R2Z2: Detecting Rendering Regressions in Web Browsers through Differential Fuzz Testing

Suhwan Song
Seoul National University
Seoul, South Korea
sshkeb96@snu.ac.kr

Jaewon Hur
Seoul National University
Seoul, South Korea
hurjaewon@snu.ac.kr

Sunwoo Kim
Seoul National University
Seoul, South Korea
sunwoo28.kim@snu.ac.kr

Philip Rogers
Google
Mountain View, CA, United States
pdr@google.com

Byoungyoung Lee*
Seoul National University
Seoul, South Korea
byoungyoung@snu.ac.kr

ABSTRACT

A rendering regression is a bug introduced by a web browser where a web page no longer functions as users expect. Such rendering bugs critically harm the usability of web browsers as well as web applications. The unique aspect of rendering bugs is that they affect the presented visual appearance of web pages, but those web pages have no pre-defined correct appearance. Therefore, it is challenging to automatically detect errors in their appearance. In practice, web browser vendors rely on non-trivial and time-prohibitive manual analysis to detect and handle rendering regressions.

This paper proposes R2Z2, an automated tool to find rendering regressions. R2Z2 uses the differential fuzz testing approach, which repeatedly compares the rendering results of two different versions of a browser while providing the same HTML as input. If the rendering results are different, R2Z2 further performs cross browser compatibility testing to check if the rendering difference is indeed a rendering regression. After identifying a rendering regression, R2Z2 will perform an in-depth analysis to aid in fixing the regression. Specifically, R2Z2 performs a delta-debugging-like analysis to pinpoint the exact browser source code commit causing the regression, as well as inspecting the rendering pipeline stages to pinpoint which pipeline stage is responsible. We implemented a prototype of R2Z2 particularly targeting the Chrome browser. So far, R2Z2 found 11 previously undiscovered rendering regressions in Chrome, all of which were confirmed by the Chrome developers. Importantly, in each case, R2Z2 correctly reported the culprit commit. Moreover, R2Z2 correctly pin-pointed the culprit rendering pipeline stage in all but one case.

*Corresponding author

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

rendering regression, web-browser, differential testing

ACM Reference Format:
Suhwan Song, Jaewon Hur, Sunwoo Kim, Philip Rogers, and Byoungyoung Lee. 2022. R2Z2: Detecting Rendering Regressions in Web Browsers through Differential Fuzz Testing. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510044

1 INTRODUCTION

A rendering regression is a bug introduced by a web browser where a web page's visual appearance no longer matches what users expect. Rendering regressions impact the usability of web browsers, the revenue of websites, as well as long-term trust in web browsers. As an example, a major cloud computing platform, ServiceNow, suffered a service outage due to a rendering regression where Chrome 89 failed to paint parts of the web page [20]. According to the Chrome team's bug tracker, 18,400 bugs were filed against rendering-related components (i.e., DOM, Style, Layout, and Paint) between 2016 and 2021 [4], highlighting the non-trivial and prohibitive development costs to maintain correct rendering in Chrome.

The unique aspect of rendering bugs[1] is that they affect the presented visual look of a web page which has no clear, formally-defined notion of a bug decision boundary. In particular, we found two key challenges related to rendering bugs. First, it is challenging to identify rendering bugs because it is difficult to determine the correctness of rendering results. The correctness of browser rendering is mostly dictated by the complex HTML and CSS specifications, which are difficult to completely express into programmable, software-friendly conditions for automated verification. Furthermore, specifications are incomplete and do not cover all aspects of rendering (e.g., table width distribution is only partially specified [16]). Second, even after identifying a rendering bug, it is challenging to analyze and fix it. Unlike memory corruption bugs, rendering bugs do not raise an immediate violation, so they do

[1]In this paper, the word "bug" implies "regression bug" if not specifically mentioned.

not leave a clue about which code is responsible for a bug. Worse yet, for performance, rendering is processed through a multi-stage pipeline, further complicating the bug analysis. We note that these unique aspects and challenges are not present in common traditional bugs such as memory corruption bugs, which can be modeled using a clear violation condition after defining a valid memory region [46, 53].

In this paper, we propose R2Z2, a differential fuzz testing technique to find rendering regressions in web browsers. In order to address the aforementioned two challenges, R2Z2's approaches can be summarized into the following two features. First, R2Z2 features a bug oracle along with rendering change detection. R2Z2 runs two versions of a browser with a randomly generated HTML file, and attempts to identify rendering image differences. In order to avoid false positive bugs, R2Z2 developed a regression oracle, which is capable of determining if a given bug is indeed a rendering regression bug or not. This regression oracle exploits two interesting characteristics in web browsers—1) if multiple independently developed browsers generate the same rendering results from the same HTML input, it is likely that both produce the correct rendering; and 2) browser developers add a testcase when there is a rendering feature update, which can be used to validate the rendering correctness with respect to the new feature.

Second, R2Z2 features two automated analyses, the bisect analysis and the rendering pipeline analysis, which pinpoint the culprit commit and pipeline stage responsible for the bug. As developers are currently doing this manually, we believe R2Z2's automated analyses can significantly reduce engineering costs.

More specifically, R2Z2 designs four components, i) change detector, ii) bisect analysis, iii) regression oracle, and iv) rendering pipeline analysis. The change detector finds any HTML file where rendering results are different across two different browser versions. Then the bisect analysis finds the culprit browser commit which first introduces the rendering difference. Once finding the culprit browser commit, the regression oracle determines if the rendering difference is truly due to a regression bug. Lastly, the rendering pipeline analysis performs in-depth differential testing to pin-point which pipeline stage is responsible for the regression.

We implemented R2Z2 and evaluated it with the popular web browser, Chrome. In the course of our evaluation, R2Z2 identified 11 new rendering bugs, all of which are confirmed by the Chrome developers. For those rendering bugs, R2Z2 correctly pinpointed the culprit commit. R2Z2 also correctly pinpointed the culprit pipeline stage in all cases except one, demonstrating its practical ability to assist in the bug fix process as well.

The key point that R2Z2 exploits is that differential testing is made possible by the existence of multiple independently developed implementations, and we can use that to bootstrap quality in all web browser implementations. Hence, the general ideas and lessons that we developed and learned from R2Z2 can further be utilized for various suites of programs meeting this criterion in the future. These programs include any set of multiple independent implementations targeting the same standards, such as PDF viewers, SVG engines, Java virtual machines, or SSL/TLS servers or clients.

To summarize, this paper makes the following contributions:

- **Design.** We designed R2Z2, a differential fuzz testing tool to automatically detect browser rendering regression bugs. After detecting rendering image differences, it features a bug oracle to filter out false positive cases. R2Z2 also performs automated analyses to spot a specific code commit and pipeline stage responsible for the bug, which can significantly reduce the entailed engineering costs.
- **Promising Results.** While performing the evaluation, R2Z2 found 11 new rendering regressions in Chrome. All of these were confirmed by the Chrome developers and six have already been fixed. For all bugs, R2Z2 correctly spotted the culprit commit. For all bugs except one, R2Z2 correctly spotted the culprit rendering pipeline stage. These results suggest the strong practical aspects of R2Z2 to detect and triage browser rendering bugs.

2 BACKGROUND

2.1 Fuzz Testing and Differential Testing

Fuzzing. Fuzzing is a popular bug finding technique. It repeatedly generates random test cases to run against a target program and monitors for erroneous behaviors such as crashing, hanging, or memory access violations. From an engineering point of view, fuzz testing does not require expert domain knowledge of a target program, so it has been widely used in various software applications.

Most fuzzers are designed to find bugs where it is easy to express the buggy conditions (so called non-semantic bugs). This is related to the fact that a fuzzer alone does not have the capability to detect a bug—it has to observe a certain buggy condition during the fuzzing. As such, most fuzzing techniques have been proposed to find memory corruption bugs [2, 14, 27, 29, 30, 40, 54], which are operated with the memory error detectors which clearly exhibit buggy conditions (e.g., ASAN [53] and UBSAN [23]).

