JANUS: Detecting Rendering Bugs in Web Browsers via Visual Delta Consistency

Chijin Zhou†, Quan Zhang†, Bingzhou Qian‡, and Yu Jiang†✉
†BNRist, Tsinghua University, Beijing, China
‡National University of Defense Technology, Changsha, China

***Abstract*—Rendering lies at the heart of our modern web experience. However, the correctness of browser rendering is not always guaranteed, often leading to rendering bugs. Traditional differential testing, while successful in various domains, falls short when applied to rendering bug detection because an HTML file is likely yield different rendered outcomes across different browsers. This paper introduces Visual Delta Consistency, a test oracle to detect rendering bugs in web browsers, aiming to make rendered pages across browsers comparable. Our key insight is that any modifications made to an HTML file should uniformly influence rendering outcomes across browsers. Specifically, when presented with two HTML files that differ only by minor modifications, the reaction of all browsers should be consistent, i.e., either all browsers render them identically or all render them differently. Based on this insight, We implemented it as a practical fuzzer named JANUS. It constructs pairs of slightly modified HTML files and observes the change statuses of the corresponding rendered pages across browsers for bug detection. We evaluated it on three widely-used browsers, i.e., Chrome, Safari, and Firefox. In total, JANUS detected 31 non-crash rendering bugs, out of which 24 confirmed with 8 fixed.**

I. INTRODUCTION

Rendering lies at the heart of our modern web experience, responsible for translating code and media into visual web pages. While important, the correctness of rendering in web browsers is not always guaranteed. On the one hand, the ongoing evolution of web standards introduces a dynamic complexity to the task. Browser developers must constantly adapt their rendering engines to keep up with these changing standards. On the other hand, the need for high-speed rendering, which is crucial for an optimal user experience, further compounds this complexity. These requirements call for intricate implementations, which can give rise to *rendering bugs*. These bugs are logic errors that cause incorrect or unintended displays of web pages.

Unlike crashes, rendering bugs often do not present clear symptoms, making their detection a challenging task. In other testing research domains, differential testing has emerged as a common approach for non-crash bug detection. The key idea behind it is to compare the execution results from different implementations of the same application when given the same inputs. Due to its easy-to-implement and effective nature, differential testing has been extensively applied to detect logic bugs in various domains, such as compilers [43], [20], [19], protocols [23], [4], database management systems [40], [41], and machine learning systems [11], [44].

✉ Yu Jiang is the corresponding author.

Although effective in other domains, differential testing falls short when applied to rendering bug detection in browsers. The rationale is that the same HTML file is likely to yield different rendered pages in different browsers [1], [5], [32], [33], [35], which make the rendered pages across different browsers *incomparable*. According to our experimental study on testcases generated by a browser fuzzer Domato [9], 8,337 out of 10,000 testcases trigger different rendered pages in the latest versions of Chrome and Firefox, while almost all the cases are false positives after our manual verification. These false positives stem from a range of factors. For example, browsers vary in their support to web standards, meaning some may not fully support or interpret certain features, leading to disparate rendering outcomes. Even in cases where a feature is universally supported, differences in default styles or the rendered appearance of elements can contribute to inconsistencies. As a result, all existing rendering bug fuzzer [32], [33] avoid implementing cross-browser testing strategies to prevent false positives, leaving a significant gap in the field.

This paper introduces *Visual Delta Consistency*, a test oracle to detect rendering bugs in web browsers, aiming to make rendered pages across browsers comparable. Our key insight is that any modifications made to an HTML file should uniformly influence rendering outcomes across browsers. Specifically, when presented with two HTML files that differ only by minor modifications, the reaction of all browsers should be consistent, i.e., either all browsers render them identically or all render them differently. Fig. 1 provides an illustrative example of how this approach uncovers rendering bugs. In this figure, *html2* is the same as *html1* except for a "`:root{float:right}`" statement appended to the CSS style part. However, the rendered pages of *html1* and *html2* differ in Safari, while they remain the same in Chrome. We define this difference as a *visual delta*, which is the visual difference between the rendered pages of an HTML file and its modified version. An inconsistent *visual delta* indicates a rendering bug in one of the two browsers. The bug shown in Fig. 1 was confirmed and fixed by Chrome developers.

Implementing this test oracle as a practical and effective fuzzer is challenging because it requires careful construction of the visual delta in order to ensure the soundness of the test oracle to avoid false positives. In the meanwhile, it must be capable of exploring a wide range of rendering features to facilitate effective bug detection. To address these challenges, we first leverage web standards and browser compatibility

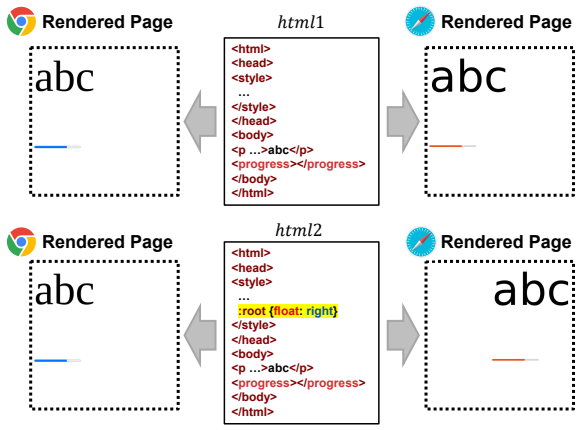


Fig. 1: A rendering bug in Chrome detected by JANUS. *html2* is a slightly modified version of *html1*. Chrome renders the two files identically, while Safari renders them differently.

to construct a universal set of rendering features that are supported by all the browsers under test. This set enables us to generate highly diverse HTML files where all rendering features are compatible with the browsers under test. Subsequently, we adopt a rendering-tree-based approach to select a node within the rendering tree and subtly alter its style attributes, creating a slightly modified version of the original HTML file. This pair of HTML files is then executed by the browsers under test, and the resulting visual deltas are analyzed to identify rendering bugs.

We implemented the above approach as a fuzzer named JANUS, and evaluated it on the latest versions of three widely-used browsers, i.e., Chrome, Safari, and Firefox. The results show that JANUS can detect 31 non-crash rendering bugs in these browsers, out of which 24 confirmed with 8 fixed by the developers. Compared to the state-of-the-art rendering bug fuzzer R2Z2 [32], JANUS detects 66.67% more rendering bugs after a 24-hour fuzzing session. Additionally, JANUS maintains a low false positive rate of 5.26% on average.

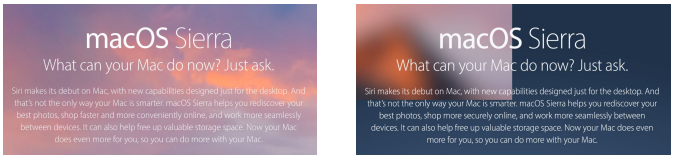
Overall, we make the following contributions:

- **New Test Oracle**. We proposed a new test oracle for rendering bug detection in browsers. This is achieved by constructing two HTML files with a minor modification and observing the visual delta across browsers.
- **Practical Fuzzer**. We designed and implemented a practical fuzzer named JANUS, which ensures the soundness of the proposed test oracle while efficiently exploring the all rendering features in browsers. We released the fuzzer at https://github.com/ChijinZ/janus-browser-fuzzer.
- **Real-World Bugs**. We evaluated JANUS on the latest versions of three widely-used browsers. In total, it has detected 31 non-crash rendering bugs with 24 confirmed.

II. BACKGROUND AND MOTIVATION

Browser Rendering One of browsers' fundamental tasks is to render web pages. This process begins when a browser retrieves the HTML, CSS, and JavaScript files for a given web page. The browser first parses the HTML into a structure known as the Document Object Model (DOM), which represents the hierarchical organization of the page's elements. Concurrently, it interprets the CSS into the CSS Object Model (CSSOM), which details the styling rules for these elements. These two structures are then combined into a render tree, which includes only the elements that will be displayed on the screen. The browser then proceeds to layout, where it calculates the exact position and size of each element, taking into account factors like screen size, viewport size, and font size. Finally, the browser paints the elements onto the screen, transforming the render tree into pixels.

