# Automatic Visual Verification of Layout Failures in Responsively Designed Web Pages

Ibrahim Althomali
University of Sheffield, UK

Gregory M. Kapfhammer
Allegheny College, USA

Phil McMinn
University of Sheffield, UK

***Abstract*—Responsively designed web pages adjust their layout according to the viewport width of the device in use. Although tools exist to help developers test the layout of a responsive web page, they often rely on humans to flag problems. Yet, the considerable number of web-enabled devices with unique viewport widths makes this manual process both time-consuming and error-prone. Capable of detecting some common responsive layout failures, the REDECHECK tool partially automates this process. Since REDECHECK focuses on a web page's document object model (DOM), some of the issues it finds are not observable by humans. This paper presents a tool, called VISER, that renders a REDECHECK-reported layout issue in a browser, adjusting the opacity of certain elements and checking for a visible difference. Unless VISER classifies an issue as a human-observable layout failure, a web developer can ignore it. This paper's experiments reveal the benefit of using VISER to support automated visual verification of layout failures in responsively designed web pages. VISER automatically classified all of the 117 layout failures that REDECHECK reported for 20 web pages, each of which had to be manually analyzed in a prior study. VISER's automated manipulation of element opacity also highlighted manual classification's subjectivity: it categorized 28 issues differently to manual analysis, including three correctly reclassified as false positives.**

## I. INTRODUCTION

Given the variety of web-enabled devices, including phones, tablets, laptops, and desktops, web developers can no longer maintain a single "mobile version" of a web site alongside a standard desktop version [1]. Instead, web developers must fully accommodate the wide variety of devices used to view their sites. Responsive Web Design (RWD) is a design and implementation paradigm enabling developers to build web pages that provide an equivalent user experience across different devices [2]. RWD enables a web page to dynamically adapt its layout by "responding" to the viewport width of the browser running on a device. Rather than requiring users to pan around or zoom on a web page, a properly responsively designed page only requires a user to vertically scroll [3].

Even though RWD helps to address the challenges of dealing with different viewport widths, it can introduce new types of presentational layout failures, referred to as "responsive layout failures" (RLFs) [4]. As the viewport width changes, web page elements can start to overlap one another, protrude from their containing elements, or disappear off the edge of the viewable portion of the page. At best, this makes for a poor presentation of the page, leading to lost credibility [5] and decreased user loyalty [6]; at worst it can lead to critical parts of the web application being inaccessible or unusable [7].

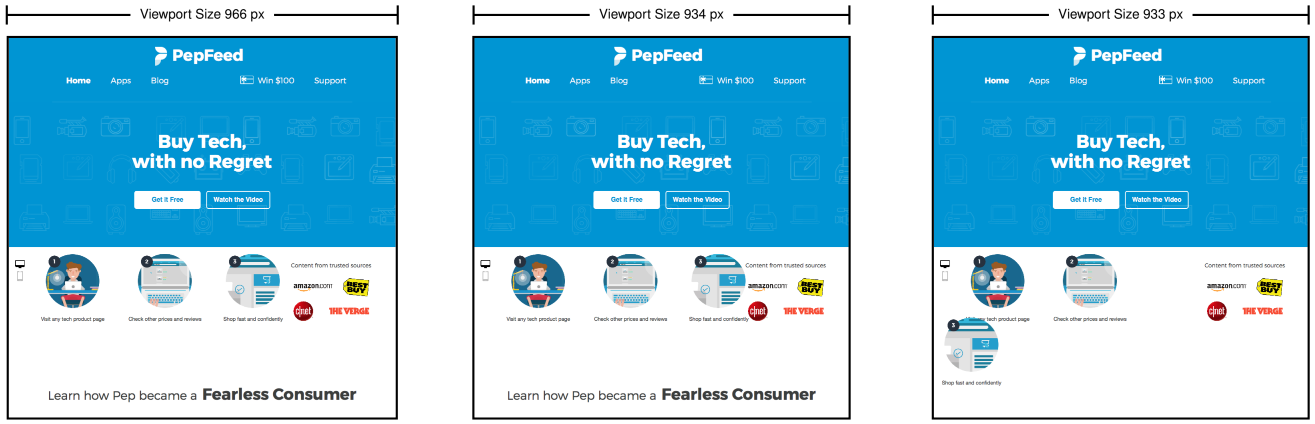
In addition to the "Responsive Design Mode" in the developer tools of web browsers like Firefox and Chrome, other tools exist to help developers test their responsive designs. For instance, viewport resizers (e.g., [8]–[10]) automatically resize browsers to common viewport widths used by web-enabled devices, conveniently allowing developers to see how their content is rendered. However, all of these tools still require a human to manually identify problems on the page.

The REDECHECK tool [11] helps a developer to identify several key types of RLFs, such as web page elements that are separated at one viewport width but then appear to collide with one another at a narrower viewport size. However, REDECHECK's analysis is limited to the document object model (DOM) of the page — a data structure which, among other things, stores the dimensions of each HTML element and how it is laid out on a page [12]. As a result, many of the issues that REDECHECK detects may not actually be observable in practice. For example, while the bounding boxes of two elements may overlap on a page, their backgrounds may be transparent and their respective content non-overlapping.

Human verification of many reported layout issues can be a time consuming, inconsistent, and error-prone task. The developer must first decode the report produced by REDECHECK, recreate the environment by starting the same web browser, navigate to the page, set the same viewport size, scroll to the graphical element in question, and visually inspect the potential failure. Each layout failure reported by REDECHECK has a viewport range: the minimum viewport width at which the issue starts to occur and the maximum viewport width at which it is still present. Since REDECHECK is DOM-based, it is possible for a failure to be observable in some parts of this range and non-observable in others. This means that it is important for the humans who verify a layout failure to identify and inspect the appropriate viewport width(s).

This paper presents "VISER" (VISual verifiER), a visual verification tool that automatically filters the DOM-based issues raised by REDECHECK. By adjusting the opacity of the elements involved in a potential layout failure and analyzing the difference in the pixels making up those elements, VISER can quickly detect the human-observable visual changes. Because it automatically filters out the non-observable issues, only presenting the observable ones to a developer, VISER makes the failure verification process fast and repeatable.

We performed an empirical evaluation of VISER on 20 web pages previously used to evaluate REDECHECK. In that


Viewport Size 966 px
Viewport Size 934 px
Viewport Size 933 px
PepFeed
Home
Apps
Blog
Win $100
Support
Buy Tech, with no Regret
Get it Free
Watch the Video
Visit any tech product page
Check other prices and reviews
Shop fast and confidently
Content from trusted sources
Learn how Pep became a Fearless Consumer

(a) No Collision

(b) Non-Observable Collision

(c) Observable Collision

Fig. 1: Three snapshots of a real-world web page that capture a correct layout, in (a), and two distinct collision responsive layout failures, in (b) and (c), as reported by ReDeCheck and correctly classified by Viser without human intervention.

previous study, the responsive layout failures had to be verified manually. That is, humans had to classify 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). In this paper's study, Viser automatically classified all 117 responsive layout failures.

Using Viser also surfaced some of the subjectivity in manually classifying layout issues: 28 failures were categorized differently by Viser, including three that were reclassified as false positives. Furthermore, Viser categorized a significant number of these RLFs differently depending on the point in the viewport range chosen to inspect them. This finding highlights the importance on the viewport point chosen to inspect failures: If humans inspect the layout failure manually, they must inspect multiple viewports to ensure the failure's correct classification, a task that is fully automated by Viser.