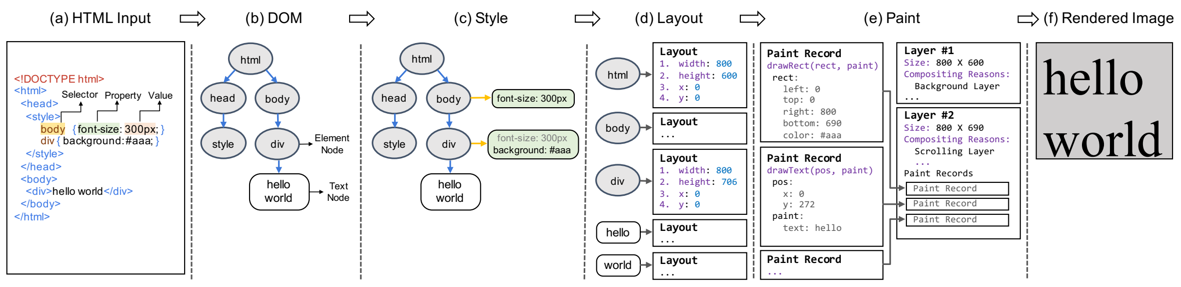
Differential Testing. Fuzz testing alone is not effective in finding semantic bugs [45] because semantic bugs are difficult to express as a bug condition and thus require domain knowledge to determine. In this regard, differential testing methods can be an effective technique to find semantic bugs. Differential testing runs multiple programs, all of which are supposed to produce the same output for the same input. Differential testing then compares the outputs, and if the output is different, it determines that the program likely has a semantic bug. For instance, previous works leveraged differential testing to find semantic bugs in Java virtual machines (JVM), SSL/TLS implementations, web browsers, graphic driver libraries, and CPU RTLs [28, 31–34, 36, 39, 43, 51].

In fact, fuzz testing and differential testing can be employed together if the input of the differential testing is provided through the fuzzing procedure, which we refer to as *differential fuzz testing* throughout this paper.

2.2 The Rendering Pipeline of a Web Browser

Rendering is the process of turning resources (e.g., HTML and CSS) into pixels. Modern web browsers, including Chrome, Firefox, and Safari, use roughly the same high-level steps [35, 38] which form the rendering pipeline: 1) DOM, 2) Style, 3) Layout, 4) Paint.

DOM. The Document Object Model (DOM) is an in-memory tree of nodes with the most common node types being `element` and `text`. The DOM is initially constructed from an HTML file using a well-defined parsing algorithm [17] and can later be modified using



```
<!DOCTYPE html>
<html>
  <head>
    <style>
      body {font-size: 300px; }
      div { background:#aaa; }
    </style>
  </head>
  <body>
    <div>hello world</div>
  </body>
</html>
```

Figure 1: The high-level rendering pipeline of modern web browsers.

JavaScript APIs. For example, in the DOM tree construction shown in Figure 1 (b), the element nodes are `html`, `head`, `style`, `body`, and `div`, and there is a text node with the string `hello world`.

Style. Visual effects are applied to DOM nodes by the style step. Cascading Style Sheets (CSS) defines style properties such as `font-size`, `background`, and `position`. CSS also defines "selectors" for mapping these properties to DOM nodes. The style step determines the CSS properties and values, together called the "computed style", that apply to each DOM node. For example, as described in Figure 1 (c), the `div` element has the CSS property `background` with value `#aaa`.

Layout. Layout takes the DOM nodes with computed styles and computes *boxes* with size and location. A box can be as simple as an individual node in the DOM tree. Multiple boxes can be created for a DOM node, such as one box for each line of text. CSS specifications define when to create boxes, as well as how to calculate their size and location. For example, as shown in Figure 1 (d), the `hello world` text node produces one box for each line. The size of the text results in the box for the parent `div` expanding to `706px` tall.

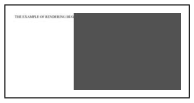
Paint. Paint iterates the boxes from the layout step in back-to-front order to produce low-level graphics instructions (i.e., paint record). CSS specifications define the order [8]. For example, in the case of the `div`, the rectangular background with color `#aaa` would be painted before the text `hello world`. Then, pixels are finally produced by rasterizing the paint records. Additionally, some compositor-only CSS properties such as scrolling effects and 3D transforms are applied at this step. "Threaded compositing", or just "compositing", is the use of two techniques to improve the efficiency of rasterization: *threading* and *compositing*. Threading is the optimization of rasterizing on one or more additional threads. Compositing is the optimization of caching portions of rasterized output that change together. The portions of rasterized output from paint records that change together are placed in the same layer. For scrolling content, the rasterized pixels for an entire scrolling area are cached in a texture, which avoids needing to rasterize on every scroll.

2.3 Rendering Bugs

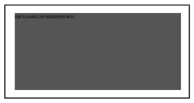
A rendering bug is a bug where the browser fails to render a given page according to HTML and CSS specifications (if specified), or how the user expects. Rendering bugs can manifest in many different ways to users. For instance, if the layout stage has a bug, the browser may incorrectly place an HTML element [18], harming the

```
<html>
  <style>
  .class7 {
    backdrop-filter: hue-rotate(1deg);
    filter: brightness(0.32269068);
    padding: 66%;
  }
  </style>
  <body>
  THE EXAMPLE OF RENDERING BUG
    <span class="class7" ></span>
  </body>
</html>
```

(a) PoC HTML code

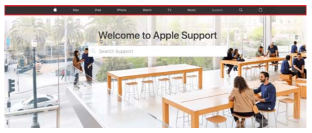


(b) Correct (Chrome 85)

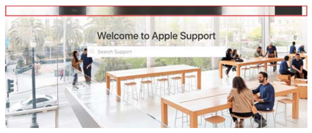


(c) Incorrect (Chrome 86)

Figure 2: A rendering bug example (Chrome Issue #1122021).



(a) Correct (Chrome 79.0.3944)



(b) Incorrect (Chrome 79.0.3945)

Figure 3: The rendering bug (Chrome Issue #1037830), which was triggered on Apple's homepage, https://support.apple.com. The menu bar is disappeared on the right (highlighted with the red box).

user experience by disrupting the web-page layout. Taking another example, if the paint stage has a bug, visual effects on some content may be incorrect [10].

We explain the example of a rendering bug through the HTML code, as shown in Figure 2. In this example, CSS style `filter` is applied to <span> element. Thus, the correct rendering is to paint the <span> element with gray, which is rendered by Chrome 85. However, Chrome 86 renders this incorrectly, painting the outside of <span> element with gray as well. The root cause of this bug is that Chrome 86 incorrectly includes the empty rectangles (i.e., width and height are zero) if their offsets are outside the frame, so the area specified by <span> is over-extended.

If such a web page is used for commercial services, rendering bugs can critically damage the reliability and fidelity of the page. For instance, Figure 3 shows a case where a rendering bug made the menu bar unusable on Apple's homepage.

3 CHALLENGES AND OUR APPROACH

This section first identifies challenges in identifying rendering regressions in web browsers (§3.1). Then we shortly describe our approach to address these challenges (§3.2).

3.1 Challenges of Rendering Bugs

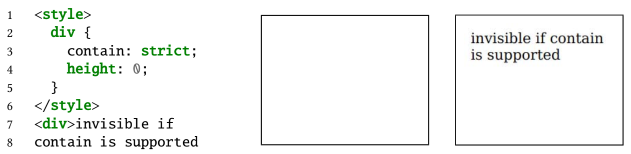
In this subsection, we elaborate on two challenges in identifying and analyzing rendering bugs.

Challenge#1: Identifying Rendering Bugs. In order to detect the rendering bugs, one should be able to clearly determine the correctness of the rendered result. However, it is challenging to design such a rendering bug oracle. This is primarily because rendering bugs are semantic bugs—the rendering bug involves semantically incorrect rendering results which violate HTML and CSS specifications where it is difficult to confirm semantic correctness/incorrectness through automated or programmatic techniques. In other words, given an HTML and its rendered result, it is challenging to develop systematic mechanisms or tools which determine if the rendered result indeed follows the HTML and CSS specifications [9, 13]. For this reason, in practice, rendering bugs are confirmed through manual inspection by domain experts (i.e., browser developers).

There have been two general research directions to handle this issue: i) formalized methods [44, 47, 48] and ii) differential testing. Formalized methods attempt to statically verify if the browser implementation of the browser follows HTML and CSS specifications. Using various static analysis techniques (such as formal verification techniques or symbolic execution), this method can completely inspect the correctness of an entire browser's rendering implementation. However, converting the HTML and CSS specifications to formalized rules is labor-intensive and error-prone, demanding expert domain knowledge of specifications as well as browser implementation.

