Metamong: Detecting Render-Update Bugs in Web Browsers through Fuzzing

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

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

ABSTRACT

A render-update bug arises when a web browser produces an erroneous rendering output due to incorrect rendering updates. Such render-update bugs seriously harm the usability and reliability of web browsers. However, we find that detecting render-update bugs is challenging because the render-update bug is a semantic bug—given a rendering result, it is difficult to determine if it is correct due to the complex rendering specification of DOM and CSS. Thus, unlike memory corruption bugs, the incorrect rendering output does not raise the violation or crash. In practice, render-update bug detection relies on the time-prohibitive manual analysis of domain experts to determine the bug.

This paper proposes Metamong, an automated framework to detect render-update bugs without false positive issues via differential fuzz testing. Metamong features two key components: (i) page mutator, and (ii) render-update oracle. The page mutator generates render-update operations, which change the content of the web page, to trigger a render-update bug. The render-update oracle exploits an HTML standard rule, so-called yielding, to produce the correct rendering result of a given web page. Combining these components, Metamong creates two HTML files where each constructs the same web page, but only one of them induces the render-update. It then uses differential testing to compare their rendering outputs to determine a bug. We implemented a prototype of Metamong, which performs differential fuzz testing on popular browsers, Chrome and Firefox. By far, Metamong identified 19 new render-update bugs, 17 in Chrome and two in Firefox. All of those have been confirmed by each browser vendor and five are already fixed, demonstrating the practical effectiveness of Metamong in identifying render-update bugs.

CCS CONCEPTS

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

KEYWORDS

rendering update, web-browser, fuzz testing

*Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616336

ACM Reference Format:
Suhwan Song and Byoungyoung Lee. 2023. Metamong: Detecting Render-Update Bugs in Web Browsers through Fuzzing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616336

1 INTRODUCTION

The rendering performance has been the key requirement for the web browsers. Traditional browsers were very slow because they re-run the entire rendering process for every web page update. Unlike traditional browsers, in order to speed up the browser's rendering process, modern browsers employ a new technique, *render-update*, which re-renders only the updated part of the web page [16, 17]. This technique reuses previous rendering results and re-renders only the region that needs to be updated for efficiency. It is very suitable and useful for most of modern web applications because they frequently update their web page.

The problem is this technique introduces a bug where the browser does not re-render the areas of the web page that should be updated, generating an incorrect rendering output. We will call such a bug a *render-update* bug in this paper. A *render-update* bug is a bug that occurs when a web browser generates an incorrect rendering output due to an incorrect render-update of the browser. Render-update bugs can severely harm the usability and reliability of the web page and its service. In particular, this bug is very fatal to modern web applications because it arises from *render-update*, and modern web applications frequently invoke *render-update* to update their web pages. As an example, a major electric car company, Tesla, had a service interruption because of a *render-update* bug where Chrome 87 does not properly change the color of the car, disturbing users from its service [19].

There are two key challenges to find *render-update* bugs. First, it is challenging to generate javascript triggering *render-update* bugs. This is because the *render-update* bugs can be triggered only when the browser performs the incorrect *render-update*. Second, it is challenging to automatically detect *render-update* bugs without false positives. To be specific, there is no oracle that can determine whether the visual appearance of the web page violates HTML/CSS specifications [4, 10]. This is because it is difficult to translate complex HTML/CSS specifications into programmable expressions, which allow automated validation. Moreover, as some of the features for rendering in specifications are under-specified, the complete specification translation is infeasible. To solve this issue, previous works leveraged differential testing and proposed two testing methods: cross-browser testing, and cross-version testing [29, 30, 43, 49]. It is possible to adapt these methods to detect *render-update* bugs, but they have a critical limitation, a false positive issue. This is because the rendering outputs can be benignly different due to the under-specified features (e.g., table width distribution [18]) or the new feature implementation (e.g., CSS has() selector [1]).