The contributions of this paper are therefore as follows:

1) A new technique, implemented in a tool called Viser, that automates a previously manual approach to verifying responsive layout failures reported by ReDeCheck.
2) An empirical evaluation that compares the automated results produced by Viser to those arising from a previously published manual analysis, finding that:
   a) Viser accurately classifies the potential responsive layout failures identified by ReDeCheck, automatically classifying all 117 from prior work.
   b) Viser can automate the manual analysis and eliminate its subjectivity: compared to humans, Viser categorized 28 failures differently, including three that were ultimately reclassified as false positives.
   c) Viser is fast to run, requiring no more than a few seconds to complete all of its automated analyses.

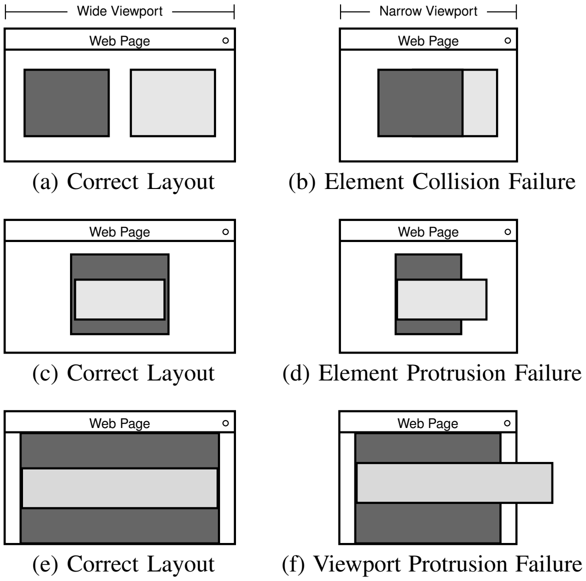
## II. Background

The presentational layer of a web application consists of a series of web pages, which are rendered by a web browser on the basis of several resources. A developer first creates a Hypertext Markup Language (HTML) document, that specifies the basic display structure of a page. An HTML document consists of a series of HTML elements that describe text, images, multimedia, forms, scripts, and other content [12]. Developers associate Cascading Style Sheets (CSS) with an HTML document to specify how a browser should graphically style the HTML elements when rendering the page. Rules in the CSS can style the size and position of elements and can control, for instance, whether the text within them should be rendered in bold face or italic [13]. A browser parses the elements in an HTML document, along with the CSS rules, to form the Document Object Model (DOM) of the web page. The DOM is a tree data structure that represents the page's visual presentation [12]. A developer can query or modify the page's DOM (and consequently, its visual appearance) through the creation and use of scripts run by the browser. An HTML element's properties, such as its width or height, can be assessed by specifying an eXtensible Markup Language (XML) path expression, known as its XPath. The final arrangement of HTML elements on a web page, as rendered by the web browser, is referred to as its layout.

After overviewing the principles of responsive web design, the remainder of this section first explains how testing tools like ReDeCheck automatically detect responsive layout failures. It then surfaces the challenges associated with triaging non-observable issues, thereby setting the stage for Viser.

### A. Responsive Web Design

The responsive web design paradigm [2] incorporates the concepts of fluid grids, flexible media, and media queries, each of which support the web page design strategies for accommodating a range of viewport sizes. Often supported by frameworks such as Bootstrap [14] and Foundation [15], these concepts are implemented using HTML and CSS. Fluid grids allow HTML elements to be arranged in layouts that smoothly adjust according to the width of the viewport, while flexible media refer to images or video content that stretches or shrinks in size depending on available



(a) Correct Layout (b) Element Collision Failure

(c) Correct Layout (d) Element Protrusion Failure

(e) Correct Layout (f) Viewport Protrusion Failure

Fig. 2: Three examples of the types of RLFs reported by REDECHECK and automatically classified by VISER. The left-hand side of this figure furnishes a responsively designed web page with a correct layout. The right-hand side depicts a situation in which a responsive layout failure manifests itself.

screen space. Finally, media queries allow developers to activate specific CSS rules depending on the user's device or browser. For example, any CSS rules contained within the media query `@media(max-width:767px)` would be enabled 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 of a desktop computer.

### *B. Testing to Detect Responsive Layout Failures*

Even with the RWD paradigm, web developers may introduce a wide variety of presentation failures [16], including responsive layout failures (RLFs) [4], for example the one shown in Figure 1. At a viewport width of 966 pixels (part (a)), no problems are apparent. Yet, the space between the third oval and the elements to the right of it becomes constricted at a viewport width of 934 pixels (part (b)). At a viewport width of 933 pixels (part (c)), the third oval element wraps to the next line, partially overlaying the content above it.

One technique, implemented into the REDECHECK tool [4], detects some of the common RLFs in responsively designed web pages. *Element Collision* failures occur in responsive design where the display space is sufficient to accommodate two HTML elements (Figure 2(a)), yet as the viewport becomes narrower, space between the elements tightens until they start to overlap one another (Figure 2(b)). Along with causing unsightly presentational effects, this can lead to a loss of functionality if important links and/or buttons are obscured. *Element Protrusion* failures occur when HTML elements "pop" out of their containers due to reduced display space. At a wide viewport width (Figure 2(c)), the available display space allows for the element to be rendered correctly within its container. However, as display space becomes smaller, the container starts to shrink. The containing element reaches its minimum size, which may be constrained by the text rendered within it. Eventually, the containing element protrudes out of its container (Figure 2(d)). *Viewport Protrusion* is similar to element protrusion, except that an HTML element has protruded out of the viewport itself — that is, it has extended out of the `body` HTML element of the page (Figure 2(e)–(f)).

### *C. The* REDECHECK *Tool for Testing Responsive Web Pages*

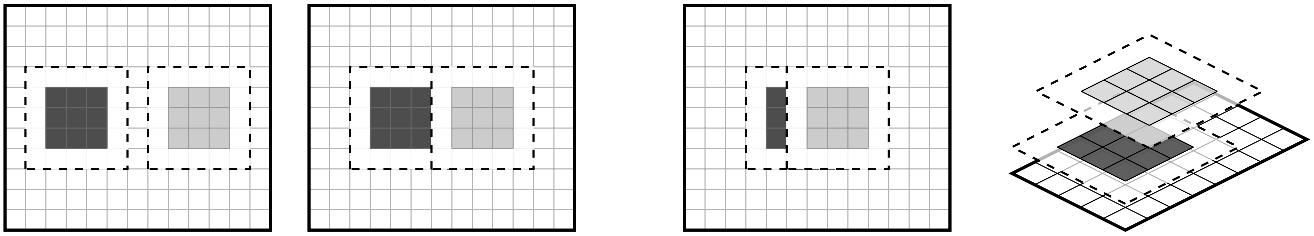
The REDECHECK (REsponsive DEsign CHECKer, pronounced "Ready Check") tool detects these RLFs by extracting a Responsive Layout Graph (RLG) of a web page [11]. An RLG is a model of the responsive layout behavior of a web page [17]. It represents, at different viewport widths, both the relative alignment of HTML elements with respect to one another (e.g., "above", "below", "contained", and "within") and which HTML elements are (and are not) set to be visible at each width. When constructing an RLG, REDECHECK collates information by driving a desktop browser and rendering a web page at different viewport widths in a specified range. This viewport range typically starts with a narrow width, 320 pixels, akin to a mobile phone; extending to a more spacious width of 1400 pixels, a viewport width corresponding to a browser open on a desktop computer. REDECHECK extracts the DOM of the web page rendered at each width and uses it to find the relative alignment of HTML elements.

