σ_0 and σ_m, a *path* π from σ_0 to σ_m is a sequence of transitions as follows:

$$\pi = \langle \sigma_0, e_1, \sigma_1 \rangle,\ \cdots,\ \langle \sigma_{m-1}, e_m, \sigma_m \rangle.$$

All the UI states in a UI state graph are reachable from the initial UI state. Also, we assume that the initial UI state is reachable from every other UI state. Thus, every UI state in a web page should have a sequence of events to get back to the initial UI state. It is a reasonable assumption because the initial UI state usually contains the main contents of the page.

Property 1: For a UI state graph G, let σ_0 be the initial UI state. Then, for every UI state $\sigma \in V$, G contains (1) a path from σ_0 to σ, and (2) a path from σ to σ_0.

In addition, we assume that execution of an event handler always results in a unique UI state even though it may involve nondeterministic semantics. Thus, even when an event handler takes user inputs or uses random numbers during its execution, it should produce a unique UI state regardless of its inputs and random numbers. This assumption helps us develop a deterministic algorithm in Section IV.

Property 2: For any transition $\langle \sigma, e, \sigma' \rangle \in E$, if there exists a transition $\langle \sigma, e, \sigma'' \rangle \in E$, then $\sigma' = \sigma''$.

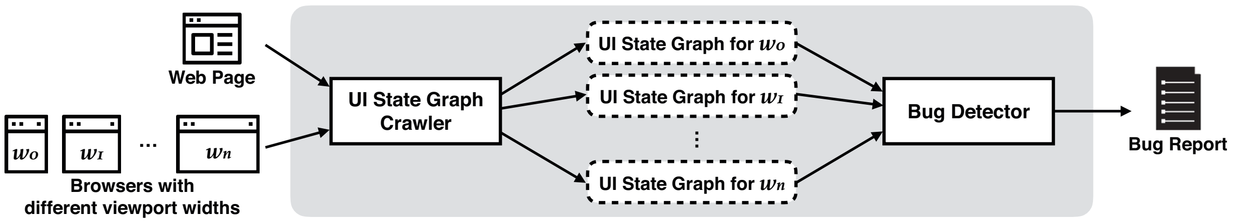


Fig. 4: Overall structure of visibility fault detection

D. Comparability

Then, let us eliminate the assumption that we use a single browser of a specific viewport width by defining the *comparability* relation $\equiv$ between UI states. We consider that all the UI state graphs for a single page with different viewport widths have the initial UI states that are comparable to each other.

Definition 8: Consider a single web page and its UI state graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with the initial UI states σ_0^1 and σ_0^2, respectively, for different viewport widths. Two UI states $\sigma_1 \in V_1$ and $\sigma_2 \in V_2$ have the comparability relation $\sigma_1 \equiv \sigma_2$, said to be comparable to each other, if one of the following conditions holds:

1) $\sigma_1 = \sigma_0^1$ and $\sigma_2 = \sigma_0^2$, or
2) if there exist $\sigma_1' \in V_1$ and $\sigma_2' \in V_2$ such that $\sigma_1' \equiv \sigma_2'$, $\langle \sigma_1', e, \sigma_1 \rangle \in E_1$, and $\langle \sigma_2', e, \sigma_2 \rangle \in E_2$ for some e.

In Figure 2, two UI states on top are comparable to each other by the first condition and two on bottom are by the second.

E. Visibility Faults

We observe that, for UI state graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, if an event object exists in the UI states $\sigma_1 \in V_1$ and $\sigma_2 \in V_2$ such that $\sigma_1 \equiv \sigma_2$, the event object should have the same visibility in σ_1 and σ_2 so that users can use the event object consistently in different UI state graphs. In other words, users should be able to use event objects in the same way on browsers of different viewport widths. Thus, we define *visibility faults* (VFs) that hinder usability of web pages due to layout changes.

Definition 9: For UI state graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, and $\sigma_1 \in V_1$ and $\sigma_2 \in V_2$ such that $\sigma_1 \equiv \sigma_2$, let o be an event object that exists in both σ_1 and σ_2.

- If the visibility of the event object o in the UI states σ_1 and σ_2 are different, o has an *inconsistent fault*.
- If o has an inconsistent fault and the visibility of o in one of σ_1 and σ_2 is full-cover, o has a *covered error*.

Inconsistent faults and covered errors are collectively called visibility faults.

In Figure 2, **btn**$_{\text{YES}}$ has a covered error. The layouts on bottom are comparable, and both have **btn**$_{\text{YES}}$. However, the visibility of **btn**$_{\text{YES}}$ is normal in bottom-left and full-cover in bottom-right, which is a covered error of **btn**$_{\text{YES}}$.

If an event object o has an inconsistent fault but not a covered error, users may miss some information or functionality of o, but they can still trigger the event handler registered to the event. On the other hand, if an event object o has a covered error, users cannot trigger the event handler registered to the event in the UI states where o has the visibility full-cover. Therefore, we identify them as a new kind of programmability errors dubbed VFs, and propose a mechanism to detect them automatically in the next section.