In this paper, we propose Metamong, a framework tailored for detecting *render-update* bugs in modern browsers without the false positive issue. In order to address the aforementioned challenges, we design two key components: 1) page mutator, and 2) *render-update* oracle. The page mutator is used to trigger *render-update* bugs. As the *render-update* bug can be triggered only when the browser runs the *render-update*, the page mutator only generates *render-update* operations, which change the content of a web page such as DOM tree and CSS styles. By executing the *render-update* operations, Metamong makes the browser run *render-update* and enhances the chance of triggering *render-update* bugs. The *render-update* oracle can identify the *render-update* bugs without the false positive issues. The key insight of *render-update* oracle is each web page has exactly one rendering output regardless of whether it is created by the whole or partial rendering (i.e., *render-update*). To leverage this key insight, we exploit an HTML standard rule, yielding [10]. By exploiting yielding, Metamong can change the web page while preventing the browser from running *render-update*. By using these components, Metamong creates two HTML files that both build an identical web page, but only one of them causes a *render-update*. Metamong then uses differential testing to verify whether their rendering outputs are the same. Finally, Metamong determines a *render-update* bug if their rendering outputs are different.

We implemented the prototype of Metamong and conducted the evaluation on two popular browsers, Chrome and Firefox. During the evaluation, Metamong was able to detect all 28 previous *render-update* bugs that were reported within two years at Chrome and Firefox bug trackers (all 15 Chrome and all 13 Firefox bugs). More importantly, Metamong has found 21 *render-update* bugs (17 in Chrome and four in Firefox). All of the bugs were confirmed by the respective browser vendors, 19 of them were new bugs, and five of them were fixed, revealing Metamong's practical capability to detect *render-update* bugs without false positives.

To summarize, this paper makes the following contributions:

- **Design.** We designed Metamong, a framework to automatically detect browser *render-update* bugs without false positives. It features two components for *render-update* bugs: (i) a *render-update* oracle to detect *render-update* bugs and (ii) a page mutator to trigger *render-update* bugs.
- **Promising Results.** While performing the evaluation, Metamong can detect all of 28 previous *render-update* bugs obtained from Chrome and Firefox bug trackers. Importantly, it found 19 new *render-update* bugs in Chrome and Firefox. All of these were confirmed by the respective developers and five have already been fixed. These results suggest the strong practical aspects of Metamong for detecting *render-update* bugs in browsers.

2 BACKGROUND

2.1 Fuzzing and Differential Testing

Fuzzing. Fuzzing is a popular bug-finding method. It continually executes a target program with the randomly generated testcases to see if the target program's behavior is incorrect (e.g., crashing). Since fuzzing does not require domain expert knowledge of a target program, it is widely used in many software applications to detect bugs. Most fuzzers are developed to hunt for memory corruption bugs [2, 3, 5, 7, 8, 21–23, 25, 26, 36, 42, 50, 52, 53]. This is because the bug cannot be discovered by a fuzzer on its own; rather, a particular bug condition must be observed during fuzzing. Because of this, most fuzzing approaches have been presented to uncover memory corruption vulnerabilities, which are operated with memory error detectors that clearly define bug situations (or conditions) (e.g., ASAN [48] and UBSAN [20]).

Differential Testing. Due to the difficulty of expressing semantic bugs as a bug condition and the requirement for domain expert knowledge to identify them, fuzz testing alone is ineffective for detecting semantic bugs. In this sense, differential testing methods are widely used for detecting semantic bugs. More specifically, differential testing employs a number of programs, each of which is meant to get the same result for the same input. If the results are different for each program, we can determine that the program might include a semantic bug. As differential testing describes the bug condition of the semantic bugs, recent research have employed differential testing with fuzzing to uncover many types of semantic bugs. For instance, it is used to discover semantic bugs in CPU RTLs, SSL/TLS implementations, web browsers, debuggers, compilers, and Java virtual machines (JVM) [24, 27–30, 33–35, 39, 43, 47, 49, 54].

2.2 The Rendering of a Web Browser

Rendering is turning the content (e.g., HTML, CSS, and javascript) into the pixels (i.e., screen output) [11]. To render the content, modern browsers such as Chrome and Firefox build the data structure called a page. The page is the browser-specific memory object representing the HTML document. Each page consists of HTML, CSS, and javascript and has its own rendering output according to DOM and CSS specifications [4, 10]. We will call the rendering output as *render* in this paper. The page can be modified by various actions such as javascript and mouse clicks, and its rendering output also changes according to the updated page. Traditional browsers re-do the whole rendering process whenever a web page is modified (or updated). However, they have a critical limitation re-doing the whole rendering process is very heavy and increases the latency. This limitation seriously degrades the performance of the browsers and it also impacts most (modern) web applications as they frequently update their page.