REDECHECK uses the RLG to find potential layout failures, such as those involving element collisions, by checking for pairs of elements that were not overlapping at a particular viewport width, but then overlap at a narrower width [4]. Intuitively, REDECHECK uses the layout at wider viewports to cross-check narrower widths. If pairs of elements were not overlapping or protruding at a particular viewport width but then do so as the viewport narrows, an RLF is likely to have manifested. This type of checking across viewport widths makes REDECHECK less likely to report false negatives than if a developer was to use, for example, the Fighting Layout Bugs tool [18] that reports anomalies at single viewport widths.

When REDECHECK finds an RLF it produces a report that states (a) the failure type (e.g., element collision or element protrusion); (b) the viewport range of the RLF (i.e., the minimum and maximum viewport width for which the RLF was evident) and finally (c) the XPaths of the HTML elements involved [11]. The next subsection summarizes results from prior studies of REDECHECK, pointing out that, even though the tool improves the testing process, it may highlight certain layout issues that, in practice, developers do not focus on first.

### *D. Non-Observables Issues and the* REDECHECK *Tool*

In a prior empirical study, REDECHECK found RLFs in 16 of 26 web pages studied, and 33 distinct RLFs in total [4]. However, since REDECHECK is based on the DOM, an abstract representation of a web page, one particular problem inherent in its technique is distinguishing issues that are



(a) No Collision (b) Non-Observable Collision (c) Observable Collision (d) Layered View of Collision

Fig. 3: Wireframes of two HTML elements, in light and dark gray, with a white border that is the same color as the background of the page (part (a)). Parts (b) and (c) respectively depict elements with a non-observable collision and an observable collision. Finally, part (d) shows how VISER manipulates opacity to perform automatic visual detection of the responsive layout failure.

observable in practice from those that are not. Figure 3 highlights this problem, depicting two HTML elements in light and dark gray, with a white border that is the same color as the background of the web page, as shown in part (a). Parts (b) and (c) reveal *non-observable* and *observable* collisions, respectively. In part (b), the two elements are technically colliding, but a person testing this web page is unlikely to see this as a problem because only the borders of the elements are overlapping — and they are the same color as the background.

Figure 3(c) shows how an observable issue arises as a result of the dark gray element's content becoming obscured by that of the light gray one. As it does not take into account visual details beyond the size and co-ordinates of the elements concerned, REDECHECK cannot distinguish between the two scenarios in part (b) and (c) and thus reports them both. While the non-observable issues exemplified by part (b) of both Figures 1 and 3 may be of interest to testers — they are latent issues that could manifest in visible RLFs in different contexts — they are unlikely to be a high-priority compared to the actual visual defect in part (c) of these figures. Yet, REDECHECK offers no way to distinguish non-observable issues from true positives, thereby limiting its effectiveness. VISER, introduced in the next section, solves this problem.

## III. AUTOMATIC VISUAL VERIFICATION OF FAILURES

Figure 4 outlines VISER's approach to automatic visual verification of NOIs, comparing it with the series of manual steps that are otherwise required. After a developer runs REDECHECK on a web page, it reports the RLFs that it detects, if any. Each report states the RLF type (e.g., element protrusion), the range of viewport widths for which the RLF was deemed to occur (in the form of a lower and an upper bound), and the XPaths of the HTML elements involved. If VISER is not used, the developer must manually decide what do with these reports. This involves loading up the web page; setting the viewport width of their browser to one within the reported range; manually identifying the elements and scrolling to the failure if necessary; and finally deciding if the RLF is a true positive, false positive, or an NOI. VISER automates these steps. The *web page explorer* component opens the browser, sets the viewport width and locates the faulty elements. It first crosschecks REDECHECK's result by examining the DOM in the *DOM Filter* step, reporting any RLFs believed to be false positives after inspecting the DOM. The final classification is automatically performed by the *image analyzer* component.

The image analysis involves the investigation of specific regions of a web page, which we refer to as *areas of concern* (AOCs). An AOC bounds a rectangle pertaining to the elements involved in a layout failure where its graphical presence is suspected to have inadvertently overwritten other graphics or content on the page, or to have been written to the page out of position. The aim of the image analysis is to determine if this is the case (i.e., the RLF produces visible, observable effects). For example, if the misplaced element has no content and is transparent, the RLF will not be detectable by a human and thus a failure report produced by REDECHECK will be of little concern to developers. After describing how VISER identifies AOCs for different types of RLF, the remainder of this section details the image analysis process and RLF classification.

### A. *Identifying Areas of Concern (AOCs)*

Figure 5 summarizes the ways in which two HTML elements can be arranged spatially with respect to one another. The two elements are depicted by dark gray and light gray boxes, respectively. The figure identifies three particular scenarios: "Contained", where one element resides inside the bounds of another; "Overlapped", where the two elements share some, but not all, of the same display space; and finally "Separated", where the two elements are set completely apart from one another. The figure then shows how AOCs are determined for each type of RLF with respect to each scenario. For *element collision*, the AOC is the portion of the secondary (light gray) element that is contained within the first (A in the fully contained scenario, or B in the overlapped scenario). For *element protrusion*, there are two potential AOCs. The first is the overlapped portion (B), if it exists; and the second the non-overlapping portion (C in the overlapped scenario, D in the separated scenario). The two portions are treated as separate AOCs to simplify the image analysis, which needs to take into account the fact that the foreground element is overlaid on different background elements. The same is true for *viewport protrusion*, except for that, in this case, the dark gray background element corresponds to the body element of the web page — the basic container for all web page elements.

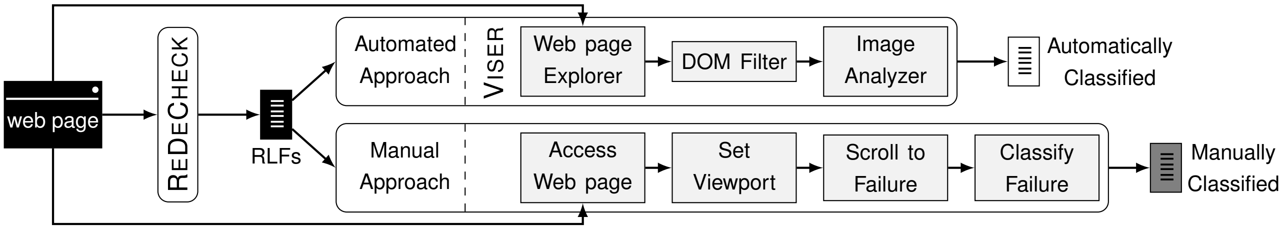


Fig. 4: The high-level architecture of the Viser tool for the automatic visual verification of layout failures. Along with the external input sources, this figure also shows a manual approach to verification and classification that requires a human expert.

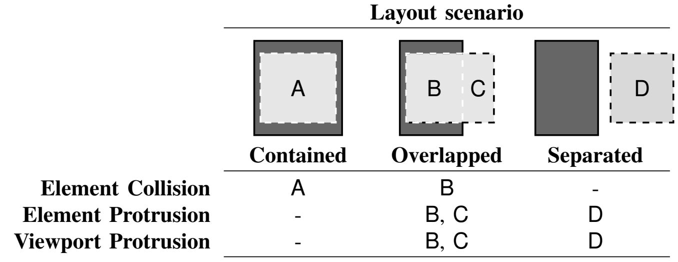
### *B. Verifying Presentation Failures*

Once an AOC has been identified by the method detailed in the last subsection, image analysis tries to determine whether or not the HTML elements involved in the RLF — which are often stacked on top of one another — render different content in the same space or out of position. The approach works to "reveal" the different layers of the AOC by removing the HTML elements concerned from the top level down to the background, systematically removing each element involved in the failure. (As an example, Figure 3(d) shows the stacking of elements involved in an element collision and the different "layers" that are involved.) A snapshot image is taken of the AOC at each layer. The layers are then compared for differences. If there are any, then Viser classifies the RLF as being visible (i.e., it is a true positive). If there are no differences, then the RLF is categorized as non-observable.