Algorithm 1: Building a UI State Graph

```
Input: web page, browser
Output: UIStateGraph
$V := \emptyset$; $E := \emptyset$; $Q := \langle\rangle$;
browser := load(web page, browser);
$\sigma_0$ := getUIState(browser);
$V := \{\sigma_0\}$;
foreach $(o, v) \in \sigma_0$ do
    if $v \neq$ full-cover then $Q$.enqueue($\langle o, \sigma_0 \rangle$);
while $Q \neq \langle\rangle$ do
    $(o, \sigma) := Q.dequeue()$;
    browser := goto(browser, $\sigma$);
    $e$ := getEvent($o$);
    browser := dispatchEvent(browser, $e$);
    $\sigma'$ := getUIState(browser);
    if $\sigma' \notin V$ then
        foreach $(o', v') \in \sigma'$ do
            if $v' \neq$ full-cover then $Q$.enqueue($\langle o', \sigma' \rangle$);
    $V \cup= \{\sigma'\}$; $E \cup= \{\langle \sigma, e, \sigma' \rangle\}$;
return $(V, E)$;
```

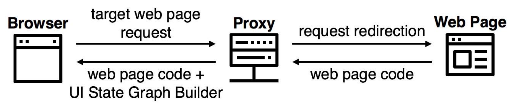
IV. Automatic Detection of Visibility Faults

This section presents a tool that analyzes HTML5 web pages and reports possible VFs in them. As Figure 4 illustrates, the tool consists of two parts: *UI State Graph Crawler* and *Bug Detector*. For a given target page and a set of browsers with different sizes, UI State Graph Crawler builds UI state graphs for all the browsers, and passes them to Bug Detector, which then analyzes the UI state graphs and detects possible VFs.

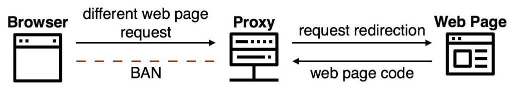
A. UI State Graph Crawler

UI State Graph Crawler consists of two parts: *UI State Graph Builder* that takes a web page and a browser, and *Proxy* that manages connections between the page and the browser.

1) UI State Graph Builder: For a given web page and a browser, UI State Graph Builder collects the UI states and transitions between them to build their UI state graph. In order



(a) Inserting UI State Graph Builder to the target web page



(b) Blocking page transitions from the target web page to others

Fig. 5: Two roles of Proxy

to collect UI states, it collects all the event objects and their visibility in the web page and the browser. Also, to collect all the transitions between UI states, it triggers the collected events to consider all the resulting UI states after the events.

Algorithm 1 describes how UI State Graph Builder constructs UI state graphs using the following helper functions:

1) load: For a web page and a browser, loads the page and *DOM API wrappers* that collect dynamic information of the page on the browser.
2) getUIState: For a browser, builds its UI state by collecting a set of event objects and their visibility.
3) goto: For a browser and a UI state, finds a path from the current UI state of the browser to the given UI state, and activates each transition in the path. Such a path always exists by Property 1 and Property 2.
4) getEvent: From a event object, extracts an event.
5) dispatchEvent: For a browser and an event, triggers the event which may change DOM elements of the browser.

To collect event objects, UI State Graph Builder keeps track of DOM API function calls that register or remove event handlers such as `addEventListener` and `removeEventListener`. It monitors such behaviors by defining wrappers of DOM APIs, which record which event handlers are registered to or removed from which DOM element and event type. For example, consider the following code:

```
let F = EventTarget.prototype.addEventListener;
EventTarget.prototype.addEventListener =
  function () {
    let target = this;
    let t = arguments[0];
    let f = arguments[1];
    if (f !== undefined) {
      // an event handler f of an event type t
    }
    return F.apply(this, arguments);
  }
```

which is a simplified version of the wrapper function of `addEventListener`. It records the fact that "an event handler `f` of an event type `t` is attached to a target element." Such wrappers are loaded before executing web pages by load. The collected information about event handlers are used when UI State Graph Builder collects event objects. When it builds UI states using getUIState, it calculates the back-to-front order among DOM elements to get the visibility of event objects. Using the tree structure of DOM elements, and their positions, sizes, and other properties provided by the browser, UI State Graph Builder determines the visibility of each event object. Starting from the initial UI states, it collects event objects that are not full-cover in the UI states because users cannot activate full-cover event objects. For each event object collected, it triggers the event, collects the resulting UI states, and repeats until no new UI states are left.