Differential testing compares the result of two different browsers to detect rendering bugs. It determines there is a rendering bug when two browsers produce different results (e.g., two rendered images) from the same HTML input. Unlike formalized methods, this approach can be easily adapted to detect the rendering bugs as it does not require domain knowledge and human efforts such as writing formalized rules. In practice, however, the rendered image generated from the same HTML can be quite different across different browsers due to benign browser incompatibilities.

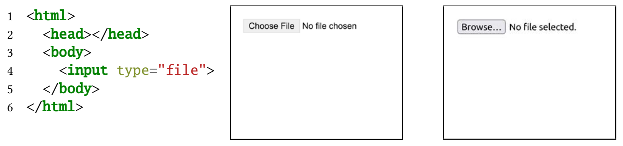
We observe two main factors behind benign browser incompatibilities. First, web browsers can have different development status in supporting features, so a certain standard feature may or may not be supported by each browser. As a result, rendering results for such a partially supported feature are different across web browsers. For instance, CSS `contain: strict` is supported by Chrome but not Safari (illustrated in Figure 4). Thus, Chrome and Safari render the HTML differently. Second, web browsers generate their own unique rendering for features that are not specified by specifications (i.e., under-specified). For instance, the designs of `<input type=file>` in Chrome and Firefox are different, introducing rendering results differences (illustrated in Figure 5). Therefore, differential testing alone can generate many false positives due to these benign browser incompatibilities.



```
<style>
  div {
    contain: strict;
    height: 0;
  }
</style>
<div>invisible if
contain is supported
</div>
```

(a) Example code (b) Chrome (c) Safari

Figure 4: Example of supported feature differences.



```
<html>
  <head></head>
  <body>
    <input type="file">
  </body>
</html>
```

(a) Example code (b) Chrome (c) Firefox

Figure 5: Example of benign design differences.

In short, both previous approaches suffer from high false positives. In practice, browser developers invest considerable manual effort to identify rendering bugs. If any rendering issue is reported, manual analysis is needed to confirm if a bug is present, implying that the complete automation of detecting rendering bugs is challenging.

Challenge#2: Analyzing Rendering Bugs. We find that it is difficult to analyze and fix rendering bugs due to the following two main factors: 1) complex rendering pipeline; and 2) semantic bug. In order to fix a rendering bug, it is essential to find out which stage of the rendering pipeline and which part of code in the stage has a bug.

However, as described in §2.2, the rendering pipeline of the browser is long, and each stage of the pipeline has high complexity as well as a dependency on the result of its previous stage (factor 1). In addition, rendering bugs are semantic bugs so they do not generate a clear violation such as crash (factor 2). In the case of memory corruption bugs, they display the violation so that the developers can obtain clear violation contexts from the violation for debugging. However, unlike the memory corruption bugs, it is challenging to get clear violation contexts without any violation signal from a rendering bug, so only rendered results may suggest the violation context. In order to pin-point the true violation context, one may need to manually trace back the complex implementation of the rendering pipeline. In this respect, previous work only focused on how to find the bug, not how to aid in fixing the bug. In practice, browser developers rely on manual analysis to fix rendering bugs.

3.2 Our Approach

To address the challenges that we described in §3.1, we propose two approaches.

Approach#1: Bug Oracle along with Change Detection. We add the bug (pseudo) oracle capability (§4.3) along with the change detection capability (§4.1). Previous work (on fuzzing or differential testing) focuses on detecting changes. However, change detection alone may generate false positives, as we mentioned in §3. In order to solve this challenge, we propose the regression bug oracle component, which can reduce the false positive issues.

The key idea of the regression oracle is to use cross-browser interoperability to detect correctness. Cross-browser interoperability is where different browsers (e.g., Chrome and Firefox) generate the same rendered result from the same HTML file, because they follow the same HTML and CSS specifications. Therefore, it is worth noting that when two different browsers render the same HTML the same, we can say both browsers properly render the HTML, and the rendered result can be the reference result of the HTML file. In this respect, we can use this reference result as an oracle to determine whether a new browser version is correctly implemented. We will explain the details of regression oracle in §4.3.

Approach#2: Automated Rendering Bug Analysis. We followed the manual bug analysis process of browser teams and automated it. In the bug analysis process, we observed that the browser team manually performed 1) version bisect, and 2) pipeline pin-point analysis. The version bisect is to find out which commit introduces a bug. The pipeline pin-point analysis is to find out which stage of the rendering pipeline triggers a bug.

R2Z2 automates the version bisect through the bisect analysis (§4.2) and the pipeline pin-point analysis through the bug rendering pipeline analysis (§4.4). First, the key idea of bisect analysis is to utilize the delta debugging technique along with change detection. The intuition behind delta debugging is that the rendered image should be significantly changed before and after the culprit commit. In order to automatically pin-point the culprit commit, the bisect analysis performs a binary search using the change detector.

Second, the key idea of the rendering pipeline analysis is to utilize the cross-version differential testing to compare the results of each pipeline stage in order. The intuitions here are 1) there should be a culprit stage which introduces the rendering bug in the rendering pipeline; and 2) the culprit stage is the stage that generates the different results in two versions for the first time, because each stage of rendering pipeline generates its output based on the output of its previous stage. In this respect, the rendering pipeline analysis sequentially compares the results of each stage from two adjacent versions of the browser, which is obtained from the bisect analysis. Then, if the results of a certain stage are different, the analysis regards that stage as the culprit stage and terminates.

4 DESIGN

Now we describe the design of R2Z2. The overall workflow of R2Z2 is shown in Figure 6, which has four components operating in order: 1) change detector (§4.1), 2) bisect analysis (§4.2), 3) regression oracle (§4.3), and 4) rendering pipeline analysis (§4.4).

The change detector in R2Z2 first generates an HTML file (which we refer to as `html`). Then it opens `html` using two different versions of the same browser, say `A` and `B`, and captures two rendered images (① in Figure 6). Next, it checks whether two rendered results are different. If different, R2Z2 minimizes `html` and determines that `html` has a potential to be a rendering bug (which we call a candidate rendering bug, $\texttt{html}^{\textbf{CandBug}}$) (②). The next is the bisect analysis, which finds the culprit version (i.e., culprit commit), raising the first

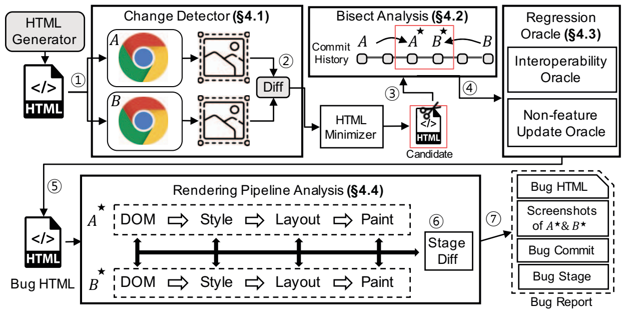


Figure 6: The overall workflow of R2Z2.

rendering difference between `A` and `B` (③). We refer to the culprit commit as $\texttt{B}^\star$ and the previous commit as $\texttt{A}^\star$.

This bisect analysis is essential for the next two components: the regression oracle and the rendering pipeline analysis. The regression oracle identifies $\texttt{html}^{\textbf{OracleBug}}$, which filters out benign issues from $\texttt{html}^{\textbf{CandBug}}$ (④). R2Z2 performs rendering pipeline analysis on the $\texttt{html}^{\textbf{OracleBug}}$ to identify which stage of rendering pipeline is responsible for the regression bug (⑤). It compares each result of four stages on two browsers $\texttt{A}^\star$ and $\texttt{B}^\star$, all of which are an internal representation during the browser rendering pipeline procedure—DOM tree, Style tree, Layout tree, and Paint output (i.e., Layers) (⑥). After this testing, R2Z2 can tell which state (as well as value) introduced $\texttt{html}^{\textbf{OracleBug}}$. We refer to the first difference-introducing stage as a bug stage. Finally, R2Z2 generates a final bug report on $\texttt{html}^{\textbf{OracleBug}}$ (⑦), which includes the following information: a minimized HTML file, the screenshots of $\texttt{A}^\star$ and $\texttt{B}^\star$, the culprit commit, and the pipeline stage and details of the bug.