Our technique accesses different graphical layers in the display space by manipulating the `opacity` CSS property of HTML elements, thus making it and its descendants invisible. We decided to use `opacity`, as opposed to removing elements completely, because removing elements can impact the layout of the remaining HTML elements on the page [13], thus potentially interfering with the classification. By instead manipulating the element's opacity, the element is still "there" as far as the layout is concerned, but the elements stacked below it are revealed for the purposes of taking a snapshot. This method also has the advantage of being browser-independent.

A technical inconvenience arises when the AOC is larger than the portion of the web page currently viewable, due to the viewport size corresponding to the RLF, a situation that is common with *viewport protrusion* failures. Since the page's responsive design is likely to dictate that the failure no longer occurs, in general the viewport size cannot be increased to bring these elements back into view for snapshotting. In this scenario, Viser horizontally scrolls the page, taking snapshots of individual portions of the page and "sewing" the AOC together as necessary. Regrettably, it is not always possible to scroll and bring protruding elements into view. In these circumstances, Viser performs a "best effort" approximation of the AOC by altering the margins of the offending elements to negative values, thereby trying to move them into view.

Algorithm 1 furnishes the top-level algorithm for Viser. This part of Viser finds the initial AOC to analyze, identified as in Figure 5. The image analysis is then performed by one or both of Algorithms 2 and 3, depending on the layout scenario. If one element contains the other, as with the *contained* scenario of Figure 5, or part of the other, as with the *overlapped* scenario, control passes to Algorithm 2. Depending on the scenario, the AOC is A or B, as shown by Figure 5.



| | Layout scenario | | |
|---|---|---|---|
| | Contained | Overlapped | Separated |
| **Element Collision** | A | B | - |
| **Element Protrusion** | - | B, C | D |
| **Viewport Protrusion** | - | B, C | D |

Fig. 5: Identifying "areas of concern" (AOCs) for different RLFs and layout scenarios involving two distinct HTML elements, as depicted by the light gray and dark gray boxes.

Algorithm 2 takes the two HTML elements involved (i.e., the dark gray and light gray elements of Figure 5) and decreases their opacity to 0%, ensuring they are transparent using MakeTransparent. Three snapshots are then taken, first of the background (where both elements are transparent), which is saved in $imgNoElemets$. Then, restoring $back$ (i.e., the dark gray element) to its original opacity level (using the Restore procedure) a further snapshot, $imgBack$, is taken. Finally, the foreground element is restored, and another snapshot is taken of the AOC and saved to $imgFront$. These three images are then compared in line 9 of the algorithm. If there are differences, and the failure type is an *element collision*, then the RLF is deemed to be visible, and the algorithm returns a true positive (TP), else the verdict is a non-observable issue (NOI). If there are differences for the other two failure types (i.e., *element protrusion* and *viewport protrusion*), then the AOC, when separated from the background element, must be analyzed to see if the content has spilled outside its containing element. In this case, control passes to Algorithm 3, with the AOC now known to be C, as per Figure 5.

In the separated scenario, Algorithm 3 may also be invoked directly from Algorithm 1, with the AOC being identified as D, as per Figure 5. The algorithm proceeds in a similar fashion to that of Algorithm 2, except there are only two layers to consider — that with the foreground element (i.e., the light gray element of Figure 5) present, and that where it is transparent. The two snapshots are compared. If the images are different, then the algorithm returns a true positive (i.e., the

**Algorithm 1** Top-Level Viser Algorithm

**INPUT:** Two HTML elements, $back$ and $front$, and the failure type, $ft$.
**OUTPUT:** TP if the RLF is deemed observable, NOI if it is not.

```
procedure Viser($back, front, ft$)
    $scenario \leftarrow$ getScenario($back, front$)
    if $scenario$ = contained then
        $AOC \leftarrow$ getContainedAOC($back, front$)    ▷ $AOC$ = A (Figure 5)
        return ContainedAOCImageAnalysis($back, front, ft, AOC$)
    if $scenario$ = overlapped then
        $AOC \leftarrow$ getContainedAOC($back, front$)    ▷ $AOC$ = B (Figure 5)
        return ContainedAOCImageAnalysis($back, front, ft, AOC$)
    if $scenario$ = separated then
        $AOC \leftarrow$ getDetachedAOC($back, front$)    ▷ $AOC$ = D (Figure 5)
        return DetachedAOCImageAnalysis($front, AOC$)
```

**Algorithm 2** Image Analysis for Contained AOCs

**INPUT:** Two HTML elements, $back$ and $front$, the failure type, $ft$, the AOC $AOC$.
**OUTPUT:** TP if the RLF is deemed observable, NOI if it is not.

```
procedure ContainedAOCImageAnalysis($back, front, ft, AOC$)
    $back \leftarrow$ makeTransparent($back$)
    $front \leftarrow$ makeTransparent($front$)
    $imgNoElements \leftarrow$ snapshot($AOC$)
    $back \leftarrow$ restore($back$)
    $imgBack \leftarrow$ snapshot($AOC$)
    $front \leftarrow$ restore($front$)
    $imgFront \leftarrow$ snapshot($AOC$)
    if $imgNoElements \neq imgBack \wedge imgNoElements \neq imgFront$ then
        if $ft$ = element_collision then
            return TP
        if $ft$ = element_protrusion $\vee$ $ft$ = viewport_protrusion then
            $AOC \leftarrow$ getDetachedAOC($back, front$) ▷ $AOC$ = C (Figure 5)
            return DetachedAOCImageAnalysis($front, AOC$)
    return NOI
```

**Algorithm 3** Image Analysis for Detached AOCs

**INPUT:** An HTML element, $front$, and the AOC $AOC$.
**OUTPUT:** TP if the RLF is deemed observable, NOI if it is not.

```
procedure DetachedAOCImageAnalysis($front, AOC$)
    $front \leftarrow$ makeTransparent($front$)
    $imgNoElement \leftarrow$ snapshot($AOC$)
    $front \leftarrow$ restore($front$)
    $imgFront \leftarrow$ snapshot($AOC$)
    if $imgNoElement \neq imgFront$ then
        return TP
    return NOI
```

RLF is deemed to be visible), else the non-difference between the layers means that the RLF is non-observable.

As stated in Section II, ReDeCheck reports, for each RLF, a viewport range that is the narrowest to the widest viewport width for which the RLF is manifested at the DOM level. Since Viser has a choice of the viewport at which it can visually inspect the RLF, we made this a configurable parameter of the tool. The default is to look at the narrowest viewport width (i.e., the "low end" of the range) since RLFs are more likely to be noticeable at screen sizes with tighter layout constraints than at wider viewports that are less constrained for space.

## IV. Empirical Evaluation

To investigate the effectiveness and efficiency of Viser, we ran it with the web pages used in the previous evaluation of ReDeCheck [4], for which the RLFs that ReDeCheck identified were classified manually by Walsh et al. Therefore, we adopt the manual classification performed in that study as a baseline to which we compare Viser. The experimental evaluation focuses on answering these three research questions:

**RQ1: Can the technique automatically distinguish non-observable issues from true positives and how does it compare to manual classification?** In this research question, we compare Viser to the results of the manual classification, using Viser's default setting of performing the image analysis at the narrowest viewport width reported for each RLF.

**RQ2: Within the viewport range of a presentation failure, what is the point of inspection that has the best chance of revealing a true positive?** In this research question, we determine if it is best to perform image analysis at the narrowest viewport width for the RLF. We compare Viser to the results of the manual classification process for three points in the RLF's viewport range: the minimum, or narrowest (as per RQ1), the middle of the range, and the maximum width.

