Automated Layout Failure Detection for Responsive Web Pages without an Explicit Oracle

Thomas A. Walsh
University of Sheffield, UK

Gregory M. Kapfhammer
Allegheny College, USA

Phil McMinn
University of Sheffield, UK

ABSTRACT

As the number and variety of devices being used to access the World Wide Web grows exponentially, ensuring the correct presentation of a web page, regardless of the device used to browse it, is an important and challenging task. When developers adopt responsive web design (RWD) techniques, web pages modify their appearance to accommodate a device's display constraints. However, a current lack of automated support means that presentation failures may go undetected in a page's layout when rendered for different viewport sizes. A central problem is the difficulty in providing an automated "oracle" to validate RWD layouts against, meaning that checking for failures is largely a manual process in practice, which results in layout failures in many live responsive web sites. This paper presents an automated failure detection technique that checks the consistency of a responsive page's layout across a range of viewport widths, obviating the need for an explicit oracle. In an empirical study, this method found failures in 16 of 26 real-world production pages studied, detecting 33 distinct failures in total.

CCS CONCEPTS

•Software and its engineering → Software defect analysis;

KEYWORDS

Responsive web design, presentation failures, layout failures.

ACM Reference format:
Thomas A. Walsh, Gregory M. Kapfhammer, and Phil McMinn. 2017. Automated Layout Failure Detection for Responsive Web Pages without an Explicit Oracle. In *Proceedings of 26th International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17),* 11 pages.
DOI: 10.1145/3092703.3092712

1 INTRODUCTION

The last decade has seen an explosion in the number and variety of devices being used to access the web [1]. As mobile-readiness increasingly drives site profitability [22] — and since web pages designed for large desktop displays are not, in general, easy to view or use on smaller screens — it is crucial for web developers to accommodate all available devices. Due to the plethora of screen sizes, from small to large phones, "phablets" and "mini" and "pro" tablets, maintaining only a single "mobile version" of a web site alongside an existing desktop version is no longer a satisfactory option [34].

Responsive web design (RWD) is a recent design and implementation approach enabling developers to build web pages that provide an equivalent user experience regardless of device size [32]. RWD enables a web page to dynamically modify its layout to adapt or "respond" to the size of a device's display, rather than requiring users to pan around pages that are too wide to fit on a smaller screen, or zoom portions that, while legible on a desktop display, are too small to read on a mobile phone. If there is more content than available space, the user should only need to scroll the page vertically [7]. Thus, in the context of RWD, *browser viewport width* is the key determinant as to how web page layout should adjust [2].

Given the clear benefits of RWD, the problem of automatically checking for *presentation failures* — visual discrepancies in the rendering of a web page that cause it to deviate from its intended appearance — is an important one. Since the aesthetics and layout of a web site have been shown to affect its perceived usability [26] and accessibility [33], boost its credibility [37], and engender user loyalty [23], it is of little surprise that visible failures in an organization's web site can lead to lost revenue [27].

However, the process of developing a responsive layout that adapts to varying display constraints introduces new possibilities for presentation problems. As viewport space tightens in unanticipated ways, web page elements may start to overlap, overspill their containers, or wrap incongruously, leading to ugly visual effects or inaccessible content. Defects in layout rules may cause elements to appear in the wrong position, be displayed when they should not be, or not be visible at all. To compound the problem, these failures can occur intermittently at different viewport widths. The fact that a responsive layout failure (RLF) may occur at only a single width effectively explodes the number of presentational states of a web page that must be checked — as it may be viewed on small smartphone displays that are as narrow as 320 pixels wide up to larger laptop or desktop displays of 1024 pixels wide or more.

Despite these problems there has been almost no previous research on automatic RLF detection. Our previous work [40] is limited to detecting differences between one responsive web page and a previous development version, presenting a technique that extracts a "responsive layout graph" (RLG) of a page that models its layout over a range of viewport widths. This approach derives an RLG for each page, reporting differences between the two graphs. However, a developer must then decide which reported differences are *intended* changes and which are *unintended* layout failures. By nature, the approach is limited to only detecting regression issues — it is not useful when a previous version of the web page is not available, nor is it easily applied if the previous version of the page is so far removed from the current one that an overwhelming number of differences are reported. Furthermore, it will not report any failures that are present in *both* versions of the web page and as such do not represent differences between the two.

While other methods for detecting presentation failures also rely on graph comparison approaches, they do so for orthogonal problems. For instance, work on detecting cross-browser issues (XBIs) [38] involves extracting a model of web page layout (i.e., an "alignment graph") of a page in two different browsers. Any difference in the graph extracted from the two different browser renderings is reported as an XBI. Work on detecting international presentation failures (IPFs) aims to find differences in the layout of a page when its text is presented in two distinct languages [17]. A graph modeling page layout is extracted for the two different languages and compared with the intent of finding layout differences. These approaches cannot handle responsive designs, since their graphs only model layout at a single viewport width. Nor can these graphs be compared for different viewport widths, since layout at these widths may vary intentionally, as per RWD principles.

A central problem in checking responsive web pages for layout failures, therefore, is the difficulty in providing a mechanism for distinguishing true failures from intended aesthetics and layouts — also known as an oracle. All of the previously discussed approaches use an alternative version of the page as an oracle — represented as a graph — but are not designed to find RLFs (in the case of XBI and IPF detection [17, 38]) or pinpoint RLFs as opposed to general changes (in the case of RWD regression checking [40]). Like current work on XBI and IPF detection, other methods for detecting presentation failures in web pages also only handle the non-responsive case, assuming a fixed viewport size and static layout. These include approaches that require a designer-provided mockup image of a web page to compare against (e.g., [30]), or the specification of layout constraints (e.g., [24]). Adapting these approaches to handle responsive designs would require many mockup images to be provided. Or, it would require new ways to express layout constraints so that they can be applied to responsive web layouts — along with the effort needed to specify them for each new page to be checked. Given this situation, current RWD development practice relies on the human functioning as the oracle, in a manual spotchecking process that may lead to RLFs being overlooked.

Addressing these concerns, we present an automated technique that can detect five types of responsive layout failure found to be prevalent in real-world production pages, without the need of an explicit oracle, such as a series of mockup images, complex layout specifications, or a graph model of the page to compare against. Our approach instead relies on implicit oracle knowledge [18] of common responsive failure types, automatically checking the layout of a responsive web page against *itself* and comparing the positioning of elements relative to one another at different viewport widths.

For example, two web page elements may overlap because of an intended graphical effect, or, because they have "collided" as horizontal screen space has decreased. Our approach differentiates the two by checking their layout behavior across consecutive viewport widths. If the elements *always* overlap, the effect is likely intended and/or easily noticed by a developer in a manual spotcheck. If the elements overlap infrequently, however, a subtle RLF is likely to be occurring. Our approach applies similar principles to detect inconsistent element wrapping; layouts that only exist for only one or two viewport widths; and intermittent protrusions of content into other elements, or out of the viewport entirely. Overall, we define four algorithms that detect these five types of RLF.

We applied our automated technique to 26 real-world production web pages. Experiments show that our approach can find failures in 16 web pages, detecting 33 distinct failures in total. Our evaluation further revealed that applying a manual spotchecking process with the assistance of currently-available tools missed between 19% and 34% of the RLFs that our technique can automatically detect.

In summary, the important contributions of this paper are:

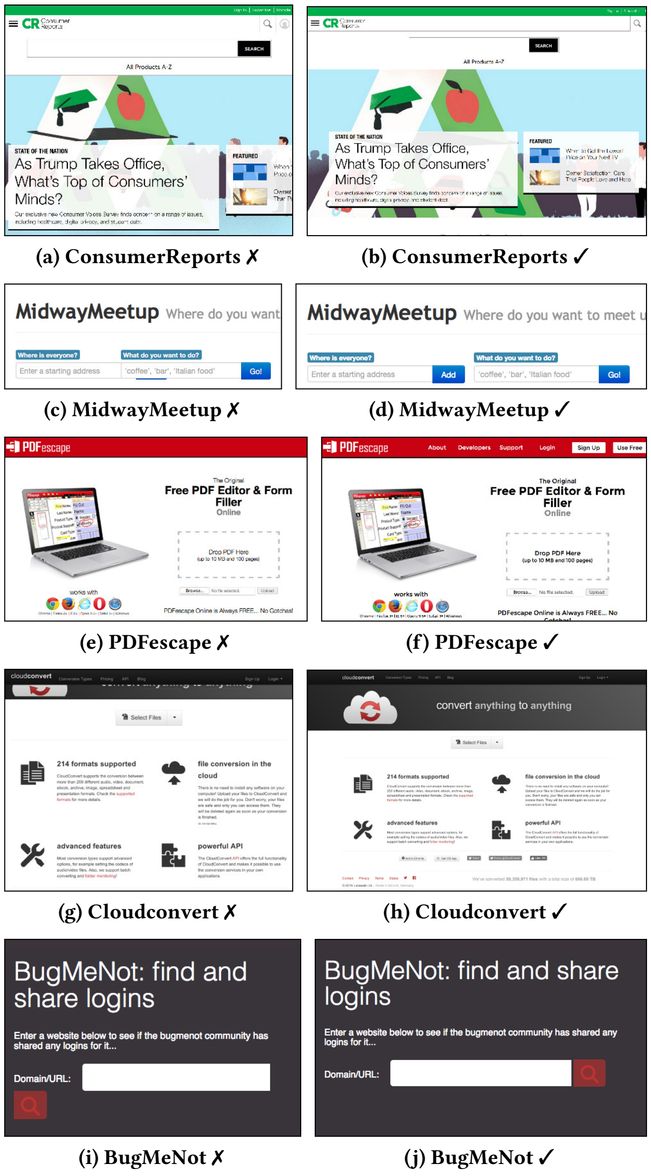
(1) A categorization of five different types of responsive layout failures (RLFs) discoverable without the need of explicit oracles.
(2) Four algorithms that can automatically detect the five types of layout failures in responsively designed web pages.
(3) An empirical study that incorporates 26 randomly selected production web pages, showing that the RLF types identified are prevalent in live sites and that our algorithms are capable of detecting them, with 33 distinct failures found in total.

2 BACKGROUND

Fluid grids, flexible media, and media queries are all core concepts of RWD that support the design of web pages that accommodate all devices and viewport widths [32], and can be implemented using HTML and cascading style sheet (CSS) code or imported through the use of an RWD framework, such as Bootstrap [6] or Foundation [9]. Fluid grids allow HTML elements to be arranged in layouts that smoothly adjust according to the viewport width, while flexible media refer to, for instance, images that stretch or shrink in size, depending on the available space. Media queries allow developers to activate specific CSS rules whenever a set of conditions regarding the user's device or browser are met [32]. For example, any CSS rules contained within the media query `@media(max-width:767px)` would activate if a user's device had a narrow screen width, while `@media(min-width:1200px)` would trigger CSS rules when the page is viewed on the wide-screen display of a desktop computer.

One well-known practice for testing responsive web sites involves the display and manual checking of a page's content on an array of physical devices with different viewport widths [36]. As testing using individual devices is a time consuming process, and a developer may not have access to all devices currently in popular use, a common strategy is to perform as much testing as possible through "spotchecking". This is a manual process in which a developer checks a web page at a few common viewport widths, often using a desktop browser. Spotchecking is supported by several tools capable of displaying a page within a customized viewport of a configurable size. Examples of such utilities include Responsinator [12], Responsive Design Checker [13], and Viewport Resizer [15]. Yet, the complexity of the HTML and CSS code needed to create a web page with a correct responsive design [19] — and the error-prone and time-consuming nature of the aforementioned approaches — still result in RLFs appearing on production web sites.

For instance, Figure 1 gives screenshots of five responsively designed web pages that contain RLFs that are emblematic of the challenges inherent in RWD, each of which was detected by the prototype tool that we present in this paper. Parts 1a and 1b of this figure highlight a responsive layout failure that was confirmed by the support staff for the *ConsumerReports* site [16]. At the wider viewport width (part 1b), the titles of the featured articles are visible. Yet, as the viewport becomes narrower, only a portion of the titles are visible (part 1a), thus limiting access to the featured reports.


MidwayMeetup
BugMeNot: find and share logins
Domain/URL:
PDFescape
Free PDF Editor & Form Filler
convert anything to anything

(a) ConsumerReports ✗ (b) ConsumerReports ✓
(c) MidwayMeetup ✗ (d) MidwayMeetup ✓
(e) PDFescape ✗ (f) PDFescape ✓
(g) Cloudconvert ✗ (h) Cloudconvert ✓
(i) BugMeNot ✗ (j) BugMeNot ✓

Figure 1: Web pages with responsive layout failures (RLFs)

3 AUTOMATIC FAILURE DETECTION FOR RESPONSIVE WEB DESIGNS

This section defines five distinct types of responsive layout failure (RLF) that are both problematic for RWD and can be identified automatically through algorithms that do not require an explicit oracle, such as an alternative reference version of the web page, mockup images, or a complex specification of layout constraints.

Instead, our approach works by automatically extracting a model of a web page's responsive layout, and then analyzing this model for potential failures by cross-checking its layout at different viewport widths. We present the web page model first (Section 3.1). We then introduce five types of RLF prevalent in responsive web pages, along with the definition of four algorithms that can be used to detect them with the web page model (Section 3.2).

3.1 Basic Concept: The rRLG

The basis of our RLF detection process is a model of a page's responsive layout that is a refinement of the Responsive Layout Graph (RLG) [40]. The RLG differs from other layout graph models of a web page (e.g., the "alignment graph" of Choudhary et al. [38] and the "layout graph" of Alameer et al. [17]) in that it models the layout of a web page over a range of viewport widths — rather than a single, static width — in order to capture its responsive design.

An RLG is automatically obtained by querying the Document Object Model (DOM) of a web page to find the HTML elements involved in the page, and their co-ordinates, at different viewport widths. The RLG organizes this information to track the dynamic visibility and relative alignment of these HTML elements as the layout of the page adjusts in relation to viewport width, in accordance with its responsive design. The "refined RLG" (rRLG) differs from our original RLG [40] in that it does not model the width of web page elements through "width constraints". While designed to trap regression issues in pages [40], width constraints do not contribute to detecting the five common types of RLF introduced in this paper; as such the rRLG does not model them.

An example rRLG is furnished by Figure 2 for the web page depicted, by wireframes, at two different viewport widths. The page involves the HTML elements div[1]–div[3], which are stacked on top of each other for narrow viewports, requiring the user to scroll to bring each into view. For wider viewports, they are aligned side-by-side, and are accompanied by a banner image, img.

An rRLG models the presentational HTML elements of a web page (i.e., the body tag and HTML tags nested within it) using a set $\mathcal{E}$. Each element $e \in \mathcal{E}$ forms a node in the graph. Edges in the graph model relationships between elements, and are represented by the set $\mathcal{R}$, where $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}$. For each given width, HTML elements are arranged into a hierarchy on the basis of the position and size of their minimum bounding rectangles as they are rendered in two-dimensional space, found by querying the DOM of the web page. An element e_2 is said to be *contained* within e_1 if the bounding rectangle of e_2 is inside that of e_1. An element e_2 is a *child* of e_1 if there is no other element e_o containing e_2 also contained by e_1. Conversely, e_1 is the *parent* of e_2. Two elements e_1 and e_2 are *siblings* (e.g., div[1] and div[2]) if they are both children of some common parent element e_p. A directed edge, $(e_1, e_2) \in \mathcal{R}$, is formed in the rRLG from e_1 to e_2 if there is at least one viewport width where e_1 is the parent of e_2, or, they are siblings.

To model variations in the layout of HTML elements across different viewport widths, each rRLG edge is associated with a set of *alignment constraints*. An alignment constraint models whether the nodes of an edge (e_1, e_2) are in a parent-child or sibling relationship for a specific range of viewport widths, along with the nature of the relative alignment of e_1 with respect to e_2 when rendered on the page. A set of attributes is used to describe this relative alignment. For example, "L" describes e_1 as aligned to the left of e_2; "R" to the right, "CJ" center-justified, and "LJ" left-justified. Formally, an alignment constraint is defined by the tuple $(amin, amax, t, P)$, where $amin$–$amax$ is an inclusive range of viewport widths for which two elements $(e_1, e_2) \in \mathcal{R}$ have the relationship denoted by $t \in \{\text{pc}, \text{s}\}$ (parent-child or sibling, respectively) and whose alignment is described by the set P of alignment attributes. There are two alignment

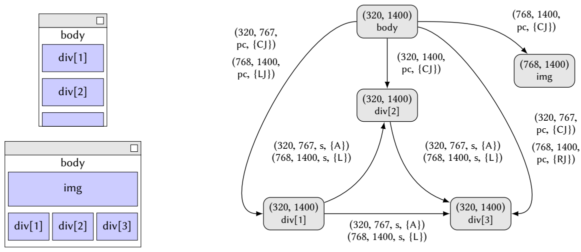


Figure 2: A wireframe example web page at two viewport widths (top and bottom left) and fragment of its rRLG (right)

constraints for the pair of elements div[1] and div[2], which label their rRLG edge in Figure 2. The initial constraint, $(320, 767, \text{s}, \{\text{A}\})$ holds between the widths 320–767 pixels as indicated by its first two values. The third value indicates that the two elements are siblings ("s"), while the final value — the set of alignment attributes — signals that div[1] is aligned above div[2], as it contains the "A" attribute. The second alignment constraint, $(768, 1400, \text{s}, \{\text{L}\})$, indicates the relative layout of the elements changes for viewport widths of 768–1400 pixels in that div[1] is now to the left of div[2].