4.1 Change Detector

The change detector leverages cross-version differential testing to detect rendering changes between two major browser versions (e.g., Chrome v91 and Chrome v92) on a given `html`. As described in §3.1, a rendering difference is required for a rendering bug, but a difference alone is not sufficient to determine a bug is present. One example of a benign difference is the implementation of a new feature which is expected to have a rendering change. Therefore, the change detector identifies a candidate bug (i.e., $\texttt{html}^{\textbf{CandBug}}$) which is used by later stages to determine true bugs (i.e., $\texttt{html}^{\textbf{OracleBug}}$).

We chose `pHash` to detect the rendering difference. `pHash` is a well known perceptual hashing algorithm, producing representative fingerprint of a given image using a graphical algorithm [3, 52, 56]. `pHash` computes the hash value based on the low frequency components of image. After completing the hash computation, each pixel in a low frequency filtered image is converted to 1 (or 0) if its value is higher (or lower) than the median value of all the pixels. We note that it is possible to use other image comparison algorithms instead of using `pHash`.

After obtaining individual `pHash` values from two rendered images, we measure the distance between two images by computing the hamming distance between two hash values, which we refer to as $|\texttt{P}_\texttt{A} - \texttt{P}_\texttt{B}|$. Here, $\texttt{P}_\texttt{A}$ and $\texttt{P}_\texttt{B}$ denote `pHash` values of `html` on the browser version `A` and `B`, respectively. It is worth noting that a `pHash` value is not a scalar value but a vectorized value, so we compute $|P_A - P_B|$ using the hamming distance for each element in the vector. The minimum value of $|P_A - P_B|$ is zero which indicates that two rendered images are visually identical. The maximum value of $|P_A - P_B|$ is the number of `pHash` bits (e.g., if the number of bits is 64, the maximum hamming distance is also 64) which indicates that two rendered images are significantly different. R2Z2's decision boundary on a candidate rendering bug is a configurable threshold value, `Thresh` —i.e., if $|P_A - P_B|$ is larger than `Thresh`, it determines the given input `html` is $\texttt{html}^{\textbf{CandBug}}$. In our preliminary experiments, we tuned the threshold value of `pHash` to have the following two properties: (i) the threshold value should avoid false negatives as much as possible and (ii) the threshold value may allow false positives to some extent. This is because R2Z2's regression oracle is able to cater false positive issues (as we will describe at §4.3), so we designed the change detection phase to focus on not missing potential bugs if possible. Using initial testcases, we obtained the empirically-tuned threshold value 140 while considering the aforementioned two properties.

4.2 Bisect Analysis

The bisect analysis locates the culprit commit, $B^\star$, responsible for the rendering difference of $\texttt{html}^{\textbf{CandBug}}$. This is important because the change detector uses major release versions of a web browser, where many code commits are taken place in between. For example, there are approximately 14,500 commits between Chrome v91 and Chrome v92. Thus, the bisect analysis pinpoints the culprit commit from a wide range of commits, allowing R2Z2 to avoid any unintentional side-effects from unrelated commits in the follow-up analyses.

In order to efficiently identify the culprit commit, a binary search is performed over the linear sequence of commits. It is possible that multiple commits are responsible for the difference, but the current design of R2Z2 assumes that the latest commit from those is the culprit commit. This is following the common practice exercised by browser development teams for engineering efficiency, i.e., the later commit likely shadows the previous commit. The culprit commit will be then used by the bug regression oracle (§4.3), to filter out false positives, and the rendering pipeline analysis (§4.4), to avoid unrelated differences.

Workflow of Bisect Analysis. The workflow of bisect analysis is as follows: 1) Given two versions of browsers, `A` and `B`, R2Z2 computes `pHash` values of `html`, P_A and P_B; 2) R2Z2 picks a version between `A` and `B`, say `M`, and then computes P_M; 3) R2Z2 computes $|P_M - P_B|$. If $|P_M - P_B|$ is larger than `Thresh`, R2Z2 will search the half between `M` and `B`. If $|P_M - P_B|$ is zero (i.e., the P_M is the same as P_B), it will search the other half, between `A` and `M`. 4) R2Z2 recursively searches through the region—i.e., it returns to step 2 and keeps continuing until the two version of browsers `A` and `B` are adjacent. 5) If `A` and `B` are adjacent, the pre-culprit commit, $A^\star$, is set to `A`. The culprit commit, $B^\star$, is set to `B`.

4.3 Regression Oracle

R2Z2 develops a pseudo bug oracle, called *regression oracle*, to handle the challenges in identifying rendering bugs (§3.1). Specifically, given $\texttt{html}^{\textbf{CandBug}}$ (§4.1) as well as the bisected browser version information (§4.2), the regression oracle aims to filter out false positive issues in detecting rendering bugs. The regression oracle operates with two chained sub-oracles, the interoperability oracle (§4.3.1) and the non-feature update oracle (§4.3.2).

	$B^\star \neq R$	$B^\star = R$
$A^\star \neq R$	Not a Bug (Case 1)	Not a Bug (Case 2)
$A^\star = R$	Bug (Case 3)	Infeasible ($A^\star \neq B^\star$)

(a) Interoperability oracle.

$A^\star$	$B^\star$	R	Decision
Fail	Fail	-	Not a Bug (Infeasible, Case 1)
Fail	Pass	Fail	Not a Bug (Case 2)
Fail	Pass	Pass	Bug (Case 3)
Pass	-	-	Not a Bug (Infeasible, Case 4)

(b) Non-feature-update oracle.

Figure 7: Regression oracle decision table.

4.3.1 *Interoperability Oracle.*

The interoperability oracle is based on the following assumption:

Assumption: Interoperable rendering is correct. If two independently-implemented browsers (e.g., Chrome and Firefox) generate the same rendered result from the same input, it is likely that both produce the correct rendering.

This assumption is based on the observation that browsers are independently implemented to follow the same HTML/CSS standards. Let us suppose two different browsers render a given `html` the same, then both browsers are either correct or incorrect. Of these two possible outcomes, it is unlikely that both browsers are incorrect—for this to be the case, both would need to have independently implemented the same bug. Furthermore, if two browsers do contain the same bug, it is possible that many web developers are already aware of the bug and thus develope their websites depending on the bug, making the buggy behavior a *de facto*, empirical standard. Therefore, we assume that the interoperable behavior is likely implicating that both browsers produce a correct rendering behavior. Running additional browsers (e.g., Chrome, Firefox, Internet Explorer, Opera) can further strengthen this assumption.

It is worth noting that cross-version testing in the change detector (§4.1) does not leverage this interoperability because two versions of the same browser are not individually implemented and can have the same bug.

Constructing Interoperability Oracle. R2Z2 constructs the interoperability oracle based on rendering interoperability. Given $\texttt{html}^{\textbf{CandBug}}$ (i.e., an HTML testcase which generates different rendering results), the interoperability oracle determines if it is a true bug or a false positive. R2Z2 runs a reference browser, denoted as `R`, which has an independent implementation from the base browser used in the previous phases (change detection (§4.1) and bisect analysis (§4.2)). The current prototype of R2Z2 runs Chrome as the base browser and Firefox as the reference browser.

Running the reference browser and two versions of the base browser with the same `html`$^{\textbf{CandBug}}$, the interoperability oracle observes three possible cases as shown in Figure 7a. First, Case 1 represents the case where both browsers have different rendered results with `R`. Since the rendering interoperability cannot be leveraged for different rendered results, the oracle cannot infer the correctness of any browsers. Thus, the oracle determines that Case 1 is not a bug. Case 2 represents the case where `A`★ has the different rendered result. In this case, as the rendered results of `B`★ and `R` are the same, we can learn that both `B`★ and `R` are correct according to the rendering interoperability. The oracle determines Case 2 is not a bug, because `A`★ is an older version of `B`★ and thus the latest `B`★ fixed the bug presented in `A`★. Case 3 represents the case where only `B`★ has a different rendered result. In this case, as the rendered results of `A`★ and `R` are the same, we can infer that both `A`★ and `R` are correct because they are interoperable. Hence the oracle determines that Case 3 is a bug—`B`★ is a newer version of `A`★ so case 3 implies that `A`★ (and `R`) correctly rendered the HTML file, and a bug is introduced in `B`★.