To address this limitation, modern browsers build the rendering process into two phases: 1) render-initial, and 2) render-update [16, 17]. Render-initial is the first rendering phase that a browser loads (or parses) an HTML file, builds a page with it, and draws the page's corresponding output. To be specific, the render-initial is

$$\begin{aligned} p^{init} &\leftarrow BuildPage(h) \\ r^{init} &\leftarrow InitRender(p^{init}) \end{aligned} \tag{1}$$

where (i) $BuildPage$ builds the initial-page p^{init} based on the HTML file h, and (ii) $InitRender$ draws the initial-render r^{init} based on the initial-page p^{init}. Note that the *render-update* is not triggered in this phase.

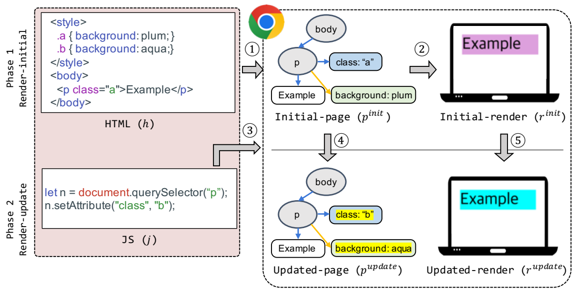


Figure 1: The example of the browser rendering process.

Render-update is the second rendering phase that the browser re-renders only the changed areas of the page whenever the page is updated. This approach reuses previous rendering results and only re-renders the changed parts of the page to efficiently generate the page's corresponding output. To be specific, the *render-update* is

$$p^{update} \leftarrow PageUpdate(p^{init}, j)$$
$$r^{update} \leftarrow RenderUpdate(r^{init}, p^{update}) \tag{2}$$

where (i) $PageUpdate$ executes the javascript j and updates the initial-page p^{init} to the updated-page p^{update}, and (ii) $RenderUpdate$ updates the initial-render r^{init} to the updated-render r^{update} which is the corresponding rendering output of p^{update}. As *render-update* does not render the entire area yet only re-renders the part of the area which should be updated, the browser can significantly decrease the performance overhead of the rendering with it. Besides, the *render-update* can be also performed on the updated-page p^{update} to efficiently generate the rendering output when the page is updated again through the javascript.

An example of the browser rendering process is shown in Figure 1. In phase 1, the browser loads the HTML file h and builds the initial-page p^{init} ①. It produces the initial-render r^{init} based on the initial-page p^{init} ②. Then, when the javascript code is executed, the browser conducts the second rendering phase. In this example, the javascript code changes the `class` attribute of <p> element node from "a" to "b" ③. As the value of the `class` attribute is changed to "b", the background color of the <p> node changes from "plum" to "aqua" and the initial-page becomes the updated-page p^{update} ④. Finally, the browser performs *render-update* based on the updated-page p^{update} and the initial-render r^{init} to generate the updated-render r^{update}. As only the background color of the <p> node is changed, the browser re-renders only the area of <p> node (i.e., plum rectangle) to change to the aqua color and it does not re-render the rest of the area ⑤.