To accommodate changing space constraints, a web page designer may choose to display HTML elements for some viewport widths while hiding them for others. To account for this, each rRLG node $e \in \mathcal{E}$ is associated with a set of *visibility constraints*. A *visibility constraint* is a pair $(vmin, vmax)$ where $vmin$–$vmax$ is an inclusive range of viewport widths for which an HTML element is displayed (i.e., defined as present in the DOM, with visibility property set to "true" and an opacity greater than zero). For the example, the img element of Figure 2 is only visible at viewport widths of 768 pixels and greater, and so its rRLG node is labeled with the visibility constraint $(768, 1400)$. All other elements are visible throughout the entire range of viewport widths modeled by this rRLG, and as such they have the constraint $(320, 1400)$.

Given VC and AC, the respective sets of visibility and alignment constraints for a web page, an rRLG is a tuple $(\mathcal{E}, \mathcal{R}, \mathcal{F}_{VC}, \mathcal{F}_{AC})$ where $\mathcal{F}_{VC} : \mathcal{E} \rightarrow 2^{VC}$ is a function mapping an element to a set of visibility constraints, and $\mathcal{F}_{AC} : \mathcal{R} \rightarrow 2^{AC}$ is a function mapping edges to individual sets of alignment constraints. Each element $e \in \mathcal{E}$ must be visible for at least one viewport width, and, for a particular viewport width, there is at most one alignment constraint that applies for each pair of web page elements $(e_1, e_2) \in \mathcal{R}$.

3.2 Common Types of Responsive Layout Failure and Algorithms for their Detection

Once an rRLG has been created for a responsive web page it can be checked for each of the five types of RLF that we introduce next along with the algorithms that can be used to detect them. The aim of each algorithm is to identify not only that a failure exists, but also which HTML elements were involved and at which particular viewport widths, to help developers diagnose the fault.

RLF 1: Element Collision. When the viewport of a responsively designed page becomes narrower, one design strategy to account for the loss of width is to move horizontally-aligned page elements closer together. As the viewport contracts, however, elements may collide into one another, causing their contents to overlap. This can result in unintended effects such as overlaid text or images, or hidden content or functional elements, thus harming page usability.

Algorithm 1: Detection of element collision & protrusion failures

In: An rRLG $(\mathcal{E}, \mathcal{R}, \mathcal{F}_{VC}, \mathcal{F}_{AC})$ for a responsive web page
Out: A set of failure reports, disseminated by calls to the REPORTFAILURE function

```
1:  for all $r = (e_1, e_2) \in \mathcal{R}$
2:      for all $(amin, amax, t, P) \in \mathcal{F}_{AC}(r)$ where $t = \text{s} \wedge \text{O} \in P$
3:          $(\ldots, P_{wider}) \leftarrow$ ALIGNMENTCONSTRAINTAT$(e_1, e_2, t, amax{+}1)$
4:          if $(\ldots, P_{wider}) \neq \bot \wedge \text{O} \notin P_{wider}$
5:              REPORTFAILURE(element-collision, $\{e_1, e_2\}$, $\{(amin, amax)\}$)
6:          else
7:              $a_1 \leftarrow$ GETANCESTORSAT$(e_1, amax + 1)$
8:              $a_2 \leftarrow$ GETANCESTORSAT$(e_2, amax + 1)$
9:              if $(e_1 \in a_2) \vee (e_2 \in a_1)$
10:                 REPORTFAILURE(element-protrusion, $\{e_1, e_2\}$, $\{(amin, amax)\}$)
```

An example is shown by Figure 1 and the *MidwayMeetup* page. At wider viewports (part 1d) space exists for the input boxes and buttons to exist side by side. Yet, when the viewport becomes too narrow, the elements collide, obscuring the left-most button (part 1c).

Algorithm: Element collisions can be detected through the rRLG by tracing pairs of elements that have the overlap alignment attribute ("O") set for one particular viewport width, but not the next. Algorithm 1 can detect such failures. It begins by iterating through all alignment constraints in the rRLG, until it finds one for a pair of elements (e_1, e_2) in a sibling relationship, and with the overlap attribute ("O") set (steps 1–2). If an alignment constraint, obtained in step 3, exists for the two elements at the wider, adjacent, viewport width to the original alignment constraint involving the overlap — and this constraint does *not* describe an overlap itself (step 4) — the issue is reported as an element-collision failure (step 5).

RLF 2: Element Protrusion. When implementing a responsive design, one concern is ensuring that, as the viewport becomes narrower, HTML elements also adapt in size so that they are still big enough to contain their contents. When elements do not resize correctly, their contents may no longer "fit" and consequently protrude into surrounding parts of the page. An example of this is shown by the *PDFescape* example of Figure 1, and its block of navigation links, displayed on the top right of the page (part 1f). As the viewport becomes narrower, the horizontal space is no longer sufficient to fit the block of links next to the logo. The links protrude out of the containing HTML element, invisibly to the user (part 1e), as the container has the CSS property "`overflow: hidden`" set. The links therefore become unclickable and the page is unusable on devices of a certain size and where viewport dimensions are fixed.

Algorithm: Element protrusion can be detected using the rRLG by checking for the changing relationship between two HTML elements across adjacent viewport widths. Normally, the elements will be in a parent-child relationship, indicating one element is contained in the other. If at narrower widths, this changes to a sibling relationship, *because* the elements are now overlapping (due to the protrusion), the original "child" element has overflown its parent. Algorithm 1 detects such failures, continuing from step 6 — where it has identified the pair of elements (e_1, e_2) as overlapping — but not as a result of an element collision RLF. If an element-protrusion failure has occurred, one of the elements (e_1 or e_2) will be a parent (or contained within some ancestor) of the other at the adjacent, wider, viewport width to the alignment constraint previously identified. The algorithm therefore retrieves the respective set of ancestors for each element at this wider viewport (steps 7 and 8). If e_1 is an ancestor of e_2 or vice versa at the wider viewport (step 9), an element-protrusion failure is reported (step 10).

Algorithm 2: Detection of viewport protrusion failures

In: An rRLG ($\mathcal{E}$, $\mathcal{R}$, $\mathcal{F}_{VC}$, $\mathcal{F}_{AC}$) for a responsive web page in the viewport width range *wmin*–*wmax*; the rRLG node representing the body element, body
Out: A set of failure reports, disseminated by calls to the REPORTFAILURE function

```
1: for all $e \in \mathcal{E}$ where $e \neq$ body
2:     $S \leftarrow \emptyset$
3:     for all $r = (e_1, e_2) \in \mathcal{R}$ where $e_2 = e$
4:         for all $(amin, amax, t, P) \in \mathcal{F}_{AC}(r)$ where $t =$ pc
5:             $S \leftarrow S \cup \{(amin, amax, t, P)\}$
6:     if $S \neq \emptyset$
7:         $L \leftarrow$ SORTBYASCENDINGMINIMUMRANGEVALUES($S$)
8:         $gmin \leftarrow wmin$
9:         while HASNEXT(L)
10:            $(amin, amax, t, P) \leftarrow$ NEXT($L$)
11:            $gmax \leftarrow amin - 1$
12:            $VR \leftarrow$ VISIBLERANGES($e$, $(gmin, gmax)$)
13:            if $VR \neq \emptyset$ then REPORTFAILURE(viewport-protrusion, $\{e\}$, $VR$)
14:            $gmin \leftarrow amax + 1$
15:        $gmax \leftarrow wmax$
16:        $VR \leftarrow$ VISIBLERANGES($e$, $(gmin, gmax)$)
17:        if $VR \neq \emptyset$ then REPORTFAILURE(viewport-protrusion, $\{e\}$, $VR$)
```

RLF 3: Viewport Protrusion. As viewport space is squeezed, elements may not only start to overflow their containers, but also start to protrude out of the page's root presentational HTML element (i.e., the body tag), thus appearing *outside* of the horizontally viewable portion of the page. The *ConsumerReports* web page of Figure 1, as introduced in Section 2, exhibits this failure type.