The whole process is complex in functionality and requires intricate optimizations for user experience. Therefore, rendering bugs are a common occurrence in browsers. They can manifest in various forms, such as misaligned elements, incorrect colors, or missing content. Fig. 2 shows an example of a rendering bug in Chrome 55.0.2852.0 when rendering a real-world web page of Apple. The background image of the actual rendered page is incorrectly clipped because a wrong implementation of screen space scaling. Such bugs significantly affect user experience and puzzle web developers, as they need to determine whether the rendering issue arises from their code or the browser itself.

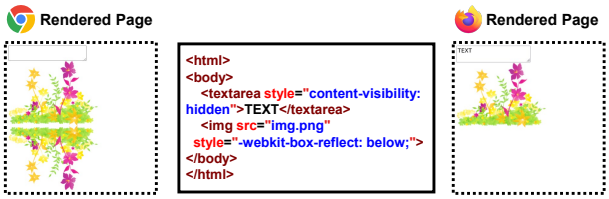


(a) Expected rendered page. (b) Actual rendered page.

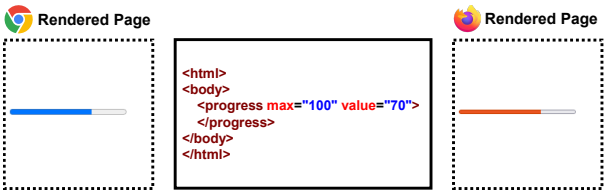
Fig. 2: A rendering bug in Chrome (issue #40483836) when rendering Apple's MacOS introduction page. The background image of the actual rendered page is incorrectly clipped.

Obstacle of Cross-Browser Testing. A common approach to detecting such logic bugs in other testing research domains is to perform *differential testing* [43], [19], [23], [11], which compares the behavior of different implementations of the same applications when given the same inputs. However, this approach is not applicable to rendering bug detection. Several research works [32], [33] have shown that cross-browser differential testing produces high false-positive rate and is not practical for rendering bug detection. The main reason is that *the same HTML file can be rendered differently by different browsers*. In other words, if we perform cross-browser differential testing and find an inconsistent outcome between browsers, it does not necessarily mean that there is a rendering bug. Instead, it could be due to the inherent differences in the browsers' rendering engines.

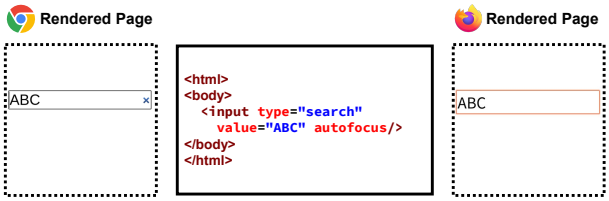
We conduct a preliminary study to illustrate this obstacle. We generate 10,000 testcases using Domato [9], a fuzzer for HTML file generation, and test them on Chrome and Firefox. 8,337 of the testcases trigger different rendered pages in the two browsers. However, After manual verification, we find that



(a) Support for some CSS styles varies cross browsers.



(b) Default style for some elements varies cross browsers.



(c) Rendered appearance for some elements varies cross browsers.

Fig. 3: Illustrative examples of why conventional differential testing fails in detecting rendering bugs: the same HTML file can be rendered differently by different browsers.

99.78% of them are false positives. This shows that directly applying cross-browser differential testing is infeasible.

From the false positives in the experiment, we observe that the main reasons are threefold. First, browsers vary in their adherence to web standards, meaning some may not fully support or interpret certain features, leading to disparate rendering outcomes. As Fig. 3a shows, the two HTML elements are rendered in a significantly different way in Chrome and Firefox. The `content-visibility` is an experimental CSS property and not fully supported by Firefox. Therefore, Firefox did not hide the text in the textarea as Chrome did. On the other hand, the `-webkit-box-reflect` is a special feature which is only supported by WebKit-family browsers such as Chrome and Safari. Therefore, Firefox did not reflect the flower image as Chrome did. Second, each browser applies its own default styles to HTML elements, which can significantly alter their appearance. As Fig. 3b shows, the progress bar is rendered as a thick blue bar in Chrome, while it is rendered as a thin orange bar in Firefox. Third, each browser draws the rendered appearance of elements in their own way, which can lead to different visual presentations. As Fig. 3c shows, when rendering a search-type input box, Chrome shows a cross icon that can be clicked on to remove the search term. In contrast, Firefox does not show this cross icon. In addition, when the input box is focused, Chrome draws a blue border around the input box, while Firefox draws an orange border.

Existing Approaches. The inherent inconsistencies across different web browsers have led existing research to sidestep cross-browser differential testing for rendering bug detection. Instead, alternative strategies have been adopted. R2Z2 [32] focuses on detecting rendering bugs by comparing the rendered pages of different versions of the same browser. Metamong [33], on the other hand, focuses on rendering-update bugs in browsers. It compares the rendered pages from two distinct rendering processes: traditional one-pass rendering versus incrementally-updated rendering. These approaches, despite their contributions to detecting a range of rendering bugs, each carry limitations. R2Z2 may overlook bugs that persist across all versions of a browser, while Metamong's scope excludes bugs that do not pertain to the rendering-update process. Thus, there's a need for an approach that complements and extends their capabilities in rendering bug detection.

Goal of This Paper. This paper aims to enable cross-browser differential testing for a more general rendering bug detection, and thus complement existing works. TABLE I summarizes the ability of JANUS compared to other related fuzzers. The key contribution of JANUS is to propose a broadly applicable test oracle called Visual Delta Consistency to make rendered pages across different browsers *comparable*. It is worth noting that the consistency is not limited to the comparison of different browsers, but also can be applied to the comparison of different versions of the same browser. Therefore, JANUS can also detect rendering regressions, which is focused by R2Z2. Metamong, on the other hand, is orthogonal to R2Z2 and JANUS because it is specialized in testing the rendering update process, which is not the focus of JANUS.

TABLE I: Features of fuzzers for rendering bug detection.

Fuzzer	Focus	CV	RU	CB
R2Z2	rendering regressions	✓	✗	✗
Metamong	render-update bugs	✗	✓	✗
JANUS	general rendering bugs	✓	✗	✓

CV: cross-version diff testing; **RU**: activate render-update process; **CB**: cross-browser diff testing.

III. VISUAL DELTA CONSISTENCY

A. Basic Idea

We propose *Visual Delta Consistency*, a simple but broadly applicable test oracle, to enable cross-browser differential testing. Our key insight is that any modifications made to an HTML file should uniformly influence rendering outcomes across browsers, i.e., either all browsers render them identically or all render them differently. This consistency stands only when all rendering features, including CSS styles, HTML elements, and HTML attributes, used in the two HTML files are supported by the browsers under test. Otherwise, the visual delta may change on one browser but not change on another due to unsupported features instead of browser rendering bugs, which leads to false positives. We will give a formal definition of visual delta consistency in what following.

Definition 1 (HTML Transformation)**.** Given an HTML file h_1, an HTML transformation $trans$ is a function that transforms one rendering style (CSS style or HTML attribute)

property $prop$ of an element el in the rendering tree of h_1 from the value v_1 to the value v_2, and outputs a modified HTML h_2, denoted as $h_2 = trans_{el,prop}^{v_1 \to v_2}(h_1)$.

Definition 2 (Visual Delta)**.** Given two HTML files h_1 and h_2, the visual delta $\Delta r_b^{h_1,h_2}$ is the difference between the rendered pages of h_1 and h_2 on browser b. $\Delta r_b^{h_1,h_2} = 1$ denotes the rendered pages are visually different, while $\Delta r_b^{h_1,h_2} = 0$ denotes the rendered pages are visually the same.