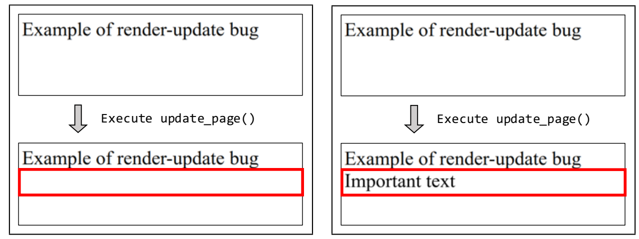
2.3 Render-Update Bug

The browser developers try to optimize the *render-update* by maximizing the reuse of previous rendering results and minimizing the areas to be re-rendered as possible. However, while making *render-update* efficient, the browser developers can introduce a bug that the browser incompletely re-renders the areas of the page that should be updated. We will call such a bug a *render-update* bug. A *render-update* bug is a bug where the browser incorrectly performs *render-update* on an updated page and generates an incorrect rendering output of the page, different from the DOM and CSS specifications. It is worth noting that the *render-update* bug and the rendering bug have different root-causes although both generate the incorrect rendering outputs which violate DOM/CSS specifications. Render-update bugs can harm users and web application developers in many ways. For instance, if the browser has a *render-update* bug and it is triggered on the web application, important elements that should be displayed may become distorted or invisible, severely harming the service quality. Furthermore, if elements are placed at awkward locations, users may not understand the contents or may be misguided. In addition, render-update bugs introduce unnecessary challenges to the web application developers. Suppose the developer found a certain bug in their web application. Then the developer starts to manually debug the issue, but such a debugging process is typically performed under the assumption that the underlying browser has no bugs. If this bug is a render-update bug, the developer will need to spend quite a time to finally notice it is a browser's issue or often fail to triage the bug.

We explain the example of a *render-update* bug with the HTML code shown in Figure 2. In this example, the Chrome browser first opens the HTML code and generates its rendering output (i.e., initial-render). It then executes `update_page()` at `line 3` to modify the page by setting the text of <div> element with `Important text`. Afterward, the Chrome browser performs *render-update* on the updated page and generates its rendering output (i.e., updated-render). If the Chrome browser works correctly, the text `Important text` (highlighted with the red box) should be drawn below the text

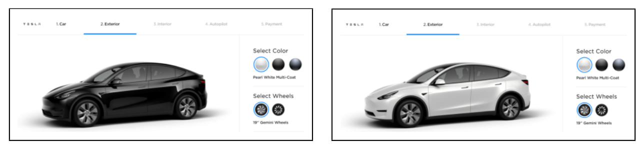
```
1 <!DOCTYPE html>
2 <script>
3   function update_page() { box.innerHTML = "Important text"; }
4 </script>
5 <style>  #box { transform: rotateY(0deg); height: 30px; } </style>
6 <body> Example of render-update bug  <div id="box"></div></body>
```

(a) PoC HTML code



(b) Actual Result (Incorrect). **(c) Expected Result (Correct).**

Figure 2: A *render-update* bug example (Chrome Issue #1167352).

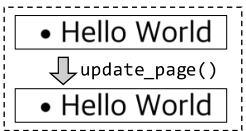


(a) Actual Result (Incorrect). **(b) Expected Result (Correct).**

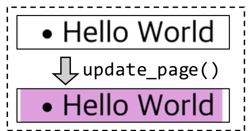
Figure 3: A *render-update* bug (Chrome Issue #1132218) triggered on Tesla's homepage. The car color does not change to white even after the user clicks the white-color button.

```html
<!DOCTYPE html>
<style> body {font-size: 30px;} </style>
<script>
  function update_page() {
    document.styleSheets[0].insertRule("ul:has(li) {background: plum;}")
  }
</script>
<body><ul><li>Hello World</li></ul></body>
```

(a) PoC HTML code



(b) Actual Result of Chrome 104 (Correct)



(c) Actual Result of Chrome 105 (Correct)

Figure 4: A false positive example of cross-version testing.

Example of render-update bug as shown in Figure 2c. However, the Chrome browser incorrectly performs *render-update*–it does not re-render the highlighted area because it determines that the highlighted area (which should be re-rendered) does not need to be re-rendered. As a result, the Chrome browser does not draw the text `Important text` and produces the incorrect rendering result as shown in Figure 2b.

The *render-update* bug can severely harm the usability and reliability of the page and its service if such a page is for commercial services. We explain such a case with a real-world example where the *render-update* bug was triggered on Tesla's homepage as shown in Figure 3. Initially, the black button on the homepage was selected, so the color of the car was black as well. Then the user can select the color to change the car color. When the user selects the white color, the browser should change the car color from black to white. However, the problem was that even though the user selects the white color (or other colors), the color does not change to white and remains black due to a *render-update* bug.

3 CHALLENGE AND APPROACH

3.1 Challenge

We elaborate on two challenges in triggering and detecting the *render-update* bugs.

Challenge #1: Triggering Render-Update Bugs. It is challenging to generate javascript that triggers *render-update* bugs. This is because the *render-update* bugs can be triggered only when the browser performs the *render-update*. If the javascript does not change the page, the browser never performs *render-update*, which is the origin of *render-update* bugs. To trigger the *render-update* bugs, the javascript needs to change the page and make the browser perform the *render-update*. Thus, to increase the chance to trigger *render-update* bugs, the generated javascript should induce the complex page and render change from the browser.