Algorithm: Even though viewport protrusion failures are essentially element protrusions of the body element, their detection using the rRLG is different than Algorithm 1. In an rRLG, every node has a parent, except the root node corresponding to the HTML body tag. The exception to this rule is when a web page element overflows the viewport window. At the viewport widths where this occurs, the element has no parent in the rRLG, as it is no longer contained within the rectangle defined by the body element. Elements overflowing the viewport are neither classed as siblings with the body element, since there is no containing, common parent.

Algorithm 2, for detecting viewport protrusion failures, works by traversing each HTML element in the rRLG and checking that, for all viewport widths at which it is visible, it is the child of some other node in the rRLG. This is determined by analyzing both the element's visibility constraints and relevant alignment constraints.

The algorithm begins by taking each element e and extracting alignment constraints for which e is a child (steps 1–5). These constraints are then sorted into a list through a function call (step 7) that orders alignment constraints by their minimum range value (i.e., *amin* for an alignment constraint $(amin, amax, t, P)$). This ensures that constraints appear in consecutive viewport order, and therefore, if the HTML element is displayed for all viewport widths, the end of the range of one alignment constraint is one pixel less than the start of the range for the next. If there are "gaps" between the sorted order of alignment constraints — that is, viewport ranges where the element has no parent — *and* the element is visible in these gaps (as discovered by analyzing the visibility constraints for that node) the element must have protruded out of the viewport. The loop of the algorithm (steps 9–14) finds gaps by iteratively forming a range $(gmin, gmax)$ to represent viewport widths *between* consecutive alignment constraints. This range is derived by adding one to the upper bound of the viewport range for the previous alignment constraint under consideration to form *gmin* (step 14, initially set to the minimum viewport width considered by the rRLG, *wmin*, in step 8), and one less than the lower bound of the current alignment constraint to form *gmax* (step 11). If the range represents a gap (i.e., *gmax* is greater than *gmin*, since if an alignment constraint finishes at 767 pixels and the next starts at 768 pixels, for instance, there is no gap, and $gmax = 767 < gmin = 768$), the function VISIBLERANGES will return the ranges within the gap at which the element is visible (by examining its visibility constraints) and a failure is reported if at least one such range exists (step 13). The final steps check for a gap before the end of the maximum viewport width modeled by the rRLG (denoted by *wmax*).

Algorithm 3: Detection of small-range layout failures

In: An rRLG ($\mathcal{E}$, $\mathcal{R}$, $\mathcal{F}_{VC}$, $\mathcal{F}_{AC}$) for a responsive web page; a small-range threshold *thres*
Out: A set of failure reports, disseminated by calls to the REPORTFAILURE function

```
1: for all $r = (e_1, e_2) \in \mathcal{R}$
2:     for all $(amin, amax, t, P) \in \mathcal{F}_{AC}(r)$
3:         if $amax - amin \leq thres$
4:             $existsNarrower \leftarrow$ EXISTSAT($e_1, e_2, t, amin-1$)
5:             $existsWider \leftarrow$ EXISTSAT($e_1, e_2, t, amax+1$)
6:             if $existsNarrower \wedge existsWider$
7:                 REPORTFAILURE(small-range-layout, $\{e_1, e_2\}$, $\{(amin, amax)\}$)
```

RLF 4: Small-Range Layouts. Responsively designed web sites tend to use many CSS rules, which are activated and deactivated by different media queries. More than one media query in the CSS rules may evaluate to true at the same time for a given viewport width. For instance, two rules, one activated when the viewport is over 768 pixels wide, and another activated when the viewport is below 1024 pixels, will both be activated in the range 769–1023 pixels. The logic that governs when a series of rules are "on" or "off" for a given sets of elements and viewport sizes can quickly become complex, to the point at which developers can frequently make mistakes that result in CSS rules being applied at viewport widths unintentionally. A common fault of this type occurs when developers mix the use of the `min-width` and `max-width` qualifiers to define changes in layout. For instance, a developer may encode a media query "`@media (max-width: 768px) {. . . }`", and another as "`@media (min-width: 768px) {. . . }`". Since the viewport ranges defined by both of these expressions are inclusive, *both* will be activated at the 768 pixel viewport width. This clash of media queries can lead to strange layout effects, as two sets of rules will be activated when only one set was intended. These types of failures are difficult for developers to spot, since they occur only in small sub-ranges of the entire range of viewport widths in which the page may be viewed. The *Cloudconvert* example of Figure 1 is an example of a small-range RLF. At a single viewport width, the page's top menu obscures the company's logo and slogan (part 1g).

Algorithm: Detection of small-range layouts, by Algorithm 3, involves inspecting the viewport ranges for each alignment constraint of an rRLG, and checking whether it is below a small threshold *thres* in steps 1–3 ($thres = 5$ for experiments in this paper). On finding a small-range constraint, the algorithm checks whether the same constraint exists for viewports immediately narrower and wider but with different alignment attributes (steps 4–5) — thereby revealing a brief shift in the position of elements relative to one another that may indicate a problem like a media query clash in the CSS rules.

RLF 5: Wrapping Elements. If a containing element on a web page is not wide enough to contain its children, but is still "tall" enough, or has a flexible height, horizontally-aligned elements contained within it will "wrap" to form an additional, yet undesirable, row of elements — and an unwanted presentational effect. An example of incorrect wrapping is shown in Figure 1 and the *BugMeNot*

Algorithm 4: Detection of wrapping failures

In: An rRLG ($\mathcal{E}$, $\mathcal{R}$, $\mathcal{F}_{VC}$, $\mathcal{F}_{AC}$) for a responsive web page
Out: A set of failure reports, disseminated by calls to the REPORTFAILURE function

```
1: for all $e \in \mathcal{E}$
2:    $C \leftarrow \emptyset$
3:    for all $r = (e_1, e_2) \in \mathcal{R}$ where $e_1 = e$
4:       for all $(amin, amax, t, P) \in \mathcal{F}_{AC}(r)$
5:          if $t = \text{pc}$ then $C \leftarrow C \cup \{e_2\}$
6:    $S \leftarrow \emptyset$
7:    for all $r = (e_1, e_2) \in \mathcal{R}$ where $e_1 \in C \land e_2 \in C$
8:       for all $(amin, amax, t, P) \in \mathcal{F}_{AC}(r)$ where $t = \text{s}$
9:          $S \leftarrow S \cup (amin, amax, t, P)$
10:   $L \leftarrow$ GETCHILDRANGESSORTEDBYASCENDINGMINIMUMRANGEVALUES($S$)
11:   $i \leftarrow 0$; $len \leftarrow$ LENGTH($L$)
12:   while $i < len - 1$
13:      $(rmin_c, rmax_c) \leftarrow L[i]$
14:      $(rmin_n, rmax_n) \leftarrow L[i+1]$
15:      $C_c \leftarrow$ GETCHILDRENINRANGE($e$, $C$, $(rmin_c, rmax_c)$)
16:      $C_n \leftarrow$ GETCHILDRENINRANGE($e$, $C$, $(rmin_n, rmax_n)$)
17:      $IR_c \leftarrow$ GETCHILDRENINROWS($C_c$, $(rmin_c, rmax_c)$)
18:      $IR_n \leftarrow$ GETCHILDRENINROWS($C_n$, $(rmin_n, rmax_n)$)
19:      for all $c \in C_c$
20:         if $|IR_c| \geq 2 \land c \notin IR_c \land c \in IR_n$
21:            REPORTFAILURE(wrapping, $\{e, c\}$, $(rmin_c, rmax_c)$)
22:      $i \leftarrow i + 1$
23: procedure GETCHILDRENINROWS($C_r$, $(rmin, rmax)$)
24:   $IR \leftarrow \emptyset$
25:   for all $r = (e_1, e_2) \in \mathcal{R}$ where $e_1 \in C_r \land e_2 \in C_r$
26:      for all $(amin, amax, t, P) \in \mathcal{F}_{AC}(r)$ where $t = \text{s}$
27:         if $amin \leq rmin \land amax \geq rmax$
28:            if $\text{L} \in P \lor \text{R} \in P \land \text{A} \notin P \land \text{B} \notin P$
29:               $IR \leftarrow IR \cup \{e_1, e_2\}$
30:   return $IR$
```

web page. At the wider viewport (part 1j), the magnifying glass button appears next to the search box. Yet, as the viewport becomes narrower, the button wraps to the next line (part 1i).