**RQ3: How long does the technique take to verify a presentation failure?** In this research question, we investigate how efficient Viser is to run, assessing if it is a practical addition to ReDeCheck for a developer's RWD testing toolset.

### A. Experimental Subjects

We selected the subjects from the pool of web pages used to evaluate ReDeCheck's effectiveness at detecting common RLF types [4]. ReDeCheck is designed to check more types of RLF that cannot result in NOIs. While the original ReDeCheck study involved 26 pages, not all of them involved the types of RLF on which this paper focuses; i.e., element collision, element protrusion, and viewport protrusion. We therefore limited our empirical comparison to the 20 web pages that concerned only these potentially NOI-involving RLF types. Furthermore, we could not re-use the *StumbleUpon* subject. We found that the tool previously used to download all of this subject's resources (see [4] for details) did not work correctly. Since the web page was no longer available in its original form, we could not reconstruct the subject's archive and thus, despite the fact that the original study reported this subject as involving an element collision failure, we had to exclude it. This gave us a total of 117 RLFs, originally reported by ReDeCheck, for use in our evaluation of Viser.

### B. Experimental Methodology

To evaluate Viser, we attempted to match the execution environment, as closely as is possible, to the setup for the original ReDeCheck experiments, thereby avoiding discrepancies in the results that might be due to differences in the experimental setup between the two evaluations. We therefore ran Viser on an iMac with 8GB of RAM, running OS version 10.13 and using Firefox browser version 46. As with ReDeCheck, Viser uses Selenium WebDriver [19] to interact with the web browser to render web pages and visually verify the failures. Viser also rendered the web pages in a browser window without scrollbars and at fixed viewport height of 1000 pixels.

To answer **RQ1**, we ran Viser on each of the 117 RLFs to reach an automatic classification. Viser was configured to use the minimum viewport width reported for the range of each RLF concerned. We then checked whether VISER agreed with the manual categorization of the RLF as decided in the original study by Walsh et al. [4]: true positive (TPs, an observable failure), non-observable issue (NOI), or false positive (FP, no failure). FPs are failures reported by REDECHECK that do not exhibit an issue visually in the design of the web page or in its internal DOM representation. We then calculated the percentage agreement of VISER with the previous manual classification, investigating any differences in the categorization.

To answer **RQ2**, we followed the same methodology as RQ1, but ran VISER using additional inspection points: the middle of the range reported for the RLF by REDECHECK and the maximum point (i.e., the upper bound) of the range. While running this experiment, VISER led us to the discovery of a defect in REDECHECK. For 35 viewport protrusion RLFs, REDECHECK incorrectly reported the upper bound of the viewport range for the failure. Rather than rendering a page at each possible viewport width to construct the RLG, REDECHECK normally samples the entire range by rendering the page at intervals, performing a binary search between the last two sampled points to identify the precise viewport widths at which the relative alignment of two HTML elements, or the visibility of an individual element, changed. Since the RLF is a false positive, the incorrectly reported upper bound had a knock-on effect on the result of VISER. We found that we could get REDECHECK to produce the correct results by changing its interval size to 1. After reconfiguration, we re-ran REDECHECK for the pages involving these particular RLFs.

To answer **RQ3**, we ran VISER for every RLF, examining each one at the lower bound of the range reported by REDECHECK, and recorded the time taken for VISER to run in each instance. We repeated this 30 times for each RLF to obtain a reliable estimate of the running time of VISER and to minimize chance effects that might be caused by, for example, the underlying operating system hosting the experiments.

### *C. Threats to Validity*

The validity of this paper's experiments hinges on accurately matching the previously published manual classification with the classifications automatically produced by VISER. Since the manual results from the experimental evaluation of REDECHECK [4] did not include the XPath of offending elements, the failures were manually matched using the snapshots available. These snapshots, combined with the type of failure, range, and name of the web page enabled us to confidently perform the matching. The second threat to validity is a defective implementation of the VISER tool. To control this threat, we configured VISER to keep a record of all the images used to evaluate each failure. Furthermore, VISER also maintained a record of the coordinates of each offending element. We consulted these records during the examination of all mismatched classifications, thereby helping us to ensure that the prototype operated correctly. To further establish a confidence in the correctness of VISER, we regularly performed additional manual and automated testing. Finally, to support the replication of this paper's experiments, we have made the VISER prototype and its documentation publicly available at `https://github.com/redecheck/viser`.

### *D. Experimental Results*

**RQ1:** Table I furnishes the results from running VISER on the outputs of REDECHECK and their agreement with the manual classification performed by Walsh et al. [4]. For completeness, Table II gives the full, broken down set of manually-classified results from the original Walsh et al. study.

For this research question, we focus on the results from the "Minimum" segment of Table I. The results show that VISER had an 87.2% overall agreement with the manual classification. The table breaks this result down by RLF type. The "Agreement with manual" section shows the ratio of failures for VISER and manual classification, resulting in the reported percentages. The second number is the manual classification total (drawn from the totals in Table II), while the first is the number of those failures that were categorized in the same manner by VISER. At 93.5%, the best level of agreement between manual and VISER is for element collision failures.

Table I shows that there were 15 instances where VISER's classification of an RLF did not agree with the manual outcome. We discuss these 15 instances in three different categories: *subjective*, *obscured*, and *misclassified* RLFs.

Seven RLFs fall into the *subjective* category. While these RLFs have a visual impact, the difference is so small they are almost imperceptible to humans. Two of these cases involved changes to two pixels, yielding no real observable visual difference. While these RLFs are technically TPs, and were classified as such by VISER, the manual analysis subjectively categorized them as NOIs. Future work needs to take these small differences into account when analyzing RLFs.

A further two RLFs were *obscured*, which occurred with the *ConsumerReports* subject. Two RLFs are TPs, and were classified manually as such, yet VISER reported them as NOIs. This was because REDECHECK did not report the most specific elements involved in the failure. While VISER's analysis was correct for the elements it was given by REDECHECK, there was a noticeable visual effect detectable by humans. Since the manual analysis was not limited to the study of only the HTML elements reported by REDECHECK, the effect of the RLF was easily spotted as a TP. This difference is really a bug in REDECHECK, rather than a problem with VISER.

Three viewport protrusion RLFs were *misclassified* by VISER for a variety of reasons. One viewport protrusion failure with *PDF-Escape* was classified by VISER as an NOI but was manually classified as a TP. This is due to the `overflow` property of the protruding element being set as `hidden`. The protruding content could therefore not be "seen" by VISER. Future work needs to modify VISER so that it manipulates the `overflow` property or tracks missing content from one viewport width to another. A further viewport protrusion involved an element that could not be correctly snapshotted by VISER due to the inherent technical limitations involved in reaching off-screen elements by scrolling or manipulating