In summary, the interoperability oracle determines `html`$^{\textbf{CandBug}}$ is a bug if it falls into Case 3 in Figure 7a, otherwise it is not a bug.

4.3.2 Non-feature-update Oracle. To avoid potential rendering regressions and ensure interoperable behavior, web browser developers are strongly encouraged to add web-platform-tests (WPT) [26] for every code commit introducing a new rendering feature. Here, a feature can refer to something in a new specification, or even a part of an existing specification. WPT tests are written in a format that can be run in all browsers and there is continuous integration testing this. R2Z2 uses the behavior of newly-added WPT tests to filter out commits where the base browser is implementing a new feature.

Constructing Non-feature-update Oracle. R2Z2 constructs the non-feature-update oracle which is capable of determining if a certain code commit (i.e., `B`★) introduces a new rendering feature that is not supported in the reference browser `R`. If the commit introduces a new feature, R2Z2 determines that it is not a bug. This is because, although `html`$^{\textbf{CandBug}}$ triggers rendering changes which break interoperability, the rendering difference can be due to the new feature, which should not be considered as a bug. If the commit does not introduce a new feature, R2Z2 determines it is a bug.

More specifically, the non-feature-update oracle is constructed as follows. First, the non-feature update oracle checks whether `B`★ has corresponding WPT tests. If not, it determines that `html`$^{\textbf{CandBug}}$ is a bug. Next, R2Z2 runs each corresponding WPT test on `A`★, `B`★, and `R`, where the decision table is summarized in Figure 7b. In this table, it is assumed that `A`★ always fails the WPT test while `B`★ should always pass because that is the original function of the WPT test. Although it is practically infeasible, it is still possible that this pre-condition does not hold if the WPT test itself has a correctness issue (i.e., the WPT testing code has a bug). As R2Z2 cannot learn any from these cases (i.e., Case 1 and Case 4 in Figure 7b), the non-feature update oracle determines this as a not-a-bug case so as to conservatively avoid false positives. For the case that `A`★ fails, `B`★ passes, and `R` fails (i.e., Case 2), the non-feature update oracle determines it is not a bug. This is because `B`★ is the first browser that introduces the new feature, so the rendering changes of `html`$^{\textbf{CandBug}}$ can possibly be due to this new feature update.

However, for the case that `A`★ fails, `B`★ passes, and `R` passes, the non-feature update oracle determines it is a bug (i.e., Case 3). This case implies that both `B`★ and `R` correctly implemented the new feature associated with the WPT test, but `html`$^{\textbf{CandBug}}$ is not relevant to this new feature—`B`★ and `R` produced different rendering results as already tested by the interoperability oracle.

To summarize, the non-feature-update oracle determines the given `html`$^{\textbf{CandBug}}$ (which passed the interoperability oracle) is a bug if `A`★ fails, `B`★ passes, and `R` passes the corresponding WPT test as shown in Case 3. Since this bug is passed by both interoperability oracle and the non-feature update oracle, we term this as `html`$^{\textbf{OracleBug}}$ which passes the regression oracle.

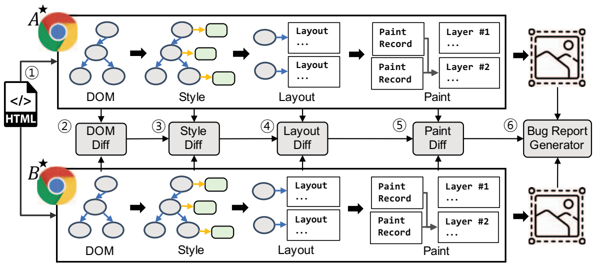
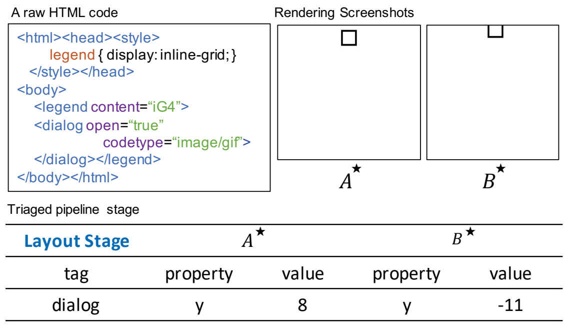


Figure 8: Overview of the rendering pipeline analysis.



```
<html><head><style>
      legend { display: inline-grid; }
  </style></head>
<body>
    <legend content="iG4">
    <dialog open="true"
                codetype="image/gif">
    </dialog></legend>
</body></html>
```

Layout Stage	A★		B★	
tag	property	value	property	value
dialog	y	8	y	-11

Figure 9: Final bug report example.

4.4 Rendering Pipeline Analysis

Once finding `html`$^{\textbf{OracleBug}}$ using the regression oracle, a rendering pipeline analysis attempts to figure out which rendering stage triggers the bug. To be specific, R2Z2 first opens an `html`$^{\textbf{OracleBug}}$ file in two versions of the same browser, `A`★ and `B`★. Then throughout the rendering pipeline stages of web browsers (§2.2), R2Z2 performs differential testing using the internal representation of each pipeline stage. More specifically, R2Z2 performs the following four differential testing on each rendering stage (Figure 8): 1) DOM test, 2) Style test, 3) Layout test, 4) Paint test. Note that this test runs four pipeline stages in order. If the test finds a difference, it stops and reports the current stage. This is because later stages will be different once a difference is present.

DOM Test. In DOM test, R2Z2 checks if two versions of the same browser generate the same DOM tree. To this end, R2Z2 performs a DOM tree equality check—both should have the same tree structure and each node should have the same tag and attributes. If any of these is different, R2Z2 determines there is a DOM bug and generates a human-readable report showing the DOM tree difference. More specifically, R2Z2 first opens $\texttt{html}^{\textbf{OracleBug}}$ on $\texttt{A}^{\star}$ and $\texttt{B}^{\star}$ (①). Then, it obtains a DOM tree from each and traverses the DOM tree in a depth-first order to compare the tag name (e.g., `body` and `div`) and attributes (e.g., `type`) of each node (②).

Style Test. After DOM test, R2Z2 compares the style information generated from each of the browser (i.e., $\texttt{A}^{\star}$ and $\texttt{B}^{\star}$) through style tree equality check—nodes in both trees should have the same property names and values. In particular, it obtains style information from the browser after a style stage in the rendering pipeline, and traverses again the DOM tree in a depth-first order while comparing the style information (e.g., `font-size: 30px`) (③). R2Z2 determines there is a Style bug if any of tree nodes have different style information, as in DOM test.

Layout Test. In Layout test, R2Z2 checks if two versions of the same browser generate the same Layout boxes. To perform the layout box equality check, each matching box (i.e., a layout box pointed by the same node in both DOM trees) should have the same size and location. Since we already checked the equality of the DOM trees, we traverse both DOM trees in the same order and compare the geometric properties of the layout boxes pointed by the same DOM node (④). R2Z2 determines that any difference in this stage is a Layout bug.

Paint Test. Finally in Paint test, R2Z2 checks if two browsers generate the same layers. R2Z2 performs the equality check—both browsers generate the same number of layers with the same size, compositing reason, and paint records. To be specific, R2Z2 obtains the layers using Chrome DevTools [11], and sequentially compares each layer's size, compositing reason, and paint records (⑤). If any of these are different, R2Z2 determines it is a Paint bug.

Final Bug Report. At the end of R2Z2's analysis, R2Z2 generates a final rendering bug report (⑥). We provide the example of the final bug report in Figure 9. The report includes 1) a raw HTML code triggering the bug, 2) rendering screenshots of $\texttt{A}^{\star}$ and $\texttt{B}^{\star}$, 3) the bug commit (i.e., $\texttt{B}^{\star}$), and 4) the triaged bug stage as well as the specific elements having different pipeline results.

5 IMPLEMENTATION

We implemented R2Z2 targeting the Chrome browser. We used `Domato` fuzzer [14], Imagehash [19], Selenium [21], and Chrome DevTools [11]. `Domato` fuzzer is a grammar-based DOM fuzzer that uses the context-free grammars to generate HTML files. We modified the `Domato` fuzzer to generate HTML files without the animations. We implemented the change detector and bisect analysis of R2Z2 by using 1) Selenium to capture the rendered results from two different versions of browser, and 2) Imagehash to compute a `phash` value of images and determine whether two images are different. We set the number of hash bits as 4,096 to compute a `phash` value and `Thresh` as 140 to determine if the two images are different. In order to implement the regression oracle, we used Firefox as a reference browser. We implemented the rendering pipeline analysis by using DOM APIs and Selenium to compare the results of DOM, style, and layout between two browsers. We used Chrome DevTools to compare the layer information such as paint records and compositing reason between two browsers. In terms of the implementation complexity, R2Z2 is implemented with 2,000 lines of Python and 100 lines of JavaScript.