Algorithm 2: Finding Faults

```
Input: graphs: $\mathcal{P}(UIStateGraph)$
Output: $\mathcal{P}(EventObject \times UIState \times UIState) \times$
        $\mathcal{P}(EventObject \times UIState \times UIState)$
$faults := \emptyset$; $errors := \emptyset$ ;
foreach $(V_1, E_1), (V_2, E_2) \in graphs$ do
    $P := \{(\sigma_1, \sigma_2) \mid \sigma_1 \in V_1 \wedge \sigma_2 \in V_2 \wedge \sigma_1 \equiv \sigma_2\}$ ;
    foreach $(\sigma_1, \sigma_2) \in P$ do
        foreach $(o_1, v_1) \in \sigma_1, (o_2, v_2) \in \sigma_2$ do
            if $o_1 = o_2 \ \wedge \ v_1 \neq v_2$ then
                if $v_1 =$ full-cover $\vee$ $v_2 =$ full-cover then
                    $errors \cup= \{\langle o_1, \sigma_1, \sigma_2 \rangle\}$;
                else
                    $faults \cup= \{\langle o_1, \sigma_1, \sigma_2 \rangle\}$;
return faults, errors;
```

2) Proxy: Since HTML5 web pages may request resources like images even after the pages have been fully loaded, possibly changing layouts, we run UI State Graph Builder on a browser connected to network using Proxy. In addition, Proxy has two special roles as shown in Figure 5; it inserts UI State Graph Builder code to the target web page, and it prohibits transitions to different web pages from the target web page.

Figure 5(a) illustrates that when a browser requests the source code of a web page for the first time, Proxy intercepts the request, receives the requested page, and returns the web page with inserted UI State Graph Builder to the browser. In order to load wrappers for DOM APIs before executing the target web page, Proxy inserts the wrappers and relevant code right after the `<head>` tag of the HTML document.

Figure 5(b) shows that when a browser requests the source code of a different page from the current page, Proxy rejects the request since we defined the UI state graph and visibility faults based on a single page. Since Proxy cannot determine whether a given request is to change to another page or nor before it receives the request response, it intercepts the request, receives the requested page, and detects web page changes.

B. Bug Detector

Taking UI state graphs as inputs, Bug Detector finds VFs in the target page. It simply collects all the pairs of comparable UI states in the graphs, and compares UI states in each pair to detect VFs. Algorithm 2 describes how Bug Detector detects inconsistent faults in *faults* and covered errors in *errors*.

Bug Detector reports detected VFs with their *"replay" descriptions*, which describe how to reproduce the detected VFs by specifying (1) which event objects have the VFs (2) in which states. Thus, replay descriptions describe how to identify event objects of interest in a given web page.

TABLE I: Evaluation of HTML5 web pages that have visibility faults (VFs) or responsive layout failures (RLFs).

Page Name (Rank)	**Size (KB)**	**RQ1: Precision**						**RQ3: Comparison with ReDeCheck**	
		IF (#)		**CE (#)**		**Average VF (#)**		**RLF (#)**	**Common (#) (VF/RLF)**
		TP (#)	**FP (#)**	**TP (#)**	**FP (#)**	**IF (#)**	**CE (#)**		
AliExpress (51) [19]	6.1	1	0	0	0	1	0	0	0
Imgur (65) [20]	14.4	17	0	1	0	17	1	3	0
Diply (66) [21]	8.3	3	0	0	0	3	0	3	0
Naver (70) [22]	8.0	1	0	0	0	1	0	1	1 (1 / 1)
Walmart (102) [23]	77.8	8	0	0	0	8	0	31	0
W3Schools (104) [24]	12.1	**16**	**1**	**80**	**1**	**35.8**	**69.2**	0	0
AdF.ly (113) [25]	10.3	4	0	1	0	4	1	11	1 (1 / 1)
SaveFrom.net (133) [26]	10.7	7	0	1	0	7	1	10	1 (6 / 1)
Dailymotion (135) [27]	6.8	2	0	7	0	2	7	11	6 (6 / 9)
AWS (148) [28]	12.4	24	0	37	0	24	37	92	0
MediaFire (158) [29]	13.3	2	0	1	0	2	1	8	1 (1 / 1)
DeviantArt (168) [30]	71.6	9	0	14	0	9	14	34	0
3-Minute Journal [31]	1.7	1	0	0	0	1	0	11	0
Accountkiller [32]	16.0	0	0	0	0	0	0	149	0
Airbnb [33]	125.9	3	0	0	0	**3.8**	**0.4**	–	–
BugMeNot [34]	1.2	3	0	1	0	3	1	5	4 (4 / 4)
Cloudconvert [35]	7.9	5	0	11	0	**5.4**	**11.0**	2	0
ConsumerReports [36]	26.1	16	0	0	0	**14.4**	**3.6**	23	2 (15 / 2)
CoveredCalendar [37]	6.1	2	0	0	0	**2.0**	**0.2**	5	0
Dictation [38]	11.6	3	0	0	0	3	0	1	0
Duolingo [39]	13.0	0	0	0	0	0	0	3	0
Honey [40]	22.7	82	0	6	0	**82.8**	**5.6**	–	–
Hotel WiFi Test [41]	9.5	1	0	0	0	**1.4**	**1.4**	1	0
Mailinator [42]	6.5	2	0	2	0	2	2	1	0
MidwayMeetup [43]	4.3	1	0	1	0	1	1	3	1 (1 / 1)
Ninite [44]	7.8	0	0	1	0	**0.2**	**1.0**	1	0
PDFescape [45]	11.1	1	0	6	0	1	6	9	0
PepFeed [46]	8.4	0	0	2	0	0	2	14	0
Pocket [47]	6.3	3	0	0	0	3	0	5	0
Rainy Mood [48]	54.1	**0**	**3**	**0**	**0**	**1.2**	**1.0**	0	0
RunPee [49]	12.6	1	0	0	0	1	0	0	0
Top Documentary Films [50]	10.7	2	0	3	0	**2.0**	**2.8**	11	0
What Should I Read Next [51]	2.9	1	0	0	0	1	0	2	0
Will My Phone Work [52]	9.7	**6**	**3**	**0**	**1**	**9.6**	**0.8**	3	0
Zero Dollar Movies [53]	9.5	20	0	60	0	**20.0**	**59.8**	0	0
Total		247	7	235	2			453	17 (35 / 20)