| | **Minimum** | | | | | | | | | **Middle** | | | | | | | | | **Maximum** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Element Collision** | | | **Element Protrusion** | | | **Viewport Protrusion** | | | **Element Collision** | | | **Element Protrusion** | | | **Viewport Protrusion** | | | **Element Collision** | | | **Element Protrusion** | | | **Viewport Protrusion** | | |
| | TP | NOI | FP | TP | NOI | FP | TP | NOI | FP | TP | NOI | FP | TP | NOI | FP | TP | NOI | FP | TP | NOI | FP | TP | NOI | FP | TP | NOI | FP |
| 3-Minute-Journal | - | 1 | - | - | 2 | - | 8 | - | - | - | 1 | - | - | 2 | - | 6 | 2 | - | - | 1 | - | - | 2 | - | - | 7 | 1 |
| AirBnb | - | 1 | - | - | 1 | 3 | 1 | 3 | - | - | 1 | - | - | 1 | 3 | 2 | 2 | - | - | 1 | - | - | 1 | 3 | 2 | 2 | - |
| BugMeNot | - | - | - | 1 | 3 | - | 1 | 1 | - | - | - | - | 1 | 3 | - | 1 | 1 | - | - | - | - | 1 | 3 | - | 1 | 1 | - |
| CloudConvert | 1 | - | - | - | - | - | - | - | - | 1 | - | - | - | - | - | - | - | - | 1 | - | - | - | - | - | - | - | - |
| Consumer-Reports | 1 | 6 | - | 1 | 3 | - | 9 | 3 | - | - | 7 | - | 1 | 3 | - | 9 | 3 | - | - | 7 | - | 1 | 3 | - | 8 | 4 | - |
| Covered-Calendar | - | - | - | - | - | - | - | 3 | - | - | - | - | - | - | - | - | 3 | - | - | - | - | - | - | - | - | 3 | - |
| Days-Old | - | - | - | - | - | - | - | 1 | - | - | - | - | - | - | - | - | 1 | - | - | - | - | - | - | - | - | 1 | - |
| Dictation | - | - | - | - | - | - | - | 1 | - | - | - | - | - | - | - | - | 1 | - | - | - | - | - | - | - | - | 1 | - |
| Duolingo | 1 | - | - | - | - | - | 2 | 2 | - | 1 | - | - | - | - | - | 2 | 2 | - | 1 | - | - | - | - | - | 2 | 2 | - |
| Honey | - | - | - | - | 8 | - | - | 2 | - | - | - | - | - | 8 | - | - | 2 | - | - | - | - | - | 8 | - | - | 2 | - |
| HotelWifiTest | - | - | - | - | - | - | - | 1 | - | - | - | - | - | - | - | - | 1 | - | - | - | - | - | - | - | - | 1 | - |
| Mailinator | - | 1 | - | - | - | - | - | - | - | - | 1 | - | - | - | - | - | - | - | - | 1 | - | - | - | - | - | - | - |
| MidwayMeetup | 1 | - | - | - | 1 | - | - | 1 | - | 1 | - | - | - | 1 | - | 1 | - | - | 1 | - | - | - | 1 | - | - | 1 | - |
| Pdf-Escape | - | - | - | - | 6 | - | 3 | 1 | - | - | - | - | - | 6 | - | 2 | 2 | - | - | - | - | - | 6 | - | 2 | 2 | - |
| Pepfeed | 4 | 3 | - | - | 2 | - | 1 | 1 | - | 4 | 3 | - | - | 2 | - | - | 2 | - | 4 | 3 | - | - | 2 | - | - | 2 | - |
| Pocket | - | 2 | - | - | 3 | - | - | - | - | - | 2 | - | - | 3 | - | - | - | - | - | 2 | - | - | 3 | - | - | - | - |
| TopDocumentary | - | 7 | - | - | 4 | - | - | - | - | - | 7 | - | - | 4 | - | - | - | - | - | 7 | - | - | 4 | - | - | - | - |
| UserSearch | - | 1 | - | - | - | - | - | - | - | - | 1 | - | - | - | - | - | - | - | - | 1 | - | - | - | - | - | - | - |
| WhatShouldIReadNext | - | - | - | - | - | - | - | 2 | - | - | - | - | - | - | - | - | 2 | - | - | - | - | - | - | - | - | 2 | - |
| WillMyPhoneWork | 1 | - | - | - | 1 | - | - | - | - | 1 | - | - | - | 1 | - | - | - | - | 1 | - | - | - | 1 | - | - | - | - |
| **Total** | 9 | 22 | - | 2 | 34 | 3 | 25 | 22 | - | 8 | 23 | - | 2 | 34 | 3 | 23 | 24 | - | 8 | 23 | - | 2 | 34 | 3 | 15 | 31 | 1 |
| **Agreement with manual** | 7/7 | 22/24 | - | 1/3 | 32/36 | - | 21/24 | 19/23 | - | 7/7 | 23/24 | - | 1/3 | 32/36 | - | 17/24 | 17/23 | - | 7/7 | 23/24 | - | 1/3 | 32/36 | - | 10/24 | 18/23 | - |
| **Agreement per failure type** | 93.5% | | | 84.6% | | | 85.1% | | | 96.8% | | | 84.6% | | | 72.3% | | | 96.8% | | | 84.6% | | | 59.6% | | |
| **Agreement per inspection point** | **87.2%** | | | | | | | | | **82.9%** | | | | | | | | | **77.8%** | | | | | | | | |

TABLE I: The results from using VISER at three inspection points. The columns labeled "Minimum", "Middle", and "Maximum" show the results after VISER inspected the respective points of the reported failure range. In this table "TP", "NOI", and "FP" respectively denote a true positive, non-observable issue, and false positive, as explained in Section IV-B. The "Element Collision", "Element Protrusion", and "Viewport Protrusion" columns correspond to the RLF types of Figure 2.

| | **Manual** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **Element Collision** | | | **Element Protrusion** | | | **Viewport Protrusion** | | |
| | TP | NOI | FP | TP | NOI | FP | TP | NOI | FP |
| 3-Minute-Journal | - | 1 | - | - | 2 | - | 8 | - | - |
| AirBnb | - | 1 | - | - | 4 | - | - | 4 | - |
| BugMeNot | - | - | - | 1 | 3 | - | 2 | - | - |
| CloudConvert | 1 | - | - | - | - | - | - | - | - |
| Consumer-Reports | - | 7 | - | 1 | 3 | - | 9 | 3 | - |
| Covered-Calendar | - | - | - | - | - | - | - | 3 | - |
| Days-Old | - | - | - | - | - | - | - | 1 | - |
| Dictation | - | - | - | - | - | - | - | 1 | - |
| Duolingo | - | 1 | - | - | - | - | 2 | 2 | - |
| Honey | - | - | - | - | 8 | - | - | 2 | - |
| HotelWifiTest | - | - | - | - | - | - | 1 | - | - |
| Mailinator | - | 1 | - | - | - | - | - | - | - |
| MidwayMeetup | 1 | - | - | - | 1 | - | - | 1 | - |
| Pdf-Escape | - | - | - | 1 | 5 | - | 1 | 3 | - |
| Pepfeed | 4 | 3 | - | - | 2 | - | 1 | 1 | - |
| Pocket | - | 2 | - | - | 3 | - | - | - | - |
| TopDocumentary | - | 7 | - | - | 4 | - | - | - | - |
| UserSearch | - | 1 | - | - | - | - | - | - | - |
| WhatShouldIReadNext | - | - | - | - | - | - | - | 2 | - |
| WillMyPhoneWork | 1 | - | - | - | 1 | - | - | - | - |
| **Total** | **7** | **24** | **-** | **3** | **36** | **-** | **24** | **23** | **-** |

TABLE II: The manual classification of RLFs from a prior study [4]. See Table I for a full description of the columns.

the `margin` property. As such, VISER labelled it as an NOI, when it is, in fact, a TP. A final viewport protrusion involved content overflowing out of the viewport that was classified by the VISER algorithm as an NOI, although the manual analysis correctly categorized it as a TP. In this case, a human had to scroll horizontally to read the overflowing content, which did not line up correctly with elements in the page's banner.

The final three element protrusion RLFs were *misclassified* by REDECHECK. VISER found that these were FPs, since there was no protrusion at the DOM level. The manual analysis reported these as NOIs, since there was no visual impact. We judge the root cause of this to be a bug in REDECHECK's collection of DOM information when constructing the RLG.

*Conclusion for RQ1:* VISER demonstrates high agreement (87.2%) with manual classification when set to study the minimum point of the viewport range reported for each RLF.