Algorithm: The algorithm for detecting wrapping failures infers the elements that appear to be in rows for a particular viewport range, by analyzing alignment constraints in the rRLG. With the *BugMeNot* example, the "Domain/URL" label, search box, and button elements at the wider viewport range will have alignment constraints with "L" attributes, indicating one element is to the left of the other. When the button wraps, its alignment constraint with the text box changes, with the "L" attribute replaced with the "A" (i.e., "above") label instead. More generally, if an element e appears in a row for one viewport range, and the same row exists in an adjacent, narrower range — but without e as a member, as it now appears below the row — a wrapping failure has likely occurred.

Algorithm 4 takes each element e in the rRLG, finds all of its children and extracts the "sibling" alignment constraints that exist between them (steps 1–9). The function call in step 10 takes these constraints and returns a list of viewport ranges, in ascending order. These ranges correspond to the individual ranges of each alignment constraint, unless they intersect, in which case ranges are spliced to form new successive pairs of ranges so that only one range is included in the list for each discrete viewport width. The algorithm then iterates through pairs of ranges in this list (steps 11–22). For each range, the algorithm identifies the set of elements that are children of e between the relevant viewport widths (steps 15–16), and finds which of these children are in rows (steps 17–18) through a call to the GETCHILDRENINROWS procedure.

In this procedure (steps 23–30), if an alignment constraint for two child elements e_1 and e_2 has the alignment attributes "L" or "R" (i.e., e_1 is to the left of or to the right of e_2), and *does not* also have the attributes "A" or "B" (i.e., e_1 is above or below e_2, and therefore despite being oriented to the left of e_2 is not horizontally aligned with it in a row), the two elements are added to a set of elements, denoted *IR*, that are deemed to constitute a row (step 28). If a certain viewport range contains more than two elements considered to be in a row (identified by GETCHILDRENINROWS), and one of the elements in this set is not in the corresponding set for the previous adjacent viewport range, a wrapping failure is reported (steps 20–21).

4 EMPIRICAL EVALUATION

To evaluate the effectiveness and efficiency of our technique we applied it to 26 real-world web pages in production use, with the ultimate aim of answering the following three research questions:

RQ1: How effective are our algorithms at detecting failures? How effective are our algorithms at detecting common types of responsive layout failures in responsively designed pages?

RQ2: How many failures may be detected using a "spotchecking" tool? Aided by viewport resizing tools, how effective would a "spotchecking" process be in comparison to our approach?

RQ3: How long do our techniques take to run when applied to responsively designed web pages? Is the time taken to detect layout failures reasonable for developers who will apply the technique during real-world RWD web page development?

4.1 Experimental Subjects

To answer the RQs, we collected a set of 25 real-world and active responsively designed web pages using the third-party URL selector randomusefulwebsites.com, which randomly selects a web site from its database of "useful" sites and presents it to the user. As not all web sites in this database are responsively designed, a manual step was necessary to determine if each recommended page was in the scope of our study. We loaded each page into a browser and resized the viewport window to observe any changes in its layout. If the web page was designed according to RWD principles, fluidly rearranging and resizing content to adapt to a changing viewport width, we saved it for later input into our tool. Note that the import of a popular RWD framework in the page's code was not enough to warrant inclusion in the study: an import does not imply proper usage, while the absence of an import does not mean the developers did not program their page's responsive design themselves, unaided by a framework. We repeated these steps until we had obtained a set of 25 subject web pages, to which we added the headline motivating example, *ConsumerReports*, shown in Figure 1, making 26 in total. The details of each of the web pages, which were live and operational as of January 2017, are shown by Table 1a.

4.2 Experimental Methodology

We implemented the rRLG and our algorithms into our prototype RWD checking tool called "REDECHECK" (Responsive Design Checker, pronounced "Ready Check") [39]. REDECHECK takes the address of a web page and derives an rRLG for the page by rendering it at a series of viewport widths and extracting the page's DOM within a viewport range of 320–1400 pixels, thus ensuring that consideration was given to viewport width sizes encompassing a wide variety of devices, from small mobile phones to widescreen laptops and desktops [3]. REDECHECK samples the page at a step size of 60 pixels within this range as well as at explicit breakpoint widths programmed by its developer, extracted by parsing the page's CSS rules. If the layout changes between two adjacent sample widths, REDECHECK performs a binary search between the two to localize the point at which the change occurred. REDECHECK extracts the page's DOM at each viewport width sampled, using the final set of DOMs to extract the properties of each HTML element at each viewport width needed to create an overall rRLG. It then applies each of the algorithms detailed in the previous section.

ReDeCheck uses Selenium [14] to drive and interact with an instance of the Firefox browser [11]. We ran all experiments on an iMac with 8GB RAM and OS X 10.12 as the operating system.

To answer **RQ1**, we classified each failure report produced by ReDeCheck, consisting of a set of HTML elements for one or more viewport ranges, as belonging to one of three categories: true positives (TPs), false positives (FPs), and "non-observable issues" (NOIs). TPs correctly reveal RLFs that are evident from viewing the web page at one of the reported viewport widths. That is, TPs find content erroneously overlaid on other content, content incorrectly rendered outside the viewport, or incongruous arrangements of HTML elements for specific viewport widths, indicating faults in the page's CSS rules and its accommodation of the reported viewport widths. In contrast, FPs are failure reports that are, following manual analysis, found to not reveal RLFs or any other issues with the page. FPs are scenarios where a failure has been flagged by ReDeCheck, but there is no actual identifiable problem — either visually in the design of the web page, or at the level of the DOM.

We further define a third category of result, NOIs, which do not manifest observable problems with a page, yet further analysis with diagnostic tools, such as Firebug [10], reveal potential issues at the level of the DOM. NOIs include elements that have collided or protruded their containers or the viewport at the co-ordinate level — yet since their edges are transparent or invisible, no observable issue is apparent when actually viewing the page. While NOIs do not represent serious problems, they may be useful to web developers as they can highlight potentially unknown issues with the application of the page's CSS rules. In the same way that linting tools point out program source code that might be the root of potential issues during maintenance or execution on different platforms, NOIs can point to structural issues or other factors related to CSS code that may negatively affect ease-of-modification or the ways in which pages may be rendered by different web browsers. Therefore, we report these NOIs in a category distinct from FPs.

Classifying presentation failures in web pages is necessarily a manual procedure (c.f. [17, 21, 35, 38]). Thus, the initial categorization was performed by the first author. To mitigate any potential subjectivity, decisions were reviewed by the third author. As space constraints prohibit us from discussing each individual categorization, more analysis details are available in our results archive [4].

When applied together, our detection algorithms might report a failure more than once for different RLF categories (e.g., an element collision in a small-range), or, report related failures involving common HTML elements that are likely to emanate from a single defect. To summarize ReDeCheck's ability to reveal *distinct* RLFs, we therefore manually analyzed the set of TPs for each page to determine the number of discrete, observable failures involved. Furthermore, multiple failure reports may be produced for the same viewport range. In practice, a developer would not need to examine each report individually, but rather view the web page within each distinct viewport range to check for RLFs. We therefore also record the number of distinct viewport ranges for all of the failure reports produced by ReDeCheck for each page in our study.

To answer **RQ2**, we followed a manual "spotchecking" process by analyzing each web page at the viewport width presets suggested by five popular responsive design testing tools designed to be used with a desktop browser. The first four tools — namely Kersley's [25], Responsinator [12], Responsive Design Checker [13], and Viewport Resizer [15] — were ranked at the top of a Google search for "responsive web testing tool", while the fifth is the popular "Responsive Design View" utility built into the Firefox browser's developer tools [5]. These tools incorporate 4, 10, 7, 12, and 11 preset viewport widths in the 320–1400 pixel range respectively, and 21 different widths overall — each corresponding to the portrait or landscape viewport width of a device in popular use. To complement the device-oriented presets, we selected and analyzed each page using a further 21 widths chosen at random from the same 320–1400 pixel range. The first author performed the spotchecking process using the Firefox browser, recording the viewport widths for which RLFs were found and especially noting if an RLF was previously discovered by ReDeCheck in our answer to RQ1. Ensuring correctness, the last author then checked the first author's findings.

To answer **RQ3**, we ran our tool 30 times to produce failure reports and execution timings for each subject, computing summary statistics (e.g., the median and inter-quartile range) of these values.

4.3 Threats to Validity

One validity threat for this paper's results is the extent to which they generalize to other web pages, which we mitigated by using a random URL generator to select the subjects. As Table 1a shows, the subjects vary considerably in complexity from 41 to 1469 HTML elements and from 50 to 16929 CSS declarations. The functionality and responsive layout of the chosen web pages also differ substantially, with, for instance, *Days Old* providing calendar features and *Airbnb* supporting an international e-commerce corporation.