Definition 3 (Visual Delta Consistency)**.** Given an HTML file h_1 and its transformed version $h_2 = trans_{el,prop}^{v_1 \to v_2}(h_1)$, for arbitrary browsers b_1 and b_2, if (1) all rendering features used in h_1 and h_2 are supported by b_1 and b_2, and (2) $trans_{el,prop}^{v_1 \to v_2}(*)$ have the same rendering effect in b_1 and b_2, then the equation $\Delta r_{b_1}^{h_1,h_2} = \Delta r_{b_2}^{h_1,h_2}$ will always stand.

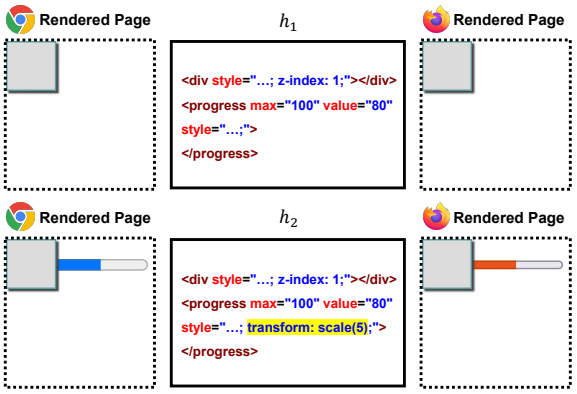
B. Soundness

Visual delta consistency can serve as a test oracle for rendering bug detection, i.e., we can continuously generate HTML files and their transformed versions, and then compare the visual deltas across browsers. If the visual deltas are inconsistent, we can conclude that there is a presence of rendering bugs. However, the effectiveness of visual delta consistency hinges on its soundness as a test oracle, this test oracle is supposed to offer the right verdict for a given test case [2]. If it is not sound, the detection will be ineffective due to considerable false positives. Since browser rendering bug detection is very special, even the same HTML file can produce different rendered pages across browsers, the soundness of visual delta consistency is not straightforward. Therefore, we will give a proof to show that visual delta consistency is sound for rendering bug detection.

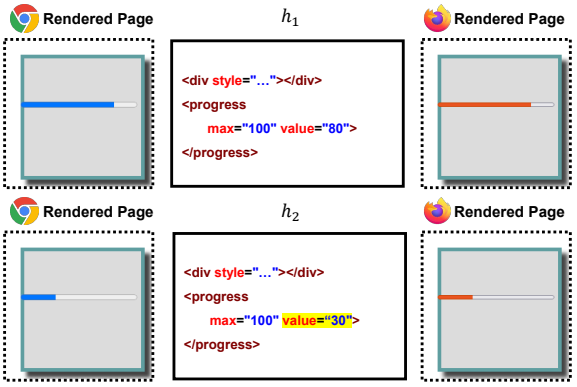
Proposition 1. *(Soundness) Visual delta consistency always stands for any two web browsers.*

Proof. Suppose we have an HTML file h_1, and its transformed version $h_2 = trans_{el,prop}^{v_1 \to v_2}(h_1)$. All rendering features used in h_1 and h_2 are supported by the two browsers b_1 and b_2, and $trans_{el,prop}^{v_1 \to v_2}(*)$ have the same rendering effect in b_1 and b_2. We want to prove that $\Delta r_{b_1}^{h_1,h_2} = \Delta r_{b_2}^{h_1,h_2}$ for any two browsers b_1 and b_2. Consider the following two situations:

- $trans.el$ is **invisible** in the rendered page of h_1.
  In this situation, the element $trans.el$ may be set as invisible in its rendering style, obscured by other elements, or outside the viewport, preventing the element from being observed. Because of all rendering features are supported by both browsers, $trans.el$ is invisible no matter rendered by which browser. After applying the transformation, we have two cases to consider: (1) If the changing of $trans.prop$ from v_1 to v_2 is not supposed to change the visibility of $trans.el$, then the rendered pages of h_1 and h_2 will be obviously the same on both b_1 and b_2, i.e., $\Delta r_{b_1}^{h_1,h_2} = \Delta r_{b_2}^{h_1,h_2} = 0$. (2) If the changing is supposed to make $trans.el$ visible, then the rendered pages of h_1 and h_2 will be different on both b_1 and b_2, i.e., $\Delta r_{b_1}^{h_1,h_2} = \Delta r_{b_2}^{h_1,h_2} = 1$, because the change have the same rendering effect in both browsers. This visibility changing can be achieved by setting the visibility property, position property, size property, or z-index property to a proper value. Take Fig 4a as an example, the progress element initially is hidden by the div element in the rendered page of h_1. After adjusting its scale, it becomes visible in the rendered page of h_2. Although the two browser render the page differently in the progress's color, they should react in the same way for the change.



(a) Change an element from being invisible to visible.



(b) Change an element's appearance.

Fig. 4: Illustrative examples for visual delta consistency.

- $trans.el$ is **visible** in the rendered page of h_1.
  In this situation, we can directly observe the element $trans.el$ in the rendered page no matter which browser renders the page. After applying the transformation, we have two cases to consider: (1) If the changing of $trans.prop$ from v_1 to v_2 is not supposed to have explicit visual effect, then the rendered pages of h_1 and h_2 will be obviously the same on both b_1 and b_2, i.e., $\Delta r_{b_1}^{h_1,h_2} = \Delta r_{b_2}^{h_1,h_2} = 0$. (2) If the changing is supposed to have explicit visual effect, then the rendered pages of h_1 and h_2 will be different on both b_1 and b_2, i.e., $\Delta r_{b_1}^{h_1,h_2} = \Delta r_{b_2}^{h_1,h_2} = 1$, because the change have the same rendering effect in both browsers. This visual effect can be achieved by setting the visibility property, color property, font-size property, or visual-related attribute to a proper value. Take Fig. 4b as an example, the value attribute of the progress element is changed from 80

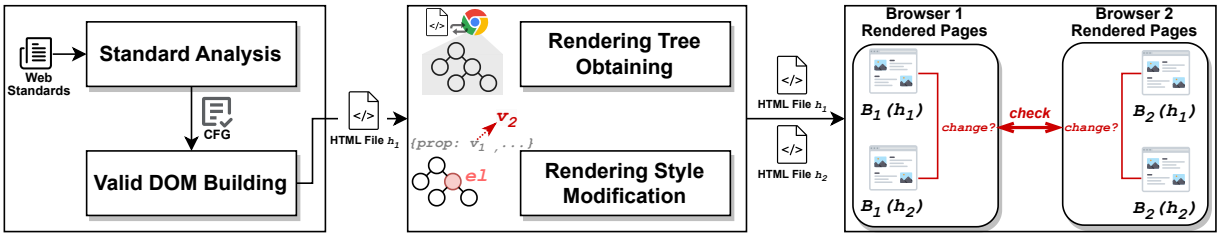


Fig. 5: The overall design of JANUS. First, it analyzes web standards and browser compatibility to generate an initial HTML file h_1. Then, it constructs a modified HTML file $h_2 = trans_{el,prop}^{v_1 \to v_2}(h_1)$ by modifying the value of a rendering style property for a selected node within h_1's rendering tree. Finally, it executes h_1 and h_2 in multiple browsers. By checking the consistency of the visual deltas across browsers, it can identify rendering bugs in these browsers.

to 30. Although the two browser render the page differently in the progress's color, they should react in the same way for the change.

To sum up, both situations lead to the same conclusion that $\Delta r_{b_1}^{h_1,h_2} = \Delta r_{b_2}^{h_1,h_2}$ for any two browsers b_1 and b_2. Therefore, visual delta consistency always stands.

□

IV. DESIGN OF JANUS

JANUS is designed to apply visual delta consistency as a test oracle for browser rendering bug detection. Based on the soundness of visual delta consistency, we can deduce that if two slightly modified HTML files, where all rendering features are supported by the browsers under test, yield inconsistent visual deltas across different browsers, it suggests that these modifications do not have consistent rendering effects across browsers, being an indicator of potential rendering bugs.