Challenge #2: Detecting Render-Update Bugs without False Positives. In order to detect *render-update* bugs, there should be an oracle that can tell the correctness of the *render-update*. However, it is very challenging to build such a *render-update* oracle because the *render-update* bug is a semantic bug. To be more specific, the *render-update* bug is triggered due to the semantically incorrect behavior of *render-update*. It entails incorrect rendering outputs, which violate DOM and CSS specifications [4, 6]. However, it is difficult to automatically validate their semantic correctness, because, unlike the memory corruption bug, the incorrect rendering output does not trigger the violation or crash. This means that it is impossible to detect whether the browser violates DOM and CSS specifications. Hence, detecting *render-update* bugs relies on the manual analysis of domain experts or the bug reports from the users and web application developers.

To resolve this issue, several studies leverage differential testing that compares the result of two different browsers for detecting rendering bugs. To be specific, the research area has proposed two testing methods by using differential testing: (i) cross-browser testing [29–31, 43] and (ii) cross-version testing [49]. Both methods can be used to find the *render-update* bugs, but they have a critical limitation–they suffer from the false positive issue on finding *render-update* bugs.

Cross-browser testing can be used to detect the *render-update* bugs by comparing the result of two independently-implemented browsers (e.g., Chrome and Firefox). It determines there is a *render-update* bug when two browsers generate different results from the same input consisting of HTML/CSS with javascript (which is used to trigger *render-update*). This method can be easily employed to detect *render-update* bugs because it does not require domain knowledge. However, R2Z2 [49] has shown that the result of two independently-implemented browsers can be different due to the benign browser incompatibilities (e.g., different feature support) so the cross-browser testing alone can trigger many false positives.

The cross-version testing compares the result of two different versions of the same browser (e.g., Chrome version 104 and 105) to detect the *render-update* bugs. It determines there is a *render-update* bug when two browsers generate different results from the same input consisting of HTML/CSS with javascript. Unlike cross-browser testing, it does not suffer from benign incompatibilities such as different supported features and different designs. This is because such incompatibilities are introduced when two browsers are independently-implemented. However, this approach still suffers from the false positive issue–the result of two different versions of the same browser can be benignly different due to the feature update or the bug fix. In other words, it is still challenging to determine which one is correct when two results are different.

We explain an example of a false positive case that can be caused by the feature update (i.e., CSS pseudo-class `:has()`) as shown in Figure 4. CSS pseudo-class `:has()` is not supported before Chrome 105 and it is first introduced in Chrome 105. In this example, Chrome 104 and 105 produce the same rendering outputs when opening PoC HTML code. After they execute `update_page()` to insert the CSS rule, their rendering outputs become different where only Chrome 105 paints the plum background on the <li> node. This is because `:has()` is not supported by Chrome 104 so Chrome 104 ignores the inserted CSS rule. The problem here is that even though both browsers produce the correct rendering outputs, the cross-version testing approach determines this case as *render-update* bug, leading to a false positive.

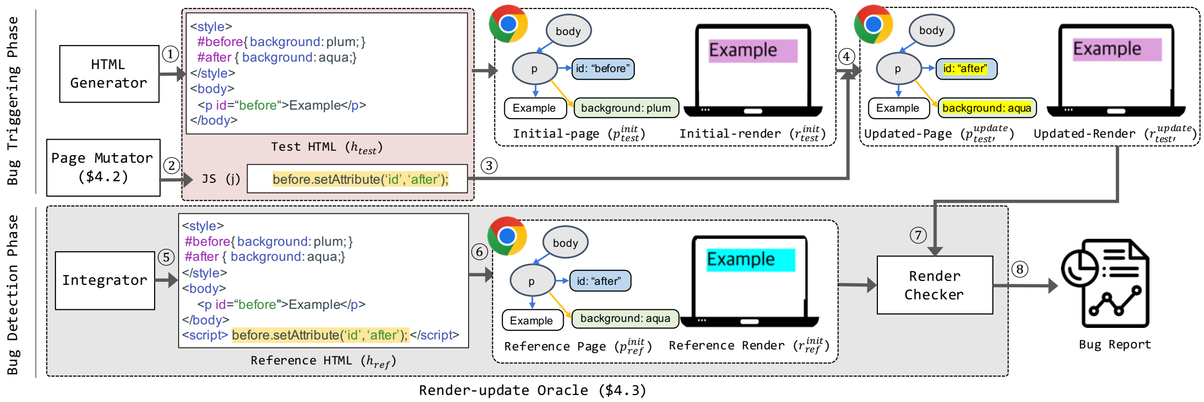


Figure 5: The overall workflow of METAMONG.