In addition, for a given event object with a VF, replay descriptions specify the viewport width of the browser used to build the corresponding UI state graph and a path from the initial UI state to a UI state that contains the given faulty event object, which is a sequence of dispatched events. Note that Property 1(1) ensures that there always exists a replay description for any UI state.

V. Evaluation

In this section, we evaluate the effectiveness of our tool using 35 real-world HTML5 web pages in terms of precision, performance, and comparison with the closest related work, ReDeCheck [54]. We call our tool VFDetector.

- **RQ1. How precisely does VFDetector detect visibility faults in real-world web pages?** We manually investigated reports by VFDetector to identify true positives.
- **RQ2. How efficiently does VFDetector detect visibility faults in real-world web pages?** We measured execution time of VFDetector in building UI state graphs for the web pages and in gathering VFs from the UI state graphs.
- **RQ3. How similar or different are visibility faults and responsive layout failures (RLFs) from ReDeCheck?** We compared the VFs reported by VFDetector and RLFs reported by ReDeCheck.

A. *Target Pages*

In order to collect HTML5 web pages that have VFs, we inspected web pages in the Alexa Global top 300 sites[2]. We visited their main web pages using developer tools of the Chrome browser [55]. From the rank 1, we visited each web page with a browser of the viewport width 320 pixels. If a rendered web page displays any suspicious elements that are candidates of VFs, we checked the page with a browser of the viewport widths 768 pixels and 1200 pixels.

To avoid redundant or duplicate analysis, we excluded web pages that meet one of the following conditions:

1) when a web page on a small screen is a scaled-down version of the ones on a large screen, since it is not "responsive";
2) when web pages provide "similar" services for different countries like `google.com` and `google.co.in`, where we analyzed only the highest rank;
3) when web pages are subdomains of another page like `service.tmall.com`, a subdomain of `tmall.com`; and
4) when web pages disallow accesses, which are often in the "Adults" category.

[2]Accessed on February 7, 2017.

We collected 12 HTML5 pages that have VFs out of 300 sites inspected, and adapted 23 pages from ReDeCheck [54] as summarized in Table I. For example, according to Alexa, the page AdF.ly has the rank 113. It has one covered error because its "Login" button is fully covered by a translucent element on browsers with the viewport width of 320 pixels, where the "Login" button is visible to users but users cannot trigger any events registered to the button. On the contrary, users can trigger events registered to the button on browsers with the viewport width of 768 pixels or 1200 pixels. We reported this layout fault to the support team of AdF.ly, and received a positive response to fix the fault. Out of 26 web pages used for ReDeCheck, we excluded three: Days Old which had an infinite loop in an event handler, StumbleUpon which was not accessible, and Usersearch which was not responsive in the range of viewport width less than 980 pixels.

B. Implementation and Experiment Setup

VFDetector makes the following implementation decisions. It focuses on only the `click` event because most event objects we observed were `click` event objects, and because `click` events are relatively simpler than other events like `keydown` events. VFDetector randomly chooses candidate events of transitions during building UI state graphs, which corresponds to *Q.dequeue()* in Algorithm 1. It dispatches events in every 8 to 9 seconds. As for a Proxy, we used a simple Python script on mitmproxy [56]. We assumed that every UI state in a UI state graph has a path to the initial UI state as Property 1(2) specifies. When pages do not satisfy the assumption, we exclude them from consideration. Thus, while Algorithm 1 terminates when its queue is empty, VFDetector terminates when the following condition holds: for the UI state of the current browser σ, for any (o, σ') in its queue, no path from σ to σ' exists. We ran VFDetector on a Chrome browser with the viewport width of 320, 768, and 1200 pixels. Each viewport width represents the size of browsers on mobile, tablet, and laptop in the Chrome developer tools [55].