However, implementing JANUS as a practical and effective fuzzer presents unique challenges. First, the soundness of visual delta consistency may not be guaranteed if the constructed HTML files use rendering features unsupported by the browsers under test, potentially resulting in false positives. Second, a lack of diversity in the constructed HTML files, while maintaining soundness, could limit the fuzzer's ability to explore a wide range of rendering features. This might lead to missed opportunities in detecting certain rendering bugs.

Fig. 5 illustrates the overall design of JANUS, which consists of three main modules: (1) *Initial DOM Generation*, which generates initial HTML files that are compatible with the browsers under test and able to cover a wide range of rendering features; (2) *HTML Transformation*, which properly constructs a modified HTML file of the initial HTML file while ensures the soundness of visual delta consistency; and (3) *Consistency Checking*, which observes the visual deltas across browsers and identify rendering bugs. We will detail the design of each module in the following subsections.

A. Initial DOM Generation

This module focuses on generating high-quality, initial HTML files. It ensures that all rendering features used are supported by the target browsers and that the files are diverse

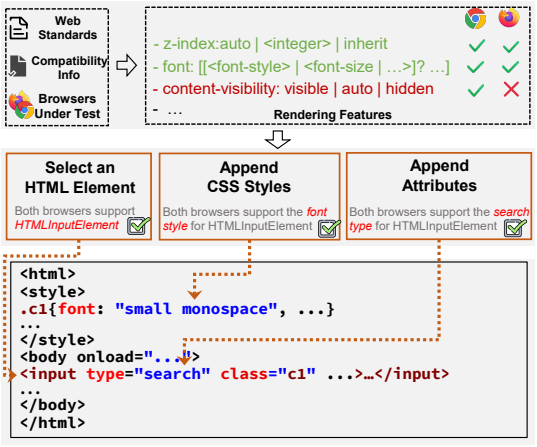


Fig. 6: The workflow of initial DOM generation.

enough to test a broad spectrum of rendering features. The process begins with an analysis of web standards and browser compatibility, identifying a set of usable rendering features. This analysis instructs the generation of HTML files, which are then utilized in the subsequent phases. Fig. 6 shows the workflow of this module.

Standard Analysis. The module first finds the intersection of all rendering features supported by the two browsers under test. We leverage Webref [34] to obtain the latest stable W3C standards, which fully defines all rendering features that browsers are supposed to implement. Rendering features not defined in the Webref are regarded as either non-standard or deprecated, and thus are excluded from our analysis. Although Webref defines a universal set of standards, not all features may receive consistent support across different browsers, as illustrated in Fig 3a. Therefore, we leverage data from MDN [21], a comprehensive web development documentation, to identify the compatibility of rendering features across browsers. These analyses pinpoint a set of rendering features that are supported by the browsers under test, which will be used in the subsequent HTML generation.

Valid DOM Building. This step aims to generate valid HTML files based on the rendering features obtained from the standard analysis. Since HTML is a highly structured

language, we conduct a model-based approach to generate valid HTML files. First, we convert the rendering features into a context-free grammar (CFG). For example, CSS style `z-index` in the standard can be converted to the following CFG production rules:

- `<line> := <selector> {<css_rule>, <css_rule>, ...};`
- `<css_rule> := z-index: <css_property>;`
- `<css_property> := auto | <integer> | inherit.`

Based on the production rules, JANUS is able to generate CSS code such as "`c1 { z-index: auto }`" and "`c1 { z-index: 20 }`". We leverage Domato [9], a DOM fuzzer, to generate valid HTML files based on the converted CFG. As illustrated in Fig. 6, the core components of the generated HTML files include HTML elements, CSS styles, and HTML attributes. For HTML elements such as `<input></input>` in the figure, we need to check the compatibility of the used elements and ensure both browsers support them. For CSS styles such as `font: "small monospace"` in the figure, we not only check the compatibility of both CSS properties and the values of corresponding properties, but also ensure they can be applied to the used elements according to the standards. For HTML attributes such as `type="search"` in the figure, similar to CSS styles, we need to check the compatibility and applicability of the used attributes. Through these meticulous checks, , this module can generate high-quality initial HTML files where all features are supported by the browsers under test.

B. HTML Transformation

This module is responsible for constructing an HTML file that is slightly modified from the initial HTML file. First, it dynamically runs the initial HTML file in the browsers under test to obtain a rendering tree. Upon the rendering tree, it selects a node and a rendering style property (CSS style or HTML attribute) to modify. The whole modification process is guided by the standard analysis to ensure the soundness of the test oracle. In the end, it outputs a modified HTML file that is used in the subsequent consistency checking module. Fig. 7 shows the workflow of this module.

Rendering Tree Obtaining. The rendering tree of an initial HTML file is obtained by running the file in one of the browsers under test. After the page of the initial HTML file is fully rendered, we inject a JavaScript code to inspect the rendering tree and the computed styles of each node in the tree. This process is achieved by leveraging Selenium [31], a tool for web browser automation, to monitor the rendering completion and execute custom JavaScript code. The computed styles contain all the rendering properties of each node, indicating how the browser renders the node. For the properties that are explicitly specified in the initial HTML file, the computed styles will inherit the specified values. For the properties that are not explicitly specified, the computed styles will inherit the default values of the browser.

Rendering Style Modification. After obtaining the rendering tree, we first select a node, i.e., an element, from the rendering tree as modified target. Next, based on its element

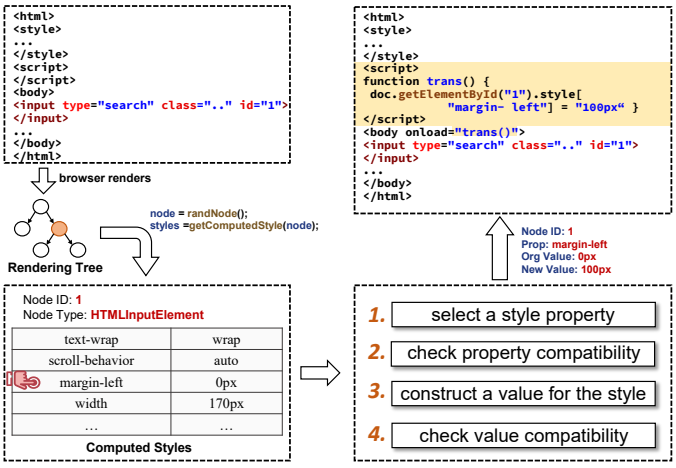


Fig. 7: The workflow of HTML transformation. It first selects a node from the rendering tree, and then modify the value of a rendering style property. The value should be valid and compatible with target browsers.

type and computed styles, we select a rendering style property and modify its value. The modification process is guided by the standard analysis to ensure both the property and constructed value are supported by the browsers under test and can be applied to the selected element type. For example, as depicted in Figure 4, the `HTMLInputElement` is targeted for modification, with its margin-left property chosen for adjustment. The property's value is altered from the default `0px` to `100px`. Following this, based on the modification information, we construct a modified HTML file by injecting corresponding JavaScript code, i.e., function `trans`, into the initial HTML file.

It is worth noting that we leverage rendering trees for HTML transformation instead of statically rewriting some fields in the initial HTML file. The rationale is that the rendering tree contains a wealth of additional properties that can be modified. For example, Even when the initial HTML file does not specify any styles, the browser's default styling rules and the cascading nature of CSS result in a rich set of computed styles within the rendering tree. By tapping into this extensive set of properties, we significantly expand the range of possible transformations.

C. Consistency Checking

This module is responsible for observing the visual deltas across different browsers and identifying rendering bugs. Specifically, given two browsers b_1, b_2 and an HTML file h_1 along with its modified version $h_2 = trans_{el,prop}^{v_1 \to v_2}(h_1)$, this module checks if $\Delta r_{b_1}^{h_1,h_2} = \Delta r_{b_2}^{h_1,h_2}$. The visual delta $\Delta r_b^{h_1,h_2}$ requires a visual comparison of the rendered pages of h_1 and h_2 in browser b. First, we leverage Selenium [31] to render the HTML files in the browsers under test and take screenshots of the rendered pages. During this process, we set the window size to be the same for all browsers to ensure the consistency of screenshots. Next, we adopt phash [27], a perceptual hash algorithm, to compare the screenshots of

rendered pages. If the phash values of the screenshots are different, it indicates that the rendered pages are visually different. If we observe different visual deltas across different browsers, it suggests that the modifications of h_2 do not have consistent rendering effects across browsers, being an indicator of potential rendering bugs.