Our methodology for answering **RQ1** and **RQ2** involved the manual analysis and classification of both the individual failures reported by our tool and the spotchecking screenshots. As with other empirical studies of presentation failures in web sites (e.g., [17, 21, 35, 38]), this task must necessarily be manual. To mitigate subjectivity affecting our results, each categorization made by the first author was verified by the last. We have put the failure reports, their classifications, screenshots, and ReDeCheck's code in an archive [4], thus allowing for their inspection by others.

The methodology for answering **RQ2** requires comparing our tool with a spotchecking process that involved looking for RLFs at both random viewport widths and the widths advocated by popular responsive design testing tools. Since this step did not involve humans — who may overlook failures during manual inspection and/or pick different viewport widths at which to spotcheck — these results may not be realistic. With that said, we judge that **RQ2**'s methodology gives a replicable insight into the number of layout failures that manual checking would detect in practice. Moreover, since the timing results for **RQ3** are subject to the interference of background operating system processes, we ran all of the experiments 30 times to minimize the possibility of bias in our results.

The use of the Firefox web browser to answer all of the research questions is another validity threat. Firefox is a popular browser that is frequently used for RWD testing and thus a good option for ensuring that the results are representative. Although other

Table 1: Experimental subject web pages and results from using the presented approach

(a) Details of the studied web pages **(b) Failure detection results**

Web Site Name	URL	# HTML Elements	# CSS Declarations	Element Collision			Element Protrusion			Viewport Protrusion			Small-Range			Wrapping			Distinct Viewport Ranges	Distinct RLFs
				TP	FP	NOI	TP	FP	NOI	TP	FP	NOI	TP	FP	NOI	TP	FP	NOI		
3-Minute Journal	www.3minutejournal.com	79	3354	-	-	1	-	-	2	8	-	-	-	1	-	-	-	-	12	2
Accountkiller	www.accountkiller.com/en	343	559	-	-	-	-	-	-	-	-	-	147	5	-	2	-	-	4	3
Airbnb	www.airbnb.com	1469	5638	-	-	1	-	-	4	-	-	4	-	2	-	2	-	-	9	2
BugMeNot	bugmenot.com	41	237	-	-	-	1	-	3	2	-	-	-	-	-	1	-	-	7	4
Cloudconvert	cloudconvert.com	907	2831	1	-	-	-	-	-	-	-	-	1	-	-	-	-	-	1	1
ConsumerReports	www.consumerreports.org	1037	6295	-	-	7	1	-	3	9	-	3	-	1	-	-	-	-	16	4
CoveredCalendar	www.coveredcalendar.com	147	5131	-	-	-	-	-	-	-	-	3	-	-	-	2	-	-	3	2
Days Old	www.daysold.com	65	1033	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	1	0
Dictation	dictation.io	194	166	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	1	0
Duolingo	www.duolingo.com	816	16929	-	-	1	-	-	-	2	-	2	-	1	-	-	2	-	7	1
Honey	www.joinhoney.com/install	460	3249	-	-	-	-	-	8	-	-	2	-	3	-	-	-	-	8	0
Hotel WiFi Test	www.hotelwifitest.com	358	4258	-	-	-	-	-	-	1	-	-	-	2	-	-	-	-	3	1
Mailinator	www.mailinator.com	279	5086	-	-	1	-	-	-	-	-	-	-	2	-	-	-	-	2	0
MidwayMeetup	www.midwaymeetup.com	85	2942	1	-	-	-	-	1	-	-	1	-	-	-	-	-	-	3	1
Ninite	ninite.com	640	2721	-	-	-	-	-	-	-	-	-	-	-	-	1	1	-	2	1
PDFescape	www.pdfescape.com	176	794	-	-	-	1	-	5	1	-	3	-	-	-	-	-	-	8	2
PepFeed	www.pepfeed.com	342	4563	4	-	3	-	-	2	1	-	1	2	14	-	1	-	-	20	6
Pocket	getpocket.com	663	5203	-	-	2	-	-	3	-	-	-	-	3	-	-	-	-	5	0
Rainy Mood	rainymood.com	88	50	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0
RunPee	runpee.com	437	7273	-	-	-	-	-	-	-	-	-	-	5	-	-	1	-	6	0
StumbleUpon	www.stumbleupon.com	283	8530	1	-	-	-	-	-	-	-	-	-	-	-	-	1	-	2	1
Top Documentary Films	topdocumentaryfilms.com	410	702	-	-	7	-	-	4	-	-	-	-	2	-	-	-	-	10	0
Usersearch	usersearch.org	865	1495	-	-	1	-	-	-	-	-	-	-	-	-	1	-	-	2	1
What Should I Read Next	www.whatshouldireadnext.com/search	111	852	-	-	-	-	-	-	-	-	2	-	-	-	-	-	-	1	0
Will My Phone Work	willmyphonework.net	781	2022	1	-	-	-	-	1	-	-	-	2	-	-	-	-	-	2	1
Zero Dollar Movies	zerodollarmovies.com	246	1802	-	-	-	-	-	-	-	-	-	-	2	-	-	-	-	2	0
Total		11322	93715	8	0	24	3	0	36	24	0	23	152	43	0	10	5	0	**137**	**33**

browsers could lead to different results, we manually confirmed, with the latest versions of both Safari and Chrome running on Mac OS X, the existence of all the distinct RLFs found for the subject web pages during the course of this paper's study. It is also worth noting that we did not compare ReDeCheck to other approaches that require oracles. This decision is justifiable for two reasons. First, the 26 subjects used in the experiments did not come with, for instance, design mockups or layout constraint specifications. Second, as we do not know the intentions of the designers of the chosen web pages, creating oracles without their advice is, in itself, a validity threat. Finally, it is important to mitigate the threats that might arise from errors in the implementation of our approach. We achieved this through automated unit and manual testing of ReDeCheck, and through manually verifying results produced with web pages that we did not include in the experimental study.

4.4 Answers to the Research Questions

RQ1: Table 1b breaks down our categorization of failure reports produced by ReDeCheck for each web page using each detection algorithm. TPs (i.e., actual RLFs) were found by each of our algorithms, with at least one TP for 16 of the 26 subjects. Given that the subjects are live and operational sites that include established commercial operations such as *Duolingo* and *StumbleUpon*, this is a compelling result, since, we surmise, such sites would have undergone an in-house testing process that missed the failures revealed by ReDeCheck. Five of the failures reported by our algorithms are the ones we used for the motivating examples. These failures were examples of content protruding off-screen for *ConsumerReports* (Figure 1a, detected by Algorithm 2); form elements obscuring one another, degrading functionality for *MidwayMeetup* (Figure 1c, detected by Algorithm 1); navigation links disappearing due to protrusion of their parent element for *PDFescape* (Figure 1e, also detected by Algorithm 1); clashing media queries and obscured layout for *Cloudconvert* (Figure 1g, detected by Algorithm 3), and wrapping elements for *BugMeNot* (Figure 1i, detected by Algorithm 4).

Further analysis of the TPs revealed that these reports conflated to 33 distinct failures in total, as reported by the "Distinct RLFs" column of Table 1b, thereby unveiling degrees of repetition in ReDeCheck's results. An extreme example is *Accountkiller*. Here, 147 small-range reports produced by Algorithm 3 are TPs, yet closer analysis revealed that all corresponded to a single distinct failure. This page involves a grid of icons, each corresponding to an rRLG node with alignment constraints connecting each pair. For one small viewport range, ReDeCheck detects elements in the grid that are not arranged consistently, leading to changes in relative alignment and some small-range constraints that cause the algorithm to trigger several individual failure reports for each. Additionally, the same distinct failure may be detected by different algorithms. For example, with *Cloudconvert*, elements collide for a small range only, triggering reports from both Algorithms 1 and 3.

Not all reports were TPs: Algorithm 3 produced small-range FPs that, in general, were the result of coincidental alignment attributes being assigned to edges in the rRLG. For example, an element might not have any particular horizontal alignment within its parent, yet for a small-range appear to be center justified, due to chance coordinate values of the child within its parent for a one-off or small series of viewport widths. In addition, Algorithm 4 produced five wrapping FPs. These were characterized by scenarios in which a set of three elements had been misclassified as a "row", for which a shift in alignment of one of the elements was identified as a failure.

The element collision and element/viewport protrusion detectors also discovered several NOIs for a number of web pages. Although non-observable, the extent to which elements had "collided" was often significant in the DOM, due to high degrees of invisible padding in the page's CSS. Element protrusion failures were often non-observable due to HTML elements having the "`overflow: hidden`" CSS property set. Subsequent changes to the page's content may result in these NOIs turning into observable RLFs — thereby representing aspects of the web page's design that the developer may want to address so as to avoid future layout failures.