A sample bug report of VFDetector is as follows:

```
1  ... 2 fault(s) | Error(1) Fault(1)
2  [CoveredError]: A(DOCUMENT.0.1) is Visible at
       {0#1}, Covered at {1#1}.
3  [InconsistentFault]: A(DOCUMENT.0.1.1.1) is Off-
       Screen at {1#1}, Visible at {2#1}.
4  ... How to replay each state:
5  0#1 is crawled with Browser(1200 X 917).
6       >> click BUTTON(DOCUMENT.3)
7       >> click A(DOCUMENT.0.2.1)
8  1#1 is crawled with Browser(320 X 830).
9       >> click BUTTON(DOCUMENT.3)
10      >> click A(DOCUMENT.0.2.1)
11 2#1 is crawled with Browser(320 X 830).
12      >> click BUTTON(DOCUMENT.3)
13      >> click A(DOCUMENT.0.2.1)
```

The first line shows that there exist two VFs, and the following two lines specify which event object has which visibility in which state for each visibility. The second line specifies the event object "`A(DOCUMENT.0.1)`", the `<a>` tag element that is the first child of the 0-th child of the `document` object. It also states that the event object is visible at the state `0#1` but covered at the state `1#1`. Thus, we can reproduce the covered error by showing the visibility of the event object in those states. The replay description specifies how to reproduce each state by triggering an event sequence starting from the event dispatched at the initial UI state. For example, the fifth to the seventh lines specify that we can reproduce the state `0#1` by triggering the event object `A(DOCUMENT.0.2.1)` after triggering the event object `BUTTON(DOCUMENT.3)` at the initial UI sate on a browser of the viewport width 1200 pixels.

To answer the RQs, we ran VFDetector for 5 times for each target web page. When UI State Graph Crawler does not finish in 30 minutes, we call them *timeout* cases; we call the others *timein* cases. We manually inspected all the reports with their replay descriptions and classified them into true and false positives; we consider a report for a web page is true positive if we can find its faulty event object in the web page by triggering the events in its replay description via mouse clicks.

C. RQ1: Precision

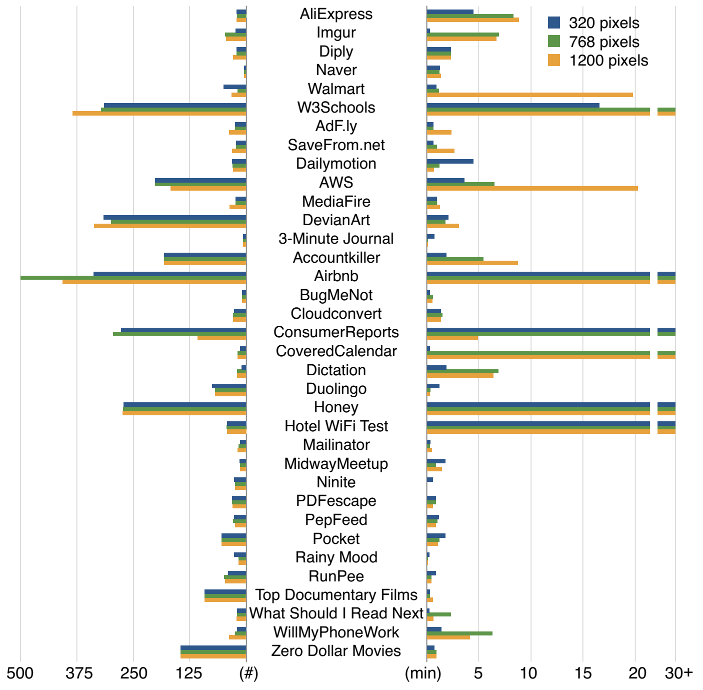
We evaluated the precision of VFDetector as summarized in Table I. **IF**, **CE**, **TP**, and **FP** stand for the numbers of detected inconsistent faults, covered errors, true positives, and false positives, respectively. The total numbers of false positive inconsistent faults and covered errors are 7 out of 254, and 2 out of 237 reports, respectively, which shows high precision.

VFDetector reported false positives only for three web pages: W3Schools, Rainy Mood, and Will My Phone Work. The false positives for the latter two pages are due to the violation of Property 2 by the pages; advertisement panels changed their contents randomly, and the twitter banner changed its data over time. However, for W3Schools, we classified "true positives" as false positives because our manual inspection could not reproduce them. While we consider a fault report true positive if we can reproduce the fault, since the faulty event object had the partial-cover visibility with a tiny area, only 2-pixel width and 48-pixel height, we could not trigger the event with a mouse click on that tiny area.

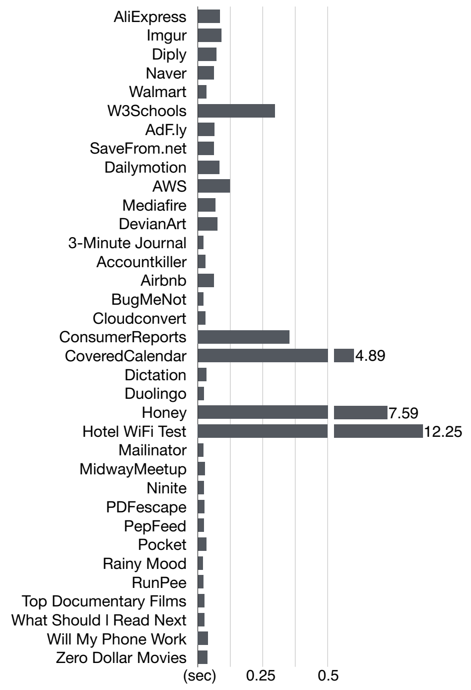
Because VFDetector may report different results for different runs, Table I also shows the average numbers of VFs for five runs. We observed two reasons for reporting different VFs for different runs. First, for timeout web pages, UI state graphs of each run may be different since Algorithm 1 involves random decision. Six web pages—W3Schools, Airbnb, ConsumerReport, CoveredCalendar, Honey, and Hotel WiFi Test—correspond to this case. Second, when web pages involve nondeterministic behaviors of event handlers, which violate the assumption of Property 2, their UI state graphs may be different for different runs. Six web pages—CloudConvert, Ninite, Rainy Mood, Top Documentary Films, Will My Phone Work, and Zero Dollar Movies—belong to this case.