V. Evaluation

In this section, we evaluate the effectiveness of JANUS by conducting a series of experiments. Our evaluation addresses the following research questions:

- **RQ1**: Can JANUS uncover unknown rendering bugs in mainstream browsers? (Section V-A)
- **RQ2**: How well does JANUS perform compared to other state-of-the-art fuzzers? (Section V-B)
- **RQ3**: Does JANUS introduce false positives during testing? (Section V-C)
- **RQ4**: What is the overhead introduced by JANUS during testing? (Section V-D)

Experiment Setup. We performed our evaluation on a machine equipped with an i7-12700KF with 12 cores, running Ubuntu 22.04 LTS. We utilized X virtual frame bffer (Xvfb) to enable browsers to run in headless mode on separate virtual display hardware. We selected Chrome, Firefox, and Safari as our fuzzing targets because they are the ones of the most representative browsers. Since Safari cannot be run on a Linux system, we used WebKitGTK, a full-featured port of Safari's rendering engine, as an alternative. We used ImageHash [3] to compute phash values of images. Once the phash values are different, we consider the corresponding rendered pages to be visually different. We set the timeout for each rendering test execution to 5 seconds.

A. Bug finding

We intermittently ran JANUS for a month to test the latest versions of the three mainstream browsers, i.e., Chrome 100.0.4896, Safari 15, and Firefox 100.0.2. We conducted two separated instances of JANUS, one for Chrome-Safari cross-browser testing and the other for Chrome-Firefox cross-browser testing. It is worth noting that we only reported bugs that violate visual delta consistency. Although a number of crashes were found during testing, we did not report them because crash bug detection is not the contribution of JANUS. Therefore, all the bugs shown in the following statistics are **non-crash rendering bugs**.

TABLE II: Statistics of rendering bugs reported by JANUS.

Browser	Reported	Confirmed	Duplicated	Fixed
Chrome	8	6	2	4
Safari	21	16	3	4
Firefox	2	2	0	0
Total	31	24	5	8

TABLE II presents the overall status of the bugs reported by JANUS. We have filed a total of 31 bugs for the three browsers, including 8 in Chrome, 21 in Safari, and 2 in Firefox. Out of these reported rendering bugs, 24 bugs have been confirmed with 8 already fixed by the developers. 5 bugs were independently reported by other contributors, annotated as "Duplicated" in TABLE II. TABLE III categorizes the reported rendering bugs by the affected component. In total, 3 bugs are related to the incorrect content rendering, 1 bug is related to wrong DOM element rendering, 7 bugs are related to form presentation, 10 bugs are related to layout, and 10 bugs are related to painting (namely CSS style drawing). These results collectively demonstrate the effectiveness of JANUS in detecting rendering bugs in browsers' various components.

TABLE III: Numbers of rendering bugs reported by JANUS in the three browsers, categorized by the affected component.

Browser	Affected Component				
	Content	DOM	Forms	Layout	Paint
Chrome	1	0	0	5	2
Safari	1	1	7	4	8
Firefox	1	0	0	1	0
Total	3	1	7	10	10

Through manual verification, we have confirmed that among the bugs reported by JANUS, only four are recently-introduced regression bugs. The remainder have existed since the initial implementation of certain rendering features in browsers. Therefore, R2Z2 [32], a fuzzer specifically designed to uncover regression bugs, is unable to detect these bugs. In addition, we have also manually confirmed that none of the reported bugs relates to the rendering-update process. Therefore, Metamong [33], another fuzzer that focuses on detecting rendering-update bugs, is also unable to detect these bugs.

Case Study 1. Fig. 8 presents a simplified rendering bug detected by JANUS in the paint component in Chrome. The bug is caused by the insufficient support for the `outline-style` property in Chrome. The `outline-style` property is used to set the style of the outline around an element. When the

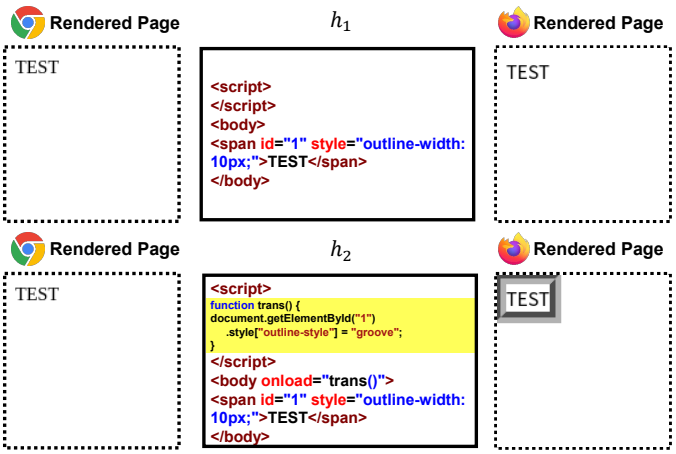


Fig. 8: Case 1: a simplified rendering bug in Chrome due to groove display error on some special elements.

value of `outline-style` is set to `groove`, the outline should be displayed as a 3D grooved border. According to web standards, the `outline-style` property and the `groove` value can be applied to any element, including the `<span>` element. However, after setting the `outline-style` to `groove` for a `<span>` element, Chrome fails to render the outline, which is inconsistent with the rendering in Safari. According to developers' feedback, the root cause is that the `box_border_painter` in Chromium's code did not paint the color when drawing the groove border under certain circumstance. Developers have confirmed and fixed this bug. Based on the testcase reported by us, developers also added a new integration test to the Chromium code base.

Case Study 2. Fig. 9 presents a simplified rendering bug detected by JANUS in the layout component in Chrome. According to web standards, when the value of `float` is set to `right` for any element, its child elements should be floated to the right side of its containing block. Therefore, once the value is set to `right` for the root element, all the child elements are supposed to be floated to the right side of the viewport. However, Chrome fails to render the child elements as expected, which is inconsistent with the rendering in Safari. According to developers' feedback, the root cause is that the internal layout object of the root element in Chrome is `LayoutView`, which is a legacy object and cannot be applied to some style properties. Developers have confirmed and fixed this bug by migrate the legacy object of the root element to their new layout engine, i.e., LayoutNG [10]. Based on the testcase reported by us, developers also added a new integration test to the Chromium code base.

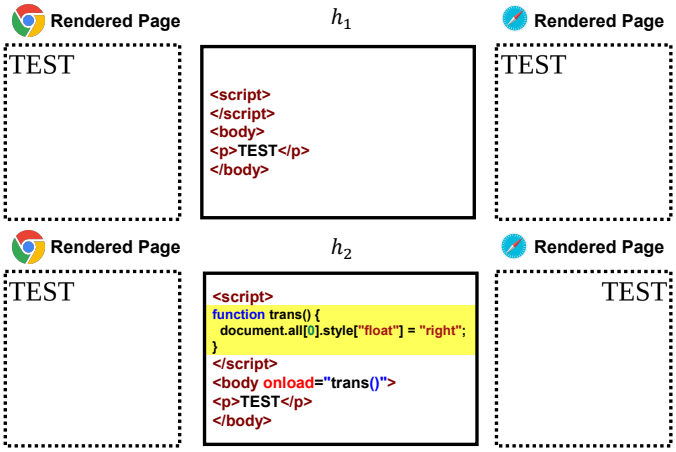


Fig. 9: Case 2: a simplified rendering bug in Chrome due to outdated implementation for the root element in HTML.