**RQ2:** Table I shows the results from when VISER was set to inspect the minimum and maximum point of the viewport range for each REDECHECK-reported RLF. The results show that VISER's classification can vary, depending on the chosen inspection point. VISER is more likely to agree with manual inspection at an RLF's minimum viewport width. Compared to an agreement of 87.2% at the minimum width, the agreement for the middle and maximum point of the range drops to 82.9% and 77.8%, respectively. We next investigate the reason for the classification differences at each of these inspection points.

*The "Middle" Inspection Point.* Overall, there are six RLFs for which VISER's classification did not agree with the manual analysis at the middle of the viewport range, for which there was agreement at the minimum. Each RLF was a viewport protrusion; we next discuss these on a case-by-case basis.

For two RLFs, the visibility of the failure varied depending on the viewport width chosen from the range reported by REDECHECK. Thus, the change in classification reported by VISER was correct. These RLFs involve the *PDF-Escape* and *PepFeed* subjects. The manual classification for these two RLFs is a true positive, which is accurate at the minimum viewport that REDECHECK reports. However, as space expands, both RLFs become non-observable in the middle of the range. Thus, the manual analysis judgement does not hold for the entire range reported for each RLF, being correct at the minimum viewport width reported for the RLF, but incorrectly classified at the wider viewport widths. Importantly, VISER can automatically detect the differences in observability.

The other four RLFs involved subjective differences and a misclassification on the part of VISER. For two RLFs, involving the *Airbnb* and *MidwayMeetup* subjects respectively, VISER coincidently agreed with the manual classification at the minimum viewport range reported, classifying the RLFs as NOIs. In both cases, VISER failed to move an element into view at the minimum width. Thereafter, VISER successfully snapshots the offending protrusion by altering their `margin` property, thus reporting the RLFs at TPs. While VISER is technically correct, the overspill is small and can be easily overlooked by a human. Therefore, we categorize these differences as *subjective*. The final two RLFs involve the *3-Minute Journal* subject. Both VISER and the manual analysis agree

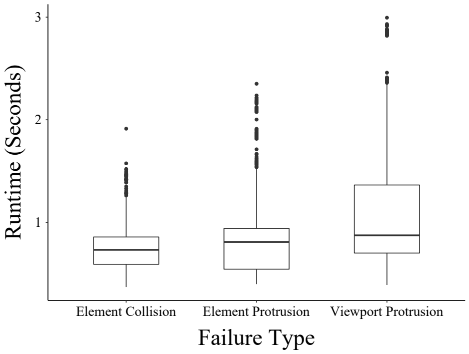


Fig. 6: VISER's execution time across all of the 117 presentation failures and 30 trials and for the three layout failure types. In these plots the bottom and top whiskers show the minimum and maximum data values excluding outliers, while the box itself represents the inter-quartile range, the middle line represents the median value, and the circles are outliers.

that the RLF is a TP at the minimum viewport width. However, VISER categorizes them as NOIs at the middle of the range. In this case, content overflows the viewport, which the tool does not properly detect, leading to a *misclassification* by VISER.

*The "Maximum" Inspection Point.* There were seven RLFs for which VISER's classification matched the manual classification at the minimum and middle inspection point, but not at the reported maximum viewport width. Again, each RLF was a viewport protrusion. In each case, the visibility of the RLF objectively changes, becoming an NOI. Similar to the first two differences identified for the middle inspection point, the manual analysis for these RLFs is a single judgement for the entire range. Hence, it does not take into account the change from visible to non-observable from narrower to wider viewport widths. Six of these RLFs are with the *3-Minute Journal* subject, the last RLF is from *ConsumerReports*.

Another notable result at the maximum point of inspection is an FP classification of a viewport failure for *3-Minute Journal*. An investigation of the issue revealed that the protruding element associated with the RLF had a single pixel difference in width, based on the DOM coordinates as retrieved by the VISER and REDECHECK, respectively. Since it illustrates the challenge of automated testing for responsively designed web pages, we will resolve this difference as part of future work.

*Conclusion for RQ2:* VISER is more likely to agree with manual inspection at the minimum viewport width reported for each RLF. The differences that are evident at wider inspection points can be explained by three phenomena: (1) the fact that a single verdict is produced for an RLF when its visibility may change throughout the viewport range for which it is reported; (2) the visibility/non-visibility of an RLF can be subjective as the viewport width changes; and (3) a small number of misclassified results by VISER, which should form the basis of future work. Notably, the RLFs involved in the classification differences were exclusively viewport protrusion failures.

**RQ3:** To analyze how long it takes to automatically verify a responsive layout failure, the runtime of the VISER prototype was recorded across 30 runs. Figure 6 gives a box plot that visualizes the runtime of the tool when it verifies each type of failure. Across all failure types and failures, the tool took 0.795 (median) / 0.91 (mean) seconds to automatically verify an RLF. Importantly, the time to load the web page and resize the browser were excluded from measurement as this cost would be shared by any technique, whether manual, semi-automated, or automated. All of these recorded times account for the overhead of finding the offending elements, visually verifying the failure, and writing all diagnostic images to disk.

Figure 6 reveals that the time to verify viewport protrusion failures has a "longer tail", resulting in a slightly higher median runtime. This result is due to the extra work that VISER does, for this type of RLF, to move elements into view or to stitch together AOCs that are larger than the viewport.

*Conclusion for RQ3:* On average, VISER took under a second to classify an RLF. Since manual analysis can take several minutes, this means that using VISER is practical and efficient.

### E. Discussion

Determining the observability or non-observability of a presentation failure is not always self evident, making the final decision subjective. Essentially, the task requires an observer to recognize a change between what is visually expected and what is visually apparent. These results show that a manual approach introduces "exemptions" based on the severity of a change. For instance, consider an element *A* that is overlapping the coordinates of an element *B*, with *n* pixels of element *A* overlapping *n* pixels of *B*. In this case, a human would decide whether the *n* pixels of overlap are negligible and if the overall aesthetics remain satisfactory. Both of these criteria are not easily defined and remain, to a great extent, subjective. Nevertheless, we aim to study them as part of future work. For example, it may be useful to measure the number of changed pixels, determine if a color change is visible to the human eye, or introduce heuristics concerning the size of an AOC.

We also note that the previously published manual classification used in this experiment exhibits multiple concerns. Analysis of the data showed that, in some cases, the manual classification was not confined to the type of failure and the XPaths reported. For instance, an element was reported as protruding out of its ancestor element, which was an NOI, but was manually classified as a TP because it was also protruding out of the parent element. As such, a strong argument can be made to reclassify a portion of the manual classification. Although justifiable, we refrained from "tampering" with the benchmark data so as to not introduce any potential bias. Moreover, reclassification would not tackle the underlying subjective nature that is inherent to manual classification.

Since all of the previous research that we investigated in the area of testing web page presentation failures used the manual visual verification approach to evaluate a prototype tool, the accuracy and consistency of the manual approach will influence, positively or negatively, the research outcomes. Although we did not investigate the output of other tools and other types of web page presentation failures, the results make it clear that there are benefits associated with the automated verification and classification of web page presentation failures.

Using the CSS `opacity` property is one way to verify presentation failures without making VISER browser dependent. Another strategy is to manipulate the `visibility` property. However, descendant HTML elements can override the inheritance of this property, meaning that VISER would have to traverse the DOM tree, potentially adding extra implementation complexity and execution time overhead. On the other hand, a limitation of manipulating the `opacity` property emerges when the snapshot is taken before the element has become fully transparent. Strangely, we discovered this to be the case for one particular viewport protrusion RLF where an `input` HTML element was only partially transparent when snapshots of the AOC were taken. This result suggests that, if this instance is not in fact an isolated case, VISER may need a small time delay before a snapshot to overcome this problem.