D. RQ2: Performance

We evaluated the performance of VFDetector by measuring its execution time in crawling UI state graphs and comparing them to detect VFs as in Figures 6(a) and 6(b), respectively.



(a) Number of event objects in UI state graphs and time to crawl them (min)



(b) Time to compare UI state graphs (sec)

Fig. 6: Performance of VFDetector: execution time in crawling UI state graphs and comparing them for bug detection

For each target web page, Figure 6(a) shows the time took in crawling UI state graphs on the right and the number of event objects in them on the left, experimented on a browser with the viewport width of 320, 768, and 1200 pixels. The bars stretching out till the time limit denote timeout cases. The average execution time of crawling UI state graphs (the average numbers of event objects) except for the timeout cases are 1.8, 2.1, and 3.4 minutes (60.7, 52.9, and 57.0) at the viewport width of 320, 768, and 1200 pixels, respectively.

For 35 target web pages with three kinds of viewport widths, VFDetector reported 15 timeout cases from six pages: W3Schools, Airbnb, ConsumerReport, CoveredCalendar, Honey, and Hotel WiFi Test. We investigated the cases and observed two reasons. First, VFDetector fell into infinite loops while finding paths in web pages, which breaks the assumption of Property 2; five cases from CoveredCalendar and Hotel WiFi Test are due to this reason. Second, the other 11 cases from the remaining five pages had many event objects to investigate in order to crawl UI state graphs; the execution time and the number of event objects for such cases are indeed large as illustrated in Figure 6(a).

Because we excluded such web pages that do not satisfy the condition of Property 1(2) as we discussed in Section V-B, we checked whether any responsive web pages are actually excluded due to this restriction from our consideration. To find web pages containing UI states that do not have paths back to their initial UI states, we checked whether crawling UI state graphs terminated with the empty queue or not, since termination with non-empty queues denotes such cases. We found that only seven pages with timeout of 30 minutes had non-empty queues; all the other pages terminated with the empty queue, which implies that they satisfy Property 1(2), and VFDetector investigated every candidate event of transitions for them.

Figure 6(b) presents the execution time of comparing UI state graphs to detect VFs. It takes less than 0.5 seconds for each target page except for CoveredCalendar, Honey, and Hotel WiFi Test, and an average of 0.8 seconds including the exceptions, which shows that VFDetector is quite efficient.

E. RQ3: Comparison with ReDeCheck

We manually compared VFs from VFDetector with RLFs from ReDeCheck [54], and analyzed their similarity as summarized in the last three columns of Table I. **RLF** denotes the number of RLFs from ReDeCheck. For 23 subjects from [54], we inspected their publicly available reports[3]. Since the screenshots from RLFs of Airbnb and Honey are very much different from their current pages, we excluded Airbnb and Honey from inspection as marked – in Table I. We also excluded the false positive reports that the ReDeCheck authors classified as false. For the other 12 subjects that we collected from the Alexa Global top 300 sites, we ran ReDeCheck from its github repository[4]. To evaluate the similarity of VFs and RLFs, we manually compared each pair of reports to see whether they report common faults for a given page. **Common** denotes the number of common faults in a web page, and **VF** and **RLF** denote the numbers of reports corresponding

[3]http://redecheck.org/issta17/
[4]https://github.com/redecheck/redecheck

to the common faults, respectively. For example, for the SaveFrom.net page, ReDeCheck detected one RLF due to a `div` element that protrudes a viewport width, while VFDetector detected six VFs for six `button`s that are off-screen, which are children elements of the same `div` element.

Excluding Airbnb and Honey, VFDetector reported 391 true positive VFs (162 inconsistent faults and 229 covered errors) and ReDeCheck reported 453 true positive RLFs. Among them, 35 VFs detected by VFDetector correspond to 20 RLFs detected by ReDeCheck, which amount to 17 common faults. Thus, most of the true positives—356 VFs (91.0%) and 433 RLFs (95.6%)—report distinct faults. Indeed, the example in Figure 3 illustrates that because ReDeCheck does not model any event system nor the back-to-front order, it cannot detect the layout fault in the figure, which are dynamically generated by a user event. While the layout fault was caused by the back-to-front order between the dialog and the black translucent background, because ReDeCheck does not consider the order, it cannot detect the fault. On the other hand, since VFDetector inspects only a fixed set of viewport widths and event objects, it may fail to detect VFs in other viewport widths or DOM elements. For example, one RLF from the 3-Minute Journal page[5] reports a label element with the `"Jul"` text that protrudes beyond the viewport width, when it is rendered by a browser of the viewport width within 992 and 1136 pixels. However, VFDetector cannot find the fault since the label element is not an event object; moreover, since VFDetector does not inspect the viewport width ranging from 992 to 1136 pixels, it cannot detect the fault. In summary, because VFDetector and ReDeCheck have different purposes, their fault detection capabilities are complementary.