Case Study 3. Fig. 10 presents a simplified rendering bug detected by JANUS in the content component in Safari. The `white-space` property is used to specify how white space inside an element is handled. When the value of `white-space` is set to `pre-line`, the element should collapse sequences of white space, but preserve line breaks. According to web standards, the `white-space` property

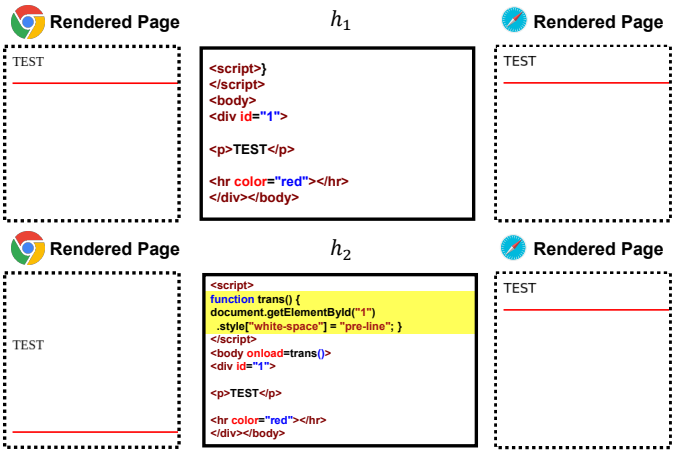


Fig. 10: Case 3: a simplified rendering bug in Safari due to incorrect `white-space` implementation for text containers.

and the `pre-line` value can be applied to any element, including the `<div>` element. In the code in h_1, within the `<div>` element, there are several empty lines before the `<p>` and between the `<p>` and `<hr>` elements. Before setting the `white-space` property, the empty lines are collapsed, and thus there is no empty line in the rendered pages in both Chrome and Safari. After setting the `white-space` property to `pre-line`, the empty lines should be preserved, and thus there should be empty lines in the rendered pages in both Chrome and Safari. However, Safari fails to render the empty lines as expected, which is inconsistent with the rendering in Chrome. According to developers' feedback, the root cause is an improper handling for the `white-space` property when applying to an inline-empty container. There is no text content directly in the `<div>` element since the `<hr>` element is a horizontal rule and the `<p>` element is a child of the `<div>` element. Therefore, the handling logic mistakenly thinks that it does not need to further process the `white-space` property, and thus does not draw any space in the rendered page. Developers have confirmed and fixed this bug. Based on the testcase reported by us, developers also added a new integration test to the WebKit code base.

B. Comparative Study

To further investigate the effectiveness of our approach, we evaluate JANUS compare to the most closely related fuzzer R2Z2 [32]. R2Z2 utilizes a cross-version differential testing approach, which involves comparing the rendered results between two different versions of a web browser to identify rendering regression bugs.

R2Z2 is designed for Chrome and lack support for other browsers, and thus we can only choose Chrome as the target for this comparative study. The Chrome version we used is 100.0.4896. For JANUS, we used Firefox 100.0.2 as the reference browser for cross-browser testing. For R2Z2, we used Chrome 99.0.4787 (the last stable release before the target version) as the reference version for cross-version testing. This reference selection strategy is in line with the evaluation of

R2Z2 paper [32]. We ran each fuzzer on the same machine for 24 hours and repeated five times. We collected the bugs reported by each fuzzer. Since the bugs are non-crash logic bug, we manually analyzed the root cause of each bug for de-duplication.

TABLE IV: Numbers of bugs reported by fuzzers in 24 hours.

	JANUS	R2Z2	Intersection
# of Bugs	8	6	4

TABLE IV presents the numbers of reported bugs after de-duplication. In total, JANUS reported 8 bugs, R2Z2 reported 6 bugs. Out of these bugs, 4 bugs were commonly reported by both fuzzers. The bugs reported by R2Z2 are caused by the recent commits of Chrome, and thus R2Z2 can detect them by comparing two versions of Chrome. Conversely, JANUS reported 4 bugs that failed to detect due to these bugs being present since the initial development of certain rendering features in Chrome. Therefore, comparing two close versions of Chrome cannot reveal these bugs. These experimental results demonstrate that JANUS's effectiveness in uncovering rendering bugs, including those that have long been embedded in the browser's code and overlooked by other fuzzers.

We were unable to directly compare JANUS with another related fuzzer, Metamong [33], due to missing some bug-triage components in its codebase. Theoretically speaking, JANUS and Metamong operate on totally different premises. Metamong is specifically engineered to identify rendering-update bugs by contrasting the outputs of two distinct rendering processes: traditional one-pass rendering versus incrementally-updated rendering. However, all testcases generated by JANUS do not involve the rendering-update process, and thus JANUS cannot detect the bugs reported by Metamong. Conversely, Metamong cannot detect the bugs reported by JANUS because they are not rendering-update bugs. To sum up, JANUS and Metamong offer complementary strengths, each focusing on distinct types of rendering bugs, mutually enhancing bug detection capabilities.

C. False Positives

Although visual delta consistency can serve as a sound test oracle for rendering bug detection, our implementation may still introduce false positives. Therefore, we conducted an experiment to investigate if JANUS report false positives during the testing. For comparison, we applied the same experiment to a naive cross-browser testing method, referred to as NAIVEDT. This approach runs the same HTML file in two different browsers and compares if the rendered pages are the same. We ran both fuzzers on Chrome 100.0.4896 and Firefox 100.0.2 for $10,000$ testcase executions.

To determine if a report is a false positive, we first minimized the test case automatically and then conducted a manual root-cause analysis. For bugs reported by JANUS, the minimized test cases were forwarded to developers to confirm their status. For bugs reported by NAIVEDT, if the minimized test case only involved a single element without additional CSS styling, it was deemed a false positive because this report stems from differences in browsers' default rendering or styling. For other cases, we manually identified the root cause and, if uncertain, sought developer verification.

TABLE V presents the numbers of reported false positives in the $10,000$ testcase executions. We can see that NAIVEDT reports an unacceptable false positive rate. As we detailed in Fig. 3, the main reasons are threefold: (1) the support for CSS styles varies cross browsers; (2) default style for some elements varies cross browsers; (3) rendered appearance of some elements varies cross browsers.

TABLE V: False positives reported by JANUS and its version that directly applies cross-browser differential testing.

Approach	#Testcase	#Report	#FP	FPR
NAIVEDT	10,000	8,337	8,319	99.78%
JANUS	10,000	19	1	5.26%

JANUS reports 1 false positives in 19 reports, resulting in a false positive rate of 5.26%, which is much better than the naive cross-browser testing. The primary cause of the false positive is behaviors that are undefined in web standards; thus, while rendering effects vary across browsers, these differences are due to distinct understanding of web standards rather than rendering bugs. Fig. 11 presents the false positive. We can see that there is a significant inconsistent change in the first `<progress>` element in the rendered pages of Chrome. However, after we reported this issue to the Chrome developers, they checked the source code of Chrome and Safari and confirmed that both browsers do not violate the web standards. They handle the transformation in a different way: after applying `border-top-right-radius` to a `<progress>` element, Chrome will render the element in a "primitive" mode rather than the original "native" mode. Since web standards do not prescribe a specific rendering mode under such circumstances, the developers concluded that

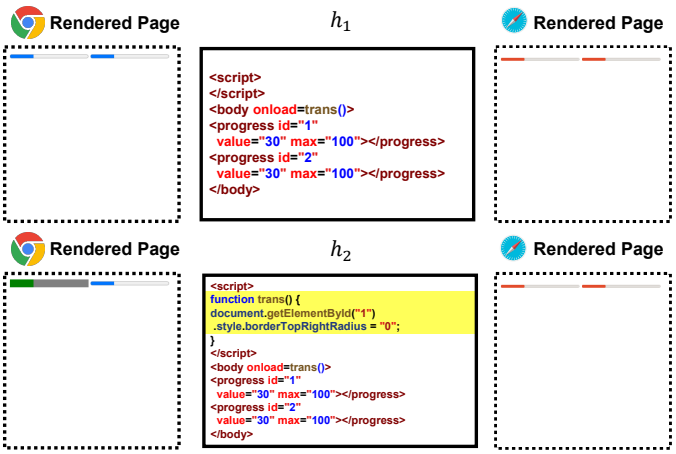


Fig. 11: A false positive reported by JANUS. It is caused by a behavior that is undefined in web standards.

this did not constitute a bug, thereby classifying it as a false positive. However, this rarely happens in practice, and thus JANUS is still a practical fuzzer for rendering bug detection.