Browser setting		**Test env. 1**	**Test env. 2**
Version (Release Date)	`A`	Chrome 84.0.4138.0 (May 07, 2020)	Chrome 91.0.4472.0 (Apr 09, 2021)
	`B`	Chrome 86.0.4188.0 (Jul 01, 2020)	Chrome 94.0.4585.0 (Jul 24, 2021)
	`R`	Firefox 82.0 (Aug 25, 2020)	Firefox 93.0a1 (Aug 09, 2021)
# of commits b/w `A` & `B`		18,091	34,052

Figure 10: Experimental configurations.

6 EVALUATION

This section evaluates various aspects of R2Z2, particularly focusing on answering the following research questions:

- **RQ 1.** How many candidate bugs can the change detector find? (§6.1)
- **RQ 2.** Can the bisect analysis (§4.2) accurately identify culprit commit of candidates? (§6.2)
- **RQ 3.** Can the regression oracle (§4.3) accurately identify true regression bugs from the candidates? (§6.3)
- **RQ 4.** Can the rendering pipeline (§4.4) analysis correctly determine the bug introducing stage? (§6.4)

Experimental Setup. We tested R2Z2 with two sets of different browser versions, as shown in Figure 10. R2Z2 considers stable (released) versions as `A` and the latest development version as `B`. This is because an important aspect in this paper is specifically finding regression bugs. Even though two versions of a browser may have the same bug, only the later version could have a "regression". In this respect, it is important to fix regression bugs that are currently impacting end-users, so we picked the stable version as `A`. Then we used the latest development version as the `B`, which (i) maximizes the commit window to be searched for and (ii) offers a certain level of stability. Next, we used the reference browser `R` with the browser released at the nearest time as `B`, because browser developers try to keep the compatibility between different ones released around the same time [6]. It is possible that two browsers (i.e., Chrome and Firefox) may support different features, but we note that this does not cause false positive issues for R2Z2. This is because the interoperability oracle identifies the bug only if $\texttt{A}^{\star}$ and `R` were the same (Case 3), which implies both browsers support all the features to run the testcase.

For the first testing environment, we selected Chrome 84.0.4138.0 (i.e., commit position 766,000) as `A`, Chrome 86.0.4188.0 (i.e. commit position 784,091) as `B`, which has 18,091 commits in between. We used Firefox 82.0 as the reference browser `R`. For the second, we selected Chrome 91.0.4472.0 (i.e., commit position 870,763) as `A`, Chrome 94.0.4585.0 (i.e., commit position 904,815) as `B`, which has 34,052 commits in between. We used Firefox 93.0a1 as the reference browser `R`. We ran all experiments on a 24-core server running Ubuntu 18.04 with Intel Xeon(R) Gold 5118 (2.30GHz) processors

Statistics	Test env 1	Test env 2
# of tested `html`	200K	200K
# of $\texttt{html}^{\textbf{CandBug}}$	6,785	16,205
Elapsed time (h)	2	2.5
Throughput (HTML/s)	27.78	22.22

(a) Change detector.

Statistics	Test env 1	Test env 2
# of tested $\texttt{html}^{\textbf{CandBug}}$	6,785	16,205
# of bisected $\texttt{html}^{\textbf{CandBug}}$	6,643	15,986
Elapsed time (h)	4	10
Throughput (HTML/s)	2.17	2.25

(b) Bisect analysis.

Figure 11: Run-time statistics of change detector and bisect analysis in R2Z2.

and 512GB memory. Then, we leveraged the Domato fuzzer to generate HTML inputs which are fed into R2Z2 for change detection and analysis.

6.1 Effectiveness of Change Detection

We counted the number of candidate bugs (i.e., $\texttt{html}^{\textbf{CandBug}}$) that the change detector identified as shown in Figure 11a. After testing 200K randomly generated HTML files, the change detector in each test environment found 6,785 and 16,205 $\texttt{html}^{\textbf{CandBug}}$, respectively. The number of $\texttt{html}^{\textbf{CandBug}}$ in the second environment is doubled from the first, which is due to the large commit gap of the second test environment.

On average, about 5% of the `html` files caused rendering differences between `A` and `B`. While 5% can be interpreted as a small number, it accounts for 6,785 or 16,205 $\texttt{html}^{\textbf{CandBug}}$. These all may be worth manually analyzing, but it would impose prohibitive engineering costs. This result supports the challenge in identifying the rendering bugs that we described in §3.1, which also signifies the importance of handling false positive issues with R2Z2's regression oracle.

6.2 Effectiveness of Bisect Analysis

In order to evaluate the effectiveness of bisect analysis, we first checked whether the bisect analysis properly finds the culprit commit of each candidate and then measured the elapsed time of bisect analysis. In this experiment, we assumed that all Chrome commit versions between `A` and `B` were pre-built for the bisect analysis, which are, in practice, already available from the automated build testing infrastructure. Thus, we did not include such overheads as it is already part of the browser development chain.

The result of bisect analysis is shown in Figure 11b. In the first testing environment, 6,643 out of 6,785 $\texttt{html}^{\textbf{CandBug}}$ (i.e., 97.9%) were successfully bisected to culprit commits. It failed to pin-point the culprit commit of 142 candidates. The result for the second testing environment was similar, with 219 $\texttt{html}^{\textbf{CandBug}}$ failing. These failure cases are mostly due to Chrome failing to render, such as the browser crashing or hanging. For instance, while Chrome was able to render $\texttt{html}^{\textbf{CandBug}}$ at both `A` and `B`, it failed to render at a certain commit in between. For the failed commit, R2Z2 cannot capture the rendering result, so R2Z2 terminates the bisect analysis. While failing to render limits the effectiveness of R2Z2's bisect analysis, this is rare and we believe addressing this issue is an orthogonal research problem [5]. In terms of the analysis time, the elapsed time was about 4 and 10 hours, for each testing environment, respectively. Translating these results into the throughput, R2Z2 processed about 2.21 $\texttt{html}^{\textbf{CandBug}}$ files per second, which we believe is reasonably fast enough to be used in the production development chain.

With respect to the correctness of the bisect analysis, R2Z2 was able to correctly pin-point the culprit commit. Specifically, as shown in Figure 12, all culprit commits of 13 $\texttt{html}^{\textbf{TrueBug}}$ (which R2Z2 found in §6.3) were confirmed to be correct by Chrome developers (i.e., the accuracy of the bisect analysis is 100%). Considering the fact that the bisect analysis is manually performed by Chrome developers, we argue that R2Z2's automation can significantly improve the efficiency of the bug triage and debug processes.

6.3 Effectiveness of Regression Oracle

This section evaluates the effectiveness of the regression oracle. We first use an interoperability oracle and then use non-feature-update oracle to find $\texttt{html}^{\textbf{OracleBug}}$. Then, for deduplication, we classified the $\texttt{html}^{\textbf{OracleBug}}$ by their culprit commit and picked one $\texttt{html}^{\textbf{OracleBug}}$ from each culprit commit. Specifically, R2Z2 detected 27/247 $\texttt{html}^{\textbf{OracleBug}}$ from 6,643/15,986 candidates by using the interoperability oracle in the first/second test environments, respectively. Then 27/247 $\texttt{html}^{\textbf{OracleBug}}$ were bisected into 10/11 unique culprit commits, respectively. From 21 unique culprit commits, we picked an $\texttt{html}^{\textbf{OracleBug}}$ from each unique culprit commit. The number of true/false bugs is 14/7 (i.e., the true positive rate is 66.7%). Among 21 commits, the culprit commits of eight true and five false bugs have web-platform-tests, respectively. Non-feature-update oracle filtered out four false positives and one true bug. It failed to filter out two false positives as they do not have web-platform-tests. To sum up, given 22,629 $\texttt{html}^{\textbf{CandBug}}$, the regression oracle identified 16 $\texttt{html}^{\textbf{OracleBug}}$. After reporting these, 13 bugs were confirmed as true regression bugs, and three $\texttt{html}^{\textbf{OracleBug}}$ were false positives of R2Z2 (i.e., the true positive rate is 81.25%, and the false positive rate is 18.75%).