Table 2: Results of the spotchecking process

Tool/Method	Total Distinct RLFs Detected
Kersley's	22 (66%)
Responsinator	22 (66%)
Responsive Design Checker	23 (69%)
Viewport Resizer	26 (78%)
Firefox Responsive Design View	27 (81%)
Random	24 (72%)
Detected using at least one tool/method	28 (85%)

Although, as Table 1b shows, ReDeCheck produces a large number of reports for some of the web pages studied, not all of which reveal distinct failures, we found that a significant number of the reports produced by the algorithms repeated the same viewport range (as reported by the "Distinct Viewport Ranges" column in the table). This is especially true when the same layout issue is reported by different algorithms, or, different but related elements are reported multiple times for the same issue at the same viewport width. In practice, a developer would not need to inspect each report individually, but rather visit the web page for each distinct viewport width range reported to confirm any failures therein. That is, the amount of effort the developer needs to invest in using ReDeCheck is not a function of the number of reports produced, nor the time potentially wasted proportional to the raw number of FPs, but instead the number of unique viewport ranges reported. As the table shows, ReDeCheck reported 137 distinct ranges for all subjects, with a total of 33 distinct RLFs actually present. Therefore, a developer would need to view a web page at no more than an average of 4.2 different viewport widths to find each actual RLF.

Finally, Table 1 does not appear to indicate a relationship between detected failures and the complexity of the web pages studied. While some pages consisting of relatively few HTML elements (e.g., *Rainy Mood* and *Days Old*) do not exhibit failures, there are others of similar low complexity (e.g., *BugMeNot* and *3-Minute Journal*) that do. Furthermore, the page involving the most failures — six for *PepFeed* — did not involve the most HTML elements. *Airbnb* and *ConsumerReports* have over 1000 HTML elements, and fewer distinct failures were detected for these pages. It is probable, however, that being created by developers at a large corporation or organization, these sites would have undergone more thorough testing.

RQ2: The spotcheck analysis revealed RLFs already discovered by our approach as part of RQ1, but no new additional failures. Table 2 reports the numbers of these distinct RLFs that were revealed at one of the viewport widths suggested by each tool, or by selecting viewport widths at random. As the table shows, 66 to 81% of these failures were revealed by the tools, depending on the set of viewport widths they suggest to check. Since these preset widths correspond to devices in popular usage, this result shows ReDeCheck is capable of revealing failures that would be displayed on these devices for users to see. Although the spotchecking tools suggest the viewport widths at which to check the web pages, the failures still need to be identified manually — in contrast, ReDeCheck relieves developers of some of this effort. The results also show that 19 to 34% of failures identified by our technique would be missed. Even if all spotchecking tools were used, complemented by a degree of further random spotchecking, five of the distinct RLFs originally identified by our approach would not be found.

RQ3: Figure 3 shows ReDeCheck's median execution times for each web page across the 30 trials. Almost all of the pages (25 of the 26) were processed by our tool within approximately three minutes

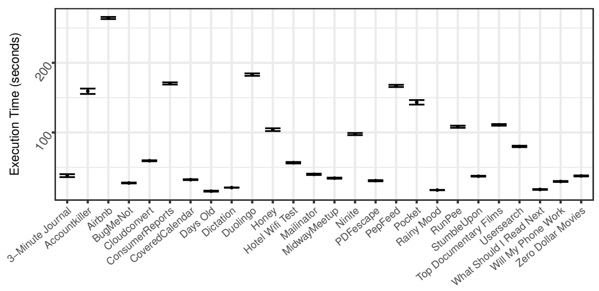


Figure 3: Execution times for ReDeCheck

When not obscured due to a small inter-quartile range (IQR), a small circle denotes the median of the timings across the 30 trials and the upper and lower hinges of the error bars respectively designate the median value added to and subtracted from the IQR of the executing timing data.

on median, with 15 pages requiring 60 seconds or less. *Airbnb* took almost 4.5 minutes — but, it is one of the most complex subject pages, as shown by Table 1a. Generally, the graph reveals that the subjects taking the most time were either some of the most complex (i.e., *Airbnb* in terms of HTML elements and *Duolingo* in terms of CSS declarations) or, yielded the most issues (i.e., *Accountkiller* and *PepFeed*), thereby incurring an additional time cost in capturing the failure screenshots and generating the annotated graphical reports (for the TPs, FPs, and NOIs), or a mixture of both web page complexity and the number of issues reported (i.e., *ConsumerReports*).

4.5 RQ Conclusions and Results Discussion

Our technique can find RLFs in live and actively maintained web pages. While it does report issues that do not correspond to observable layout problems, these tend to conflate to a smaller number of viewport ranges that a developer would need to inspect. Our results for **RQ1** show that, by using the reports of our tool, only 4.2 visual checks are required per failure for the web pages studied. However, we judge this to be a relatively low investment compared to the potential gains of discovering presentation problems with a web page that may hinder its functionality, affect a user's experience of the site, reduce users' opinion of the professionalism of the service offered by a web page, or, a combination of these factors. Nevertheless, future work will seek to reduce this figure further by exploiting potential overlap between the ranges reported to optimize the number of checks required, thereby reducing the number of FPs our algorithms produce, while also automatically distinguishing observable from non-observable issues. These steps will better enable developers to prioritize visible problems.

The results for **RQ2** show that the popular, yet manual, approach of "spotchecking" detects only between 66 and 81% of the RLFs detected by ReDeCheck, providing empirical support for the benefits of our automated technique that does not require an oracle.

The results for **RQ3** show that our current prototype is fast, completing its analysis in less than 60 seconds for the majority of the pages and at most 4.5 minutes for the most complex. Given that the tool does not need to be frequently re-run, we judge that its performance would be acceptable to practicing web developers.

Our appraisal of the prototype tool does not include the possibility of false negatives: We do not know if, for the pages studied, ReDeCheck missed failures. Anecdotally, our analysis of the chosen subjects did not discover further failures that ReDeCheck had not reported. Yet, our algorithms specialize in finding subtle failures, as evidenced by the detection of 33 on active web pages.

5 RELATED WORK

To our knowledge, there has been no research on automated checking of responsive web pages for presentation failures, except our own work on identifying differences between two versions of a responsive page [40]. This work introduced the RLG, of which the rRLG (c.f. Section 3.1) is a simplified version. With our prior technique, the RLG of a previous developmental version of a responsive page is compared with the RLG of the current version, in order to fully identify all changes to the page. While this comparison will likely identify intentional changes to the page's layout, unintentional side-effects of these changes are also highlighted. This is useful as responsive designs are hard to maintain, and changes to one part of the design specific to one viewport width can easily have unexpected knock-on effects on page layout at other widths. Yet, the technique requires a prior version of the page in order to function — and it must be one that is not so far removed from the current version that an overwhelming number of differences are identified, thus requiring manual inspection. Furthermore, the technique, implemented into an earlier version of ReDeCheck, does not preclude the possibility that an RLF could be present in *both* versions of the page, and as such not included in the list of differences.

Other techniques also use graphs modelling web page layout to identify different types of presentation failures. The "alignment graph" (AG) of Choudhary et al. [38] models layout with the aim of detecting cross-browser issues (XBIs) — differences in layout of a page when rendered in different browsers. The technique computes two graphs: an AG of page layout when rendered in a reference browser that represents the page's "correct" layout, and an AG of the same page when rendered in an alternative browser. The technique compares the two AGs, reporting any differences between the two as XBIs. The "layout graph" (LG) of Alameer et al. [17] models the layout of a page with the aim of discovering international presentation failures (IPFs), or differences in layout when a page is presented in another language. IPFs occur due to differences in space needed to render textual content in different languages: if a segment of text is longer than expected it may overspill its container, causing unwanted visual effects. The technique takes the LG of a page in a reference language, again representing "correct" layout, and automatically compares it with an LG of the page presented in an alternative language. Any differences are reported as IPFs. Unlike the RLG/rRLG, both the AG and the LG model the layout of a web page at a single viewport width — that is, they do not capture a page's potentially different layout over a range of viewport widths, thus making them unsuitable for detecting RLFs.