F. Threats to Validity

Our experiments have several validity threats. First, our selection of 12 pages from the Alexa Global top 300 sites may not be representative HTML5 pages. To reduce possible biases, we chose web pages of diverse sizes from various categories, and added 23 target pages from the closest related work [54].

Second, manual inspection involved in the experiments is another validity threat. Given that the edges in UI state graphs represent user interaction via the event system, manual inspection is necessary to correctly evaluate the precision of VFDetector. In order for a fair comparison of VFs and RLFs, which have different definitions and purposes, we characterized common faults in terms of concrete features such as DOM tree structure and counted them.

VI. Related Work

Selenium [57] and Sikuli [58] provide interfaces to define bugs of interests and ways to find them. On the contrary, we formally defined layout-related bugs and a mechanism to detect them without any help from developers. Hallé *et al.* [59] and Mahajan *et al.* [60] defined overlapping DOM elements as layout-related bugs. While they did not consider them as a functionality bug, we defined them as covered errors.

[5]http://redecheck.org/3-Minute-Journal-failure-3.html

Cornipickle [59] and Cassius [61] generate constraints about page layouts from HTML5 code. Because they do not handle JavaScript, they cannot detect problems due to dynamic execution of event handlers. Crawljax [62] supports simple crawling of dynamic information of web pages using browsers. It often provides infeasible states of pages during crawling such as clicking full-cover elements, and it spends much time in clicking elements that do not trigger any dynamic changes of pages. However, VFDetector does not visit infeasible UI states nor generates events that have no effects.

WebDiff [10] and CrossT [63] detect cross-browser incompatibility in visual effects and functional effects, respectively. Similarly for VFDetector, they also compare results from different browsers. However, because they check the equality of DOM tree structures and screenshots for web pages, they compare layout changes without considering that responsive web pages often make layouts differently across the range of viewport widths intentionally. The notion of comparability in our mechanism solves such a limitation by adapting the features of RWD. CrossCheck [64] is a combination of WebDiff and CrossT, but it does not consider the fact that visual differences may have effects on functionality differences, which is the main concept of our covered errors.

While ReDeCheck [9], [54] automatically finds responsive layout failures, it cannot address behaviors due to event handlers since it does not model the event system that can further cause layout-related bugs. Responsinator [65], ResponsiveDesignChecker [66], and Responsive Web Design Testing Tool [67] are tools that detect bugs in responsive web pages. They support execution of web pages in various sizes of the viewport width simultaneously, but they require developers manually inspect each web app, which is very tedious, labor-intensive, and error-prone.

VII. Conclusion

Web pages that can interact with users dynamically and change their layouts responsively to the sizes of their running devices are extremely useful and convenient, but it often leads to broken layouts or even to limited functionality. In this paper, we formally define visibility faults, which are caused by broken layouts of HTML5 web pages. We present an automated method to detect such problems, and show the effectiveness of the method by evaluating its prototype implementation on 35 real-world HTML5 web pages. Our experiments detected 247 inconsistent faults and 235 covered errors from the 35 subject web pages. We believe that our method is applicable to detect visibility faults derived by browser incompatibility or unportable design for various devices. In addition, it may be applicable to detect other kinds of faults by collecting more dynamic information via our UI State Graph Crawler such as unreachable JavaScript code at a given UI state.