New Regression Bugs. Through the evaluation, R2Z2 identified 13 true regression bugs. Out of these, 11 regression bugs were newly identified by R2Z2 and two were previously (and independently) identified by other users and researchers. The list of new rendering regression bugs is shown in Figure 12. After reporting, six of these regression bugs were fixed by developers and thus R2Z2 prevented such regression bugs from harming users, demonstrating R2Z2's impact on spotting regression bugs.

False Positives. Two false positive cases were due to the fact that R2Z2 was not able to use the non-feature update oracle—i.e., browser developers did not include WPT tests for these two, so the non-feature update oracle simply determined they were $\texttt{html}^{\textbf{OracleBug}}$. While these false positive issues may seem to be the limitation of R2Z2, the testcases were useful to browser developers. Specifically, after reporting these bugs, browser developers mentioned that $\texttt{html}^{\textbf{OracleBug}}$ in fact tests the new features that were updated by B* [24]. Therefore, `html`$^{\text{OracleBug}}$ can be used to add missing testcases.

6.4 Correctness of Rendering Pipeline Analysis

In order to evaluate the correctness of the rendering pipeline analysis, we first obtained the ground truth, a true culprit stage for all `html`$^{\text{TrueBug}}$. As we reported `html`$^{\text{TrueBug}}$, the browser developers performed the manual analysis and left the annotation on which pipeline stage is responsible for the bug. We were able to obtain the ground-truth for 12 `html`$^{\text{TrueBug}}$ (out of total 13 `html`$^{\text{TrueBug}}$ that R2Z2 detected). For the one remaining unlabeled one, browser developers failed to pinpoint a specific stage although they confirmed the bug.

According to our evaluation, the rendering pipeline analysis correctly spotted all of 12 `html`$^{\text{TrueBug}}$ (i.e., the accuracy is 100%), showing the effectiveness of this analysis. Because no `html`$^{\text{TrueBug}}$ were in DOM and Style, we prepared an extra evaluation with existing DOM and style bugs which are reported by other Chrome contributors. Searching Chrome bug tracker [1], we selected three DOM bugs, which are reproduced in older versions than our evaluation environment, because there was no recent DOM bug. We also selected four Style bugs, which are valid in the version range of our evaluation environment. The result is that R2Z2 correctly pin-pointed all DOM bugs. However, in the case of Style bugs, it correctly pin-pointed three out of four Style bugs. As shown in this evaluation, R2Z2 was able to correctly spot a culprit pipeline stage for all tested bugs except one Style bug, demonstrating its strong potential to significantly save manual engineering work performed by browser developers. Finally, we conducted a case study on the incorrectly pin-pointed Style bug to clarify the flaw of pipeline analysis as described below.

Case Study: A Failure Case of Pipeline Analysis. In Style test, the pipeline analysis failed to find the culprit stage of the bug (Chrome issue #1154537 [22]). This is because R2Z2 relies on `getComputedStyle` DOM API to retrieve the style information, but for some cases it does not return the complete internal information used in the style stage. For these cases, the complete internal information can only be retrieved through inspecting the raw memory while handling virtual address differences between two browser instances. While this clearly is a limitation of R2Z2, we leave this as our future work as it only occurred in one case out of 19.

7 DISCUSSION

This section discusses future research directions of R2Z2, particularly stating how R2Z2 can further be utilized for other use-cases.

Detecting Regressions in Other Web Browsers. R2Z2 currently supports Chrome but does not support other web browsers (e.g., Safari, Firefox, and Edge). Since the design of R2Z2 is generic, these can be supported by R2Z2 with moderate implementation effort. In particular, R2Z2's run script needs to be modified to spawn these other browser instances. The bisect analysis requires including a browser-specific build script for each version. The rendering pipeline analysis can be implemented in a similar way by using each browser's development APIs to collect intermediate rendering information.

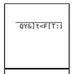
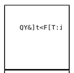
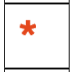
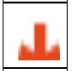
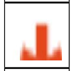
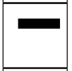
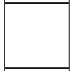
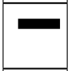
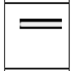
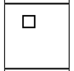
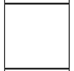
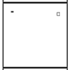
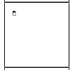
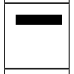
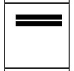
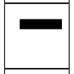
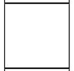
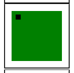
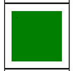
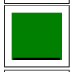
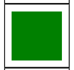
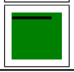
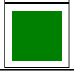
Issue ID	Culprit Commit	Culprit Stage	Correct	Incorrect	Confirmed	Fixed
#1121082	775116 (✓)	Paint (✓)			✓	✓
#1164652	779663 (✓)	Layout (✓)			✓	
#1226558	780992 (✓)	Layout (✓)			✓	✓
#1231397	770064 (✓)	Paint (✓)			✓	
#1237352	885372 (✓)	Paint (✓)			✓	✓
#1240854	885961 (✓)	Paint (✓)			✓	
#1240856	890916 (✓)	Layout (✓)			✓	✓
#1241345	889344 (✓)	Undecided			✓	
#1241356	888805 (✓)	Layout (✓)			✓	✓
#1241436	885635 (✓)	Paint (✓)			✓	✓
#1242851	887727 (✓)	Layout (✓)			✓	
#1245637*	784040 (✓)	Paint (✓)			✓	✓
#1245639*	766419 (✓)	Paint (✓)			✓	✓

Figure 12: The list of 13 `html`$^{\text{TrueBug}}$ found by R2Z2 in Chrome. 11 bugs were newly identified by R2Z2, and two were independently identified by other contributors (annotated with *).

Using Regression Oracle for Other Programs. While this paper leverages the regression oracle for rendering regression bugs, it can be adopted to find regressions in other programs in the future. There are many programs in which multiple independent programs implement the same standards (e.g., PDF viewers, SVG engines, Java virtual machines, or SSL/TLS servers or clients). Since these programs have the same challenge in identifying regression issues as browser rendering (i.e., complex and unclear standard specifications), R2Z2's regression oracle can be utilized to correctly identify regressions while avoiding false positives.

8 RELATED WORK

Browser Layout Testing. Previous work has proposed tools to help web-page developers discover cross-browser incompatibilities in web applications [33, 34, 43]. They automatically identified cross-browser incompatibilities in web applications by detecting rendering differences of two different browsers. Compared to this previous work, the focus of R2Z2 is in identifying rendering bugs in web browsers rather than web applications. Moreover, these do not employ the bug oracle, so they suffer from false positive issues. There is another work that found HTML presentation failures in web pages [41, 42], which requires manually-developed oracles by domain experts. Compared to these, R2Z2's oracle does not involve the manual processes.

Browser Layout Verification. There are several works that partly formalized the browser layout algorithm [44, 47, 48]. [48] proposes visual logic to express accessibility guideline and leverages finitization reductions to properly formalize the fragment of the browser layout algorithm. Troika [49] proposes modular layout proofs for verification of web page layout. As described in §3.1, converting HTML and CSS specifications to formalized rules is labor-intensive and error-prone, requiring expert domain knowledge of specifications as well as browser implementation. Compared to these, R2Z2 is able to perform the automated rendering bug detection without domain expert's knowledge.

Fuzzing to find Semantic Bugs. NEZHA [51] exploits the behavioral asymmetries between multiple test programs to find semantic bugs. DeepXplore [50] and DLFuzz [37] guide Deep Learning (DL) systems to expose incorrect behaviors using neuron coverage. Some studies employ differential testing to discover semantic bugs in Java Virtual Machine (JVM) implementations [28, 31, 32]. GLFuzz [36] leverages metamorphic testing to find shader compiler bugs. It inserts dead code into the graphics shading languages, such as OpenGL, and checks whether the original code and variant code are semantically equivalent based on their rendered results. GLFuzz determines that the variant code is a bug case when the rendered results are significantly different.