D. Overhead Breakdown

Fuzzing throughput is a critical metric for evaluating the effectiveness of a fuzzer [38], [39], [49], [15], [22]. Therefore, we conducted an experiment to measure the average time taken for each step during testing. We use JANUS to test Chrome 100.0.4896 and Firefox 100.0.2 for $10,000$ testcase executions, and collected the time taken for each step. TABLE VI presents the overall result. On average, browser execution is the most time-consuming step, accounting for 87.03% of the total time. This part of overhead is not introduced by JANUS, but is the inherent time taken for the browser to render the HTML file. The time spent in initial DOM generation and HTML transformation are much less than the time in browser execution, accounting for 0.09% and 0.87% of the total time, respectively. Consistency checking needs to take screenshots of rendered pages and compute phash values, making a 12.01% overhead. Overall, we can conclude that JANUS's overhead only takes a small portion of the total time, and thus is acceptable for practical use.

TABLE VI: Average time taken for each step during testing.

	Initial DOM Generation	HTML Transformation	Browser Execution	Consistency Checking
Time (ms)	1.35	12.88	1286.45	177.45
Percentage	0.09%	0.87%	87.03%	12.01%

VI. DISCUSSION

Limitations. While JANUS marks a significant advancement in detecting rendering bugs in web browsers, it does possess inherent limitations. Its design as a cross-browser fuzzer means it cannot identify rendering bugs that manifest uniformly across all browsers. Furthermore, JANUS is not equipped to detect bugs lacking visual manifestations, such as functionality issues. Such non-visual bugs necessitate alternative testing strategies, like metamorphic testing. JANUS's focus is specifically on rendering bugs that both affect visual output and exhibit inconsistencies across different browsers.

In addition, JANUS cannot avoid false negatives, which is a typical limitation of differential oracles. If a transformation is applied and both browsers change their rendered pages, but one does so incorrectly, it cannot identify such bugs. Given that JANUS can detect rendering bugs that cannot be found by other tools, we regard it as a complementary tool to existing rendering bug detection techniques, rather than a replacement.

Identification of Buggy Browsers. Similar to other differential testing techniques, when a visual delta inconsistency is detected, JANUS cannot determine which browser is buggy. It can only identify that a rendering bug exists in one of the browsers. To pinpoint the buggy browser, we now manually refer to the relevant documentation and other browsers to understand the expected behavior, and determine the buggy browser. In the future, inspired by recent research [14], [18], [37], we plan to use large language models to automatically identify the buggy browser after detecting a visual delta inconsistency.

Feedback Mechanisms. JANUS did not adopt a feedback-driven fuzzing primarily for reasons of usability. Recent software practice has shown that coverage feedback can significantly improve the effectiveness of a fuzzer. However, some research [42], [51] indicates that coverage-guided browser fuzzing is not as effective as expected. The main reason is that a browser has multiple processes and each spawns many background threads that concurrently deal with input-unrelated tasks, e.g. resource requests from the network [51]. As a result, coverage differs in browsers even when the same invocation is executed. We believe that incorporating other feedback mechanisms is a promising direction for JANUS improvement. We leave it as future work.

Future Work. JANUS enables several opportunities for future work. First, a large language model could be used to reduce the number of false positives. For example, an LLM can read the descriptions in web standards to determine whether a given transformation is an undefined behavior. Second, JANUS's the current methodology of randomly selecting CSS style values or HTML attributes for modification could be refined by incorporating program analysis techniques to guide selections, thereby uncovering more nuanced rendering bugs. Third, the concept of visual delta consistency holds promise for application beyond web browsers, such as in GUI testing, where it could be used to identify discrepancies in GUI rendering across different platforms.

VII. RELATED WORK

Browser Fuzzing. As one of the most complicated applications, browsers naturally become an attractive target to security researchers and attackers. Finding browser vulnerabilities by fuzzing browser's sub-components such as third-party libraries [8] and JavaScript engines [26], [12] gained significant traction in academic research as well as in industry. In the meanwhile, defense mechanisms also have been proposed to protect browsers from attackers' vulnerabilities exploitation [48], [46], [47], [45]. Despite these efforts, the rendering engine in browsers has received relatively little attention. Whole browser fuzzers [9], [42], [51], [50], which aim to generate high-quality HTML files, represent the closest related work. For example, FreeDom [42] designs several generation and mutation strategies for maintaining context information during input generation; Minerva [51] tries to explore deeper paths by generating API-dependent invocations with mod-ref relations between APIs. Different from them, JANUS's key contribution is to propose new test oracle for rendering bug detection.

Rendering Bug Detection. The pursuit of detecting rendering bugs in web browsers has seen limited exploration. R2Z2 [32] focuses on detecting rendering bugs by comparing the rendered pages of different versions of the same browser.

Metamong [33], on the other hand, focuses on rendering-update bugs in browsers. It compares the rendered pages of two different rendering processes: one for one-pass rendering and the other for incrementally-updated rendering. JANUS, in contrast, introduces a novel approach by establishing visual delta consistency as a test oracle. This allows for the detection of rendering inconsistencies across different browsers, thereby filling a significant gap in the field.

Logic Bug Detection in Other Fields. The detection of logic bugs necessitates domain-specific test oracles. In various domains, researchers have developed innovative methods to tackle these bugs [36], [7], [16], [17], [13], [6]. Unlike crashes or memory corruptions, logic bugs are more difficult to detect and diagnose, and need domain-specific test oracles. For example, in web development, efforts [24], [25] have been made to formalize layout guidance of web development into a formal language for correctness verification; and in database management systems, tools like SQLancer [30], [29], [28] validate query results using predefined equivalence rules. Different from them, JANUS only focuses on rendering bug detection in web browsers. It uses browser-specific knowledge to construct a test oracle, i.e., visual delta consistency.

VIII. CONCLUSION

This paper introduces visual delta consistency as a test oracle for detecting rendering bugs in web browsers. This is achieved by constructing two HTML files with a minor modification and observing the visual delta across browsers. We implemented this approach as a fuzzer named JANUS, which ensures the soundness of the proposed test oracle while efficiently exploring all rendering features in browsers. JANUS has detected 31 bugs in widely-used browsers, with 24 confirmed by developers. Our future work is to further improve detection efficiency and reduce false positives.

IX. ACKNOWLEDGEMENT

We thank the anonymous reviewers for their insightful feedback. This research is sponsored in part by the National Key Research and Development Project (No.2022YFB3104000) and NSFC Program (No.92167101,62021002).

REFERENCES

[1] Ibrahim Althomali, Gregory M. Kapfhammer, and Phil McMinn. Automatic visual verification of layout failures in responsively designed web pages. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, pages 183–193. IEEE, 2019.

[2] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Trans. Software Eng.*, 41(5):507–525, 2015.

[3] Johannes Buchner. A python perceptual image hashing module. https://github.com/JohannesBuchner/imagehash, 2023. (visited on September 1, 2023).

[4] Yuting Chen and Zhendong Su. Guided differential testing of certificate validation in SSL/TLS implementations. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 793–804. ACM, 2015.

[5] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. WEBDIFF: automated identification of cross-browser issues in web applications. In Radu Marinescu, Michele Lanza, and Andrian Marcus, editors, *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, pages 1–10. IEEE Computer Society, 2010.

[6] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. Fuzzing deep-learning libraries via automated relational API inference. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 44–56. ACM, 2022.

[7] Wensheng Dou, Ziyu Cui, Qianwang Dai, Jiansen Song, Dong Wang, Yu Gao, Wei Wang, Jun Wei, Lei Chen, Hanmo Wang, Hua Zhong, and Tao Huang. Detecting isolation bugs via transaction oracle construction. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1123–1135. IEEE, 2023.