One aspect that all prior methods have in common with each other is the modelling of a "correct" or reference layout with a graph that is then compared to another graph for an alternative version of a page. With these methods, differences between two graphs correspond to likely presentation failures. That is, there is a reference layout that functions as an "oracle" for the technique concerned. This is different from this paper's method that does not require an explicit "oracle" or comparison layout. Instead, it checks a graph for internal consistency, using four different algorithms that aim to detect five different types of RLF that frequently occur in RWD and on web pages in production use. In other words, this paper is connected to prior work leveraging implicit oracles [18].

The use of an explicit oracle — that is not another graph but instead a reference image or specification — is also common in other works for detecting presentation failures. For example, the work of Mahajan and Halfond compares the rendering of a page with an oracle image using image comparison techniques [29]. This concept was later implemented in the WebSee tool [30]. Other tools, such as FieryEye, build on WebSee to both detect and localize presentation failures in a web page [28, 31]. All of these tools are similar to this paper's method since they find failures in a web page's presentation. However, both WebSee and FieryEye analyze a graphical mockup of the page under test at a specific viewport width — a strategy requiring an explicit oracle for all of the viewport widths if used to check a responsive web page. Alternatively, the Cornipickle tool requires the tester to specify a web page's layout properties before testing commences [24]. That is, the "oracle" is a formal specification. The Cornipickle approach also defines some common types of layout failures — but, in contrast to this paper's five different types of responsive layout failures, Cornipickle's are static in nature and thus do not require the cross-checking of a page at different viewport widths. Unlike tools such as WebSee, FieryEye, and Cornipickle, this paper's technique does not need a mockup or an intended layout specification to detect RLFs.

Finally, there are many tools that support manual developer checking of responsive pages, such as those introduced in Section 4. For instance, multi-screenshot tools (e.g., [12, 13, 25]) showcase a web page at a few common viewport widths, while others (e.g., [5, 15]) allow the tester to resize the viewport to a custom size. As shown in Section 4, these methods overlook layout failures that the presented method can detect. Also, in contrast to this paper's automated checkers that create annotated graphical failure reports, all of these tools have another limitation: the tester must inspect each screenshot, a process that is manual and error-prone. While Fighting Layout Bugs detects some types of layout failures [8], it only checks static layout properties and thus, unlike this paper's method, is not applicable to the testing of responsive pages.

6 CONCLUSIONS AND FUTURE WORK

Responsive web design (RWD) advocates for the creation of web pages with an enhanced user experience across many viewport widths [32]. Since it is challenging to implement a web page in accordance with RWD principles [20], our prior work presented a regression checking technique that compared two models of a page and presented any differences [40]. Focused on handling problems that are orthogonal to that of checking responsive web pages, other prior works (e.g., [17, 38]) did not employ oracles designed for checking multiple viewports. In contrast, this paper's automated method leverages implicit oracle information to detect responsive layout failures. Experiments show that it is effective: along with finding 2 or more failures in 8 pages, it discovered 6 failures in one web page, detecting a total of 33 different failures across 26 subjects.

In future work, we will enable our method to exploit the potential overlap between viewport ranges, thereby reducing the number of checks and false positives. We will also enhance the tool to handle dynamic pages that use JavaScript. Finally, we will extend the experiments by including more subjects and having external developers inspect the failure reports and spotchecking screenshots.

REFERENCES

[1] Android device fragmentation. http://opensignal.com/reports/2015/08/android-fragmentation/.
[2] Responsive web design — The viewport https://www.w3schools.com/css/css_rwd_viewport.asp.
[3] Know your mobile device (pixel-ratio, CSS width, features). http://mydevice.io/devices.
[4] ReDeCheck tool and ISSTA results archive. http://redecheck.org/issta17/.
[5] Firefox developer tools: Responsive design mode. https://developer.mozilla.org/en-US/docs/Tools/Responsive_Design_Mode.
[6] Bootstrap: Responsive front-end framework. http://getbootstrap.com/.
[7] Creative bloq: Web design trends 2015-16: the long scroll http://www.creativebloq.com/web-design/web-design-trends-2015-16-long-scroll-81516343.
[8] Fighting layout bugs. https://code.google.com/archive/p/fighting-layout-bugs/.
[9] Foundation: Responsive front-end framework. http://foundation.zurb.com/.
[10] Mozilla Firebug. http://getfirebug.com.
[11] Mozilla Firefox web browser. http://getfirefox.com.
[12] Responsinator. https://www.responsinator.com/.
[13] Responsive design checker. http://responsivedesignchecker.com.
[14] Selenium: Web browser automation. http://www.seleniumhq.org/.
[15] Viewport resizer. http://lab.maltewassermann.com/viewport-resizer/.
[16] Personal communication: Confirmation of responsive layout fault in the home page of Consumer Reports, January 2017.
[17] A. Alameer, S. Mahajan, and W. G. J. Halfond. Detecting and localizing internationalization presentation failures in web applications. In *Proceedings of the 9th International Conference on Software Testing, Verification, and Validation*, 2016.
[18] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *Transactions on Software Engineering*, 41(5), 2015.
[19] Z. D. Blog. Five reasons not to use Twitter Bootstrap http://www.zingdesign.com/5-reasons-not-to-use-twitter-bootstrap/.
[20] C. Bloq. 8 RWD problems (and how to avoid them) http://www.creativebloq.com/rwd/responsive-design-problems-12142790.
[21] S. R. Choudhary, H. Versee, and A. Orso. WebDiff: Automated identification of cross-browser issues in web applications. In *Proceedings of the 26th International Conference on Software Maintenance*, 2010.
[22] R. Contartesi. Eight reasons why responsive web design will increase profit for your business. http://www.business2community.com/web-design/8-reasons-responsive-web-design-will-increase-profit-business-01488386.
[23] D. Cyr, M. Head, and A. Ivanov. Design aesthetics leading to M-loyalty in mobile commerce. *Information & Management*, 43(8), 2006.
[24] S. Hallé, N. Bergeron, F. Guérin, G. Le Breton, and O. Beroual. Declarative layout constraints for testing web applications. *Journal of Logical and Algebraic Methods in Programming*, 85, 2016.
[25] M. Kersley. Responsive design testing. http://mattkersley.com/responsive/.
[26] S. Lee and R. J. Koubek. The effects of usability and web design attributes on user preference for e-commerce web sites. *Computers in Industry*, 61(4), 2010.
[27] W. Li, M. J. Harrold, and C. Görg. Detecting user-visible failures in AJAX web applications by analyzing users' interaction behaviors. In *Proceedings of the 25th International Conference on Automated Software Engineering*, 2010.
[28] S. Mahajan, K. B. Gadde, A. Pasala, and W. G. J. Halfond. Detecting and localizing visual inconsistencies in web applications. In *Proceedings of the 23rd Asia-Pacific Software Engineering Conference*, 2016.
[29] S. Mahajan and W. G. J. Halfond. Finding HTML presentation failures using image comparison techniques. In *Proceedings of the 29th International Conference on Automated Software Engineering*, 2014.
[30] S. Mahajan and W. G. J. Halfond. WebSee: A tool for debugging HTML presentation failures. In *Proceedings of the 8th International Conference on Software Testing, Verification and Validation*, 2015.
[31] S. Mahajan, B. Li, P. Behnamghader, and W. G. J. Halfond. Using visual symptoms for debugging presentation failures in web applications. In *Proceedings of the 10th International Conference on Software Testing, Verification and Validation*, 2016.
[32] E. Marcotte. *Responsive Web Design*. A Book Apart, 2014.
[33] G. Mbipom and S. Harper. The interplay between web aesthetics and accessibility. In *Proceedings of the 13th International Conference on Computers and Accessibility*, 2011.
[34] J. McMillen. 10 reasons your website needs to be mobile optimized. http://blog.teamtreehouse.com/10-reasons-website-needs-mobile-optimized.
[35] A. Mesbah and M. R. Prasad. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering*, 2011.
[36] D. Montague and L. Hogan. Building a device lab. http://buildingadevicelab.com/.
[37] D. Robins and J. Holmes. Aesthetics and credibility in web site design. *Information Processing & Management*, 44(1), 2008.
[38] S. Roy Choudhary, M. R. Prasad, and A. Orso. X-PERT: Accurate identification of cross-browser issues in web applications. In *Proceedings of the 35th International Conference on Software Engineering*, 2013.
[39] T. A. Walsh, G. M. Kapfhammer, and P. McMinn. ReDeCheck: An automatic layout failure checking tool for responsively designed web pages. In *Proceedings of the International Conference on Software Testing and Analysis (Demonstration Papers)*, 2017.
[40] T. A. Walsh, P. McMinn, and G. M. Kapfhammer. Automatic detection of potential layout faults following changes to responsive web pages. In *Proceedings of the 30th International Conference on Automated Software Engineering*, 2015.