DOM Fuzzing. Many DOM fuzzers [7, 12, 14, 15, 25, 55] are proposed to find the memory-related bugs from the web browsers. For instance, cross fuzz [7] dynamically generates the sequences of DOM APIs to bind multiple HTML documents for stress-testing the garbage collection mechanisms of web browsers. In this way, it identified about one hundred memory-related bugs from web browsers (e.g., Internet Explorer, Firefox, and Opera). DOMFuzz [15] uses DOM API calls to test some parts of browser engines (e.g. layout). Wadi [25], Domato [14] and Dharma [12] are the generation-based DOM fuzzers, which generate HTML inputs based on their HTML, CSS, and JavaScript grammars. FreeDOM [55] leverages a context-aware intermediate representation to generate semantically-valid HTML documents.

9 CONCLUSION

This paper proposed R2Z2, a differential fuzz testing technique to find rendering regressions in web browsers. R2Z2 features two unique techniques to find and analyze rendering bugs. First, it features a regression oracle along with the rendering change detection so as to detect regression bugs while avoiding the false positives. Second, it features the bisect analysis and the rendering pipeline analysis, allowing R2Z2 to spot the culprit commit and pipeline stage, which are responsible for the bug. With the prototype implementation for the Chrome browser, it identified 11 new rendering bugs in Chrome, all of which were confirmed by Chrome developers.

10 DATA AVAILABILITY

We disclosed our data used in this paper at https://doi.org/10.6084/m9.figshare.16569561.v1.

ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for their insightful comments. This work was partially supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.2020-0-01840, Analysis on technique of accessing and acquiring user data in smartphone) and National Research Foundation (NRF) of Korea grant funded by the Korean government MSIT (NRF-2019R1C1C1006095). The Institute of Engineering Research (IOER) and Automation and Systems Research Institute (ASRI) at Seoul National University provided research facilities for this work.

REFERENCES

[1] Chromium bug tracker. https://bugs.chromium.org/p/chromium/issues/list.

[2] american fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[3] Average hashing (ahash). http://www.hackerfactor.com/blog/index.php?/archives/529-Kind-of-Like-That.html.

[4] Rendering-related chromium bugs reported between 2016 and 2021. https://bugs.chromium.org/p/chromium/issues/list?q=Component%3AInternals%3ECompositing%2CBlink%3EFonts%2CBlink%3EFullscreen%2CBlink%3ELayout%2CBlink%3ETextAutosize%2CBlink%3ECSS%2CBlink%3EEditing%2CBlink%3EPaint%2CBlink%3ECompositing%2CBlink%3ESVG%2CBlink%3EImage%2CBlink%3EHitTesting%2CBlink%3EGeometry%2CBlink%3ECanvas%2CBlink%3EDOM%2CBlink%3EHTML%2CBlink%3EForms%20Type%3DBug-Regression%2CBug%20opened%3E2016-1-1%20opened%3C2021-1-1%20-label%3APerformance-sheriff%20-label%3AClusterFuzz&can=1.

[5] Clusterfuzz. https://google.github.io/clusterfuzz/.

[6] Chrome git commit 4f8ac4264908b7717de500800688c13c028c0831. https://chromium.googlesource.com/chromium/src/+/4f8ac4264908b7717de500800688c13c028c0831.

[7] Canonical randomized crawl version optimal for most browsers. https://lcamtuf.coredump.cx/cross_fuzz/.

[8] Appendix e. elaborate description of stacking contexts, . https://www.w3.org/TR/CSS2/zindex.html.

[9] Css standard, . https://www.w3.org/TR/?tag=css.

[10] Chrome issue 1037830:page https://support.apple.com/ rendering error, . https://bugs.chromium.org/p/chromium/issues/detail?id=1037830.

[11] Chrome devtools. https://developer.chrome.com/docs/devtools/.

[12] Dharma. https://github.com/MozillaSecurity/dharma.

[13] Dom standard, . https://dom.spec.whatwg.org/.

[14] Domato, . https://github.com/googleprojectzero/domato.

[15] Domfuzz, . https://github.com/MozillaSecurity/domfuzz/tree/master/dom.

[16] A concrete example with tables which are underspecified. https://crbug.com/1214206.

[17] Html specification, . https://html.spec.whatwg.org/.

[18] Chrome issue 1003810: https://web.whatsapp.com/ rendering problems, . https://bugs.chromium.org/p/chromium/issues/detail?id=1003810.

[19] Imagehash python libaray. https://github.com/JohannesBuchner/imagehash.

[20] Chrome issue 1184357: Chrome 89 doesn't paint new fallback nodes added to slot. https://bugs.chromium.org/p/chromium/issues/detail?id=1184357.

[21] Selenium webdriver. https://www.selenium.dev/.

[22] Issue 1154537: text-decoration-thickness doesn't work without text-underline-offset. https://bugs.chromium.org/p/chromium/issues/detail?id=1154537.

[23] Undefinedbehaviorsanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[24] Issue 1237040: The thin line is drawn when using <colgroup>. https://bugs.chromium.org/p/chromium/issues/detail?id=1237040#c8.

[25] Wadi. https://github.com/sensepost/wadi.

[26] web-platform-tests documentation. https://web-platform-tests.org/.

[27] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert. Nautilus: Fishing for deep bugs with grammars. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[28] T. Brennan, S. Saha, and T. Bultan. Jvm fuzzing for jit-induced side-channel detection. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, Seoul, South Korea, June–July 2020.

[29] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[30] P. Chen, J. Liu, and H. Chen. Matryoshka: fuzzing deeply nested branches. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[31] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao. Coverage-directed differential testing of jvm implementations. In *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Santa Barbara, CA, June 2016.

[32] Y. Chen, T. Su, and Z. Su. Deep differential testing of jvm implementations. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, Montreal, QC, Canada, May 2019.

[33] S. R. Choudhary, H. Versee, and A. Orso. Webdiff: Automated identification of cross-browser issues in web applications. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[34] S. R. Choudhary, M. R. Prasad, and A. Orso. Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 171–180. IEEE, 2012.

[35] L. Clark. Inside a super fast css engine: Quantum css (aka stylo). https://hacks.mozilla.org/2017/08/inside-a-super-fast-css-engine-quantum-css-aka-stylo/.

[36] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson. Automated testing of graphics shader compilers. *Proceedings of the ACM on Programming Languages*, 1 (OOPSLA):1–29, 2017.

[37] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun. Dlfuzz: Differential fuzzing testing of deep learning systems. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Lake Buena Vista, FL, Nov. 2018.

[38] C. Harrelson. Overview of the renderingng architecture. https://developer.chrome.com/blog/renderingng-architecture/#rendering-pipeline-structure.

[39] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2021.

[40] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[41] S. Mahajan and W. G. Halfond. Finding html presentation failures using image comparison techniques. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Västerås, Sweden, Sept. 2014.

[42] S. Mahajan and W. G. Halfond. Detection and localization of html presentation failures using computer vision-based techniques. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, 2015.

[43] A. Mesbah and M. R. Prasad. Automated cross-browser compatibility testing. In *Proceedings of the 33th International Conference on Software Engineering (ICSE)*, Honolulu, HI, May 2007.

[44] L. A. Meyerovich and R. Bodik. Fast and parallel webpage layout. In *Proceedings of the 19th International World Wide Web Conference (WWW)*, Raleigh, NC, Apr. 2010.

[45] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[46] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, June 2009.

[47] P. Panchekha and E. Torlak. Automated reasoning for web page layout. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Amsterdam, Netherlands, Nov. 2016.

[48] P. Panchekha, A. T. Geller, M. D. Ernst, Z. Tatlock, and S. Kamil. Verifying that web pages have accessible layout. In *Proceedings of the 2018 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Philadelphia, PA, June 2018.

[49] P. Panchekha, M. D. Ernst, Z. Tatlock, and S. Kamil. Modular verification of web page layout. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA): 1–26, 2019.

[50] K. Pei, Y. Cao, J. Yang, and S. Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.

[51] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. Nezha: Efficient domain-independent differential testing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[52] P. Porwik and A. Lisowska. The haar-wavelet transform in digital image processing: its status and achievements. *Machine graphics and vision*, 13(1/2):79–98, 2004.

[53] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, June 2012.

[54] J. Wang, B. Chen, L. Wei, and Y. Liu. Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[55] W. Xu, S. Park, and T. Kim. Freedom: Engineering a state-of-the-art dom fuzzer. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Virtual, USA, Nov. 2020.

[56] C. Zauner. Implementation and benchmarking of perceptual image hash functions. 2010.