A final limitation was evident when VISER could not move into view an element involved in a viewport protrusion failure. This result means that, in future work, we must develop strategies to automatically classify RLFs in these scenarios.

## V. Related Work

While, to our knowledge, there has been no research on the automatic visual verification of reported presentation failures in web pages, there is an extensive literature on web testing. For instance, WEBDIFF [20], CROSST [21] CROSSCHECK [22], and X-PERT [23] are all cross-browser testing tools that use the DOM and/or screenshots to detect variations when a page is viewed on different browsers. Similarly, tools such as WEBSEE [24] and FIERYEYE [25] use the prior version of a page as an oracle to detect presentation failures [24]. Unlike this paper, none of the aforementioned tools manipulate HTML element opacity and, critically, they all require a human to verify the detected presentation failures.

The initial version of the REDECHECK tool targeted regression issues by comparing the responsive layout of two versions of a web page [17]. Walsh et al. subsequently presented the version of REDECHECK that we used to automatically detect the RLFs studied in this paper [4]. Like REDECHECK, the VFDETECTOR tool also finds responsive layout failures, even when they are triggered by human interaction [26]. While both of these tools automatically detect different types of responsive layout failures, a developer must still manually inspect problems at multiple viewport widths to determine whether or not they are visible to humans — this is the task that VISER effectively handles in an automated fashion.

There are also several tools that support the verification of the layout properties of a web page. For instance, CASSIUS formalizes some of the semantics of CSS and supports automated reasoning about the behavior of CSS style sheets [27]. The VIZASSERT tool extends the formal model in CASSIUS, further supporting the automated verification of a web page's accessible layout [28]. Finally, the CORNIPICKLE tool verifies that a web page supports the layout properties specified by a tester [29]. Unlike VISER, these tools all require some formal specification of page layout. Moreover, while these tools focus on automatically verifying layout properties, the presented tool instead verifies layout failures reported by a testing tool.

Finally, there are many tools that support the design, implementation, and testing of visual web pages. For instance, SCRY is a reverse engineering tool that surfaces how changes in the underlying source code will influence a page's visual appearance [30]. Moreover, the VISTA tool repairs the broken tests that focus on a web page's visual characteristics [31] and MFIX repairs problems with the responsive nature of a page [32]. Since all of these tools complement VISER's focus on automatically verifying the layout failures reported by REDECHECK, together they form a suite of techniques can improve the quality of responsively designed web pages.

## VI. Conclusions and Future Work

While responsive web design helps to simplify the development of web front-ends for a wide variety of devices with differing screen sizes, developers may still create presentational problems in web pages. Even though the REDECHECK tool automatically checks a web page for responsive layout failures, the manual task of verifying REDECHECK's failure reports is time consuming, imprecise, and error prone. As such, this paper presented a new technique to automatically verify and classify the element collision, element protrusion, and viewport protrusion failures reported by REDECHECK. Implemented into a prototype tool called VISER, this automated technique adjusts the opacity of the HTML elements in an area of concern, looking for visible differences.

Using the results from a previous manual classification as a baseline, this paper's experiments showed that VISER's automatically generated classification agrees with the manual one 87.2% of the time. The results also demonstrate that VISER is most likely to agree with a manual classification when it analyzes a web page at the minimum point of the failure range reported by REDECHECK. Since it takes less than a second to verify a responsive layout failure, this paper's results suggest that VISER is superior to manual classification.

This paper focused on the automatic visual verification of layout failures in responsively designed web pages. Since it adopts Selenium [19], VISER also can be used to visually verify the same failures under different, for instance, runtime environments or browsers, thereby furnishing better support for cross-browser testing with tools like CROSSCHECK [22]. Given its promising integrating with REDECHECK, we also plan to integrate VISER with other tools for responsive web testing, like VFDETECTOR [26]. As part of future work, we also intend to improve VISER to resolve some of the limitations highlighted by this paper's experiments. For instance, since VISER confirms a responsive layout failure even if it only involves a few pixels, we plan to develop heuristics for highlighting those differences most noticeable to humans. Next, we plan to improve our prototype so that it appropriately pauses before taking time-dependent screenshots of an AOC. Finally, we will enhance VISER so that it better handles HTML elements that are "hidden" during the image analysis process.

# REFERENCES

[1] J. McMillen, “10 reasons your website needs to be mobile optimized. http://blog.teamtreehouse.com/10-reasons-website-needs-mobile-optimized.”

[2] E. Marcotte, *Responsive Web Design*. A Book Apart, 2014.

[3] “Creative bloq: Web design trends 2015-16: the long scroll http://www.creativebloq.com/web-design/web-design-trends-2015-16-long-scroll-81516343.”

[4] 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 International Conference on Software Testing and Analysis*, 2017.

[5] D. Robins and J. Holmes, “Aesthetics and credibility in web site design,” *Information Processing & Management*, vol. 44, no. 1, 2008.

[6] D. Cyr, M. Head, and A. Ivanov, “Design aesthetics leading to M-loyalty in mobile commerce,” *Information & Management*, vol. 43, no. 8, 2006.

[7] 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.

[8] “Responsinator,” https://www.responsinator.com/.

[9] “Responsive design checker,” http://responsivedesignchecker.com.

[10] “Viewport resizer,” http://lab.maltewassermann.com/viewport-resizer/.

[11] 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.

[12] World Wide Web Consortium (W3C), “HTML 5.2,” 2017. [Online]. Available: https://www.w3.org/TR/html52/

[13] R. Connolly and R. Hoar, *Fundamentals of Web Development*. Pearson, 2017.

[14] “Bootstrap.” [Online]. Available: https://getbootstrap.com/

[15] “Foundation: Responsive front-end framework,” http://foundation.zurb.com/.

[16] 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.

[17] 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.

[18] “Fighting layout bugs,” https://code.google.com/archive/p/fighting-layout-bugs/.

[19] “Selenium: Web browser automation.” [Online]. Available: http://www.seleniumhq.org/

[20] 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.

[21] A. Mesbah and M. R. Prasad, “Automated cross-browser compatibility testing,” in *Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[22] 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 5th International Conference on Software Testing, Verification and Validation*, 2012.

[23] 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.

[24] S. Mahajan and W. G. J. Halfond, “Detection and localization of HTML presentation failures using computer vision-based techniques,” in *Proceedings of the 8th International Conference on Software Testing, Verification and Validation*, 2015.

[25] 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.

[26] Y. Ryou and S. Ryu, “Automatic detection of visibility faults by layout changes in HTML5 web pages,” in *Proceedings of the 11th Conference on Software Testing, Validation and Verification*, 2018.

[27] P. Panchekha and E. Torlak, “Automated reasoning for web page layout,” in *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016.

[28] P. Panchekha, A. T. Geller, M. D. Ernst, Z. Tatlock, and S. Kamil, “Verifying that web pages have accessible layout,” in *Proceedings of the 39th Conference on Programming Language Design and Implementation*, 2018.

[29] 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*, vol. 85, 2016.

[30] B. Burg, A. J. Ko, and M. D. Ernst, “Explaining visual changes in web interfaces,” in *Proceedings of the 28th Annual Symposium on User Interface Software and Technology*, 2015.

[31] A. Stocco, R. Yandrapally, and A. Mesbah, “Visual web test repair,” in *Proceedings of the 26th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, 2018.

[32] S. Mahajan, N. Abolhassani, P. McMinn, and W. G. J. Halfond, “Automated repair of mobile friendly problems in web pages,” in *Proceedings of the 40th International Conference on Software Engineering*, 2018.