Acknowledgment

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

REFERENCES

[1] "HTML5," http://www.w3.org/TR/html5/.
[2] "Document Object Model (DOM)," https://www.w3.org/DOM/.
[3] "React: A JavaScript Library for building user interfaces." https://facebook.github.io/react/.
[4] "AngularJS: One framework. Mobile and desktop." https://angularjs.io/.
[5] "Why Responsive Design Support is the Most Important Feature You Can Add To Your Website," https://www.awwwards.com/why-responsive-design-support-is-the-most-important-feature-you-can-add-to-your-website.html, July 2015.
[6] "Mobile share of organic search engine visits in the United States from 3rd quarter 2013 to 4th quarter 2016," https://www.statista.com/statistics/297137/mobile-share-of-us-organic-search-engine-visits/.
[7] E. Marcotte, *Responsive Web Design*. A Book Apart, 2014.
[8] A. Gustafson, *Adaptive Web Design*, 1st ed. Easy Readers, LLC, 2011.
[9] T. A. Walsh, P. McMinn, and G. M. Kapfhammer, "Automatic detection of potential layout faults following changes to responsive web pages (N)," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 709–714.
[10] S. R. Choudhary, H. Versee, and A. Orso, "WebDiff: Automated identification of cross-browser issues in web applications," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2010, pp. 1–10.
[11] "Keyword Research, Competitor Analysis, and Website Ranking : Alexa," https://www.alexa.com.
[12] "Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification - 9 Visual formatting model," https://www.w3.org/TR/CSS2/visuren.html, June 2011.
[13] "CSS Transforms Module Level 1," https://www.w3.org/TR/css-transforms-1/, November 2013.
[14] "Compositing and Blending Level 1," https://www.w3.org/TR/compositing-1/, January 2015.
[15] "Scalable Vector Graphics (SVG) 2, Chapter 3: Rendering Model," https://www.w3.org/TR/SVG2/render.html, September 2016.
[16] "DOM Level 3 Events," https://www.w3.org/TR/DOM-Level-3-Events/, August 2016.
[17] "HTML5: APIs in HTML documents," http://www.w3.org/TR/2011/WD-html5-20110525/apis-in-html-documents.html.
[18] "HTML5, A vocabulary and associated APIs for HTML and XHTML," https://www.w3.org/TR/html5/sections.html#the-body-element, October 2014.
[19] "AliExpress," http://m.aliexpress.com/helpcenter/index.htm.
[20] "Imgur," http://imgur.com/tos?forcedesktop=1.
[21] "Diply," http://diply.com/static/privacy.
[22] "Naver," http://nid.naver.com/user2/common/terms/terms.nhn?m=viewTermOfUseNaver.
[23] "Walmart," http://www.walmart.com/reviews/seller/4693.
[24] "W3Schools," http://www.w3schools.com/cssref/css_initial.asp.
[25] "AdF.ly," http://adf.ly/.
[26] "SaveFrom.net," http://en.savefrom.net/webmaster.php.
[27] "Dailymotion," http://faq.dailymotion.com/hc/en-us/articles/203655666-Encoding-parameters.
[28] "AWS," http://aws.amazon.com/marketplace/b/2649337011.
[29] "MediaFire," http://www.mediafire.com/help/submit_a_ticket.php.
[30] "DeviantArt," http://www.deviantart.com/morelikethis/collections/619247091.
[31] "3-Minute Journal," http://www.3minutejournal.com.
[32] "Accountkiller," http://www.accountkiller.com/en.
[33] "Airbnb," http://www.airbnb.com.
[34] "BugMeNot," http://bugmenot.com.
[35] "Cloudconvert," http://cloudconvert.com.
[36] "ConsumerReports," http://www.consumerreports.org.
[37] "CoveredCalendar," http://www.coveredcalendar.com.
[38] "Dictation," http://dictation.io.
[39] "Duolingo," http://en.duolingo.com.
[40] "Honey," http://www.joinhoney.com.
[41] "Hotel WiFi Test," http://www.hotelwifitest.com.
[42] "Mailinator," http://www.mailinator.com.
[43] "MidwayMeetup," http://www.midwaymeetup.com.
[44] "Ninite," http://ninite.com.
[45] "PDFescape," http://www.pdfescape.com.
[46] "PepFeed," http://www.pepfeed.com.
[47] "Pocket," http://getpocket.com.
[48] "Rainy Mood," http://rainymood.com.
[49] "RunPee," http://runpee.com.
[50] "Top Documentary Films," http://topdocumentaryfilms.com.
[51] "What Should I Read Next," http://www.whatshouldireadnext.com/search.
[52] "Will My Phone Work," http://willmyphonework.net.
[53] "Zero Dollar Movies," http://zerodollarmovies.com.
[54] T. A. Walsh, G. M. Kapfhammer, and P. McMinn, "Automated layout failure detection for responsive web pages without an explicit oracle," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2017, pp. 192–202.
[55] "Chrome DevTools Overview," https://developer.chrome.com/devtools.
[56] "mitmproxy," https://mitmproxy.org.
[57] "Selenium," http://docs.seleniumhq.org/.
[58] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: using GUI screenshots for search and automation," in *Proceedings of the ACM symposium on User interface software and technology (UIST)*. ACM, 2009, pp. 183–192.
[59] S. Hallé, N. Bergeron, F. Guerin, and G. Le Breton, "Testing web applications through layout constraints," in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–8.
[60] S. Mahajan, B. Li, P. Behnamghader, and W. G. Halfond, "Using visual symptoms for debugging presentation failures in web applications," in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 191–201.
[61] P. Panchekha and E. Torlak, "Automated reasoning for web page layout," in *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2016, pp. 181–194.
[62] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-Based Web Applications Through Dynamic Analysis of User Interface State Changes," *ACM Trans. Web*, vol. 6, no. 1, pp. 3:1–3:30, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2109205.2109208
[63] A. Mesbah and M. R. Prasad, "Automated cross-browser compatibility testing," in *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 2011, pp. 561–570.
[64] S. R. Choudhary, M. R. Prasad, and A. Orso, "Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications," in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2012, pp. 171–180.
[65] "Responsinator," http://www.responsinator.com/.
[66] "Responsive Design Checker," http://responsivedesignchecker.com/.
[67] "Responsive Design Testing," http://mattkersley.com/responsive/.