[8] Google. Ossfuzz. https://github.com/google/oss-fuzz, 2016. (visited on September 1, 2023).

[9] Google. Domato: A dom fuzzer. https://github.com/googleprojectzero/domato, 2017. (visited on September 1, 2023).

[10] Google. Layoutng. https://www.chromium.org/blink/layoutng/, 2023. (visited on September 1, 2023).

[11] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. Dlfuzz: differential fuzzing testing of deep learning systems. In Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 739–743. ACM, 2018.

[12] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[13] Pinjia He, Clara Meister, and Zhendong Su. Structure-invariant testing for machine translation. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 961–973. ACM, 2020.

[14] Yu Jiang, Jie Liang, Fuchen Ma, Yuanliang Chen, Chijin Zhou, Yuheng Shen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Shanshan Li, and Quan Zhang. When fuzzing meets llms: Challenges and opportunities. In Marcelo d'Amorim, editor, *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, pages 492–496. ACM, 2024.

[15] Shaohua Li and Zhendong Su. Accelerating fuzzing through prefix-guided execution. *Proc. ACM Program. Lang.*, 7(OOPSLA1):1–27, 2023.

[16] Shaohua Li and Zhendong Su. Finding unstable code via compiler-driven differential testing. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 238–251. ACM, 2023.

[17] Shaohua Li and Zhendong Su. Ubfuzz: Finding bugs in sanitizer implementations. In Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir, editors, *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pages 435–449. ACM, 2024.

[18] Tsz On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. Nuances are the key: Unlocking chatgpt to find failure-inducing tests with differential prompting. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 14–26. IEEE, 2023.

[19] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming*

Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023, pages 530–543. ACM, 2023.

[20] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for C and C++ compilers with yarpgen. *Proc. ACM Program. Lang.*, 4(OOPSLA):196:1–196:25, 2020.

[21] Mozilla. Mdn web docs. https://developer.mozilla.org/, 2005. (visited on September 1, 2023).

[22] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 787–802. IEEE, 2019.

[23] Pengbo Nie, Chengcheng Wan, Jiayu Zhu, Ziyi Lin, Yuting Chen, and Zhendong Su. Coverage-directed differential testing of X.509 certificate validation in SSL/TLS implementations. *ACM Trans. Softw. Eng. Methodol.*, 32(1):3:1–3:32, 2023.

[24] Pavel Panchekha, Michael D. Ernst, Zachary Tatlock, and Shoaib Kamil. Modular verification of web page layout. *Proc. ACM Program. Lang.*, 3(OOPSLA):151:1–151:26, 2019.

[25] Pavel Panchekha, Adam T. Geller, Michael D. Ernst, Zachary Tatlock, and Shoaib Kamil. Verifying that web pages have accessible layout. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 1–14. ACM, 2018.

[26] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing javascript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1629–1642. IEEE, 2020.

[27] phash team. The open source perceptual hash library. https://www.phash.org/, 2023. (visited on September 1, 2023).

[28] Manuel Rigger and Zhendong Su. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1140–1152. ACM, 2020.

[29] Manuel Rigger and Zhendong Su. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.*, 4(OOPSLA):211:1–211:30, 2020.

[30] Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 667–682. USENIX Association, November 2020.

[31] selenium team. Selenium automates browsers. that's it! https://www.selenium.dev/, 2023. (visited on September 1, 2023).

[32] Suhwan Song, Jaewon Hur, Sunwoo Kim, Philip Rogers, and Byoungyoung Lee. R2Z2: detecting rendering regressions in web browsers through differential fuzz testing. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1818–1829. ACM, 2022.

[33] Suhwan Song and Byoungyoung Lee. Metamong: Detecting render-update bugs in web browsers through fuzzing. In Satish Chandra, Kelly Blincoe, and Paolo Tonella, editors, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1075–1087. ACM, 2023.

[34] W3C. Webref - machine-readable references of terms defined in web browser specifications. https://github.com/w3c/webref, 2023. (visited on September 1, 2023).

[35] Thomas A. Walsh, Gregory M. Kapfhammer, and Phil McMinn. Automated layout failure detection for responsive web pages without an explicit oracle. In Tevfik Bultan and Koushik Sen, editors, *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 192–202. ACM, 2017.

[36] Jiannan Wang, Thibaud Lutellier, Shangshu Qian, Hung Viet Pham, and Lin Tan. EAGLE: creating equivalent graphs to test deep learning libraries. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 798–810. ACM, 2022.

[37] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Trans. Software Eng.*, 50(4):911–936, 2024.

[38] Mingzhe Wang, Jie Liang, Chijin Zhou, Yu Jiang, Rui Wang, Chengnian Sun, and Jiaguang Sun. RIFF: reduced instruction footprint for coverage-guided fuzzing. In Irina Calciu and Geoff Kuenning, editors, *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 147–159. USENIX Association, 2021.

[39] Mingzhe Wang, Jie Liang, Chijin Zhou, Zhiyong Wu, Xinyi Xu, and Yu Jiang. Odin: on-demand instrumentation with on-the-fly recompilation. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 1010–1024. ACM, 2022.

[40] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. Industry practice of coverage-guided enterprise-level DBMS fuzzing. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*, pages 328–337. IEEE, 2021.

[41] Zhiyong Wu, Jie Liang, Mingzhe Wang, Chijin Zhou, and Yu Jiang. Unicorn: detect runtime errors in time-series databases with hybrid input synthesis. In Sukyoung Ryu and Yannis Smaragdakis, editors, *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, pages 251–262. ACM, 2022.

[42] Wen Xu, Soyeon Park, and Taesoo Kim. FREEDOM: engineering a state-of-the-art DOM fuzzer. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 971–986. ACM, 2020.

[43] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011.

[44] Daniel Hao Xian Yuen, Andrew Yong Chen Pang, Zhou Yang, Chun Yong Chong, Mei Kuan Lim, and David Lo. ASDF: A differential testing framework for automatic speech recognition systems. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2023, Dublin, Ireland, April 16-20, 2023*, pages 461–463. IEEE, 2023.

[45] Quan Zhang, Yiwen Xu, Zijing Yin, Chijin Zhou, and Yu Jiang. Automatic policy synthesis and enforcement for protecting untrusted deserialization. In *Network and Distributed System Security (NDSS) Symposium (NDSS 2024)*, 2024.

[46] Quan Zhang, Binqi Zeng, Chijin Zhou, Gwihwan Go, Heyuan Shi, and Yu Jiang. Human-imperceptible retrieval poisoning attacks in llm-powered applications. In Marcelo d'Amorim, editor, *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, pages 502–506. ACM, 2024.

[47] Quan Zhang, Chijin Zhou, Yiwen Xu, Zijing Yin, Mingzhe Wang, Zhuo Su, Chengnian Sun, Yu Jiang, and Jia-Guang Sun. Building dynamic system call sandbox with partial order analysis. *Proc. ACM Program. Lang.*, 7(OOPSLA2):1253–1280, 2023.

[48] Chijin Zhou, Lihua Guo, Yiwei Hou, Zhenya Ma, Quan Zhang, Mingzhe Wang, Zhe Liu, and Yu Jiang. Limits of I/O based ransomware detection: An imitation based attack. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 2584–2601. IEEE, 2023.

[49] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 858–870. IEEE, 2020.

[50] Chijin Zhou, Quan Zhang, Lihua Guo, Mingzhe Wang, Yu Jiang, Qing Liao, Zhiyong Wu, Shanshan Li, and Bin Gu. Towards better semantics exploration for browser fuzzing. *Proc. ACM Program. Lang.*, 7(OOPSLA2):604–631, 2023.

[51] Chijin Zhou, Quan Zhang, Mingzhe Wang, Lihua Guo, Jie Liang, Zhe Liu, Mathias Payer, and Yu Jiang. Minerva: browser API fuzzing with dynamic mod-ref analysis. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 1135–1147. ACM, 2022.