Summary: Both cross-browser and cross-version testings trigger many false positives. This is a very critical problem because the browser developers should spend their time and effort on the bugs, which actually do not exist.

3.2 Our Approach

Approach #1: Generating Render-Update Operations Only. To address challenge #1, we build the page mutator, which generates *render-update* operations to change the page and trigger the *render-update*. We do this based on the fact that the *render-update* is the origin of the *render-update* bugs and the browser performs *render-update* when it changes the page and has to re-render the page. In this respect, the page mutator generates the mutation primitives changing the DOM tree or the CSS style of the page. By executing such mutation primitives, METAMONG can find more *render-update* bugs because the browser draws the page based on its DOM tree and CSS style and performs *render-update* whenever the DOM tree and CSS style are changed. We will describe the detail of the page mutator at §4.2.

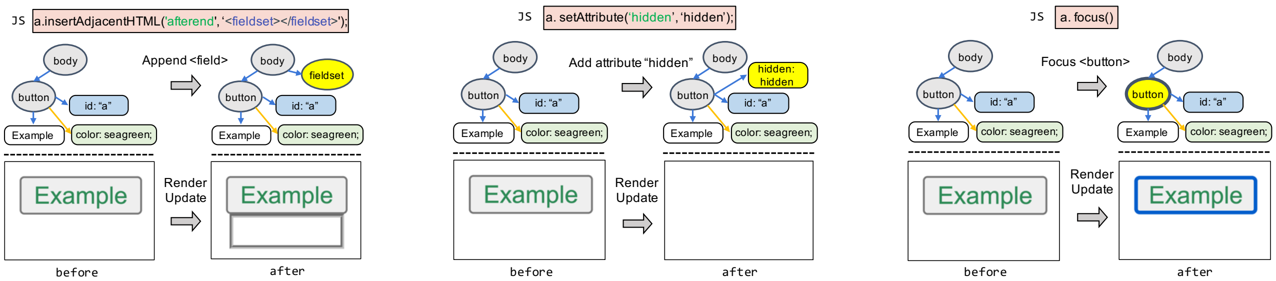
Approach #2: Exploiting an HTML Standard Rule to Build Render-Update Oracle. To address challenge #2, we build the *render-update* bug oracle which can identify the *render-update* bugs without the false positive issues. The key assumption to build the *render-update* oracle is that an initial-render can be used as a reference (or answer) render from the perspective of the *render-update* bug. We can use this assumption because when the browser generates the initial-render, it does not perform *render-update* which is the root cause of *render-update* bugs. To leverage this key assumption, we exploit one of the HTML standard rules called yielding. Yielding is when the browser encounters the javascript while parsing an HTML file, it first blocks the parsing, executes the javascript, and then resumes parsing the HTML file to build the page. By exploiting yielding, we can make the browser build the page that we aim for without performing the *render-update*. In other words, the browser only performs the render-initial on the page so that we can get the reference render of the aimed page. To be specific, METAMONG generates two HTML files where they build the same page but one triggers the *render-update* and the other does not. METAMONG then leverages differential testing to check whether their rendering outputs are the same. If they generate different rendering outputs, the *render-update* bug oracle determines this case as a *render-update* bug. We will describe the detail of the *render-update* oracle at §4.3.

4 DESIGN

We design METAMONG, a framework for finding *render-update* bugs in the modern browsers through differential fuzz testing without the false positive issue. First, we introduce the overall design of METAMONG (§4.1). Then, we introduce the page mutator, which generates the mutation primitives in javascript to trigger the *render-update* bug (§4.2). METAMONG uses the HTML file and the mutation primitives to build the test page and get its rendering result, which is used to determine the *render-update* bug later in §4.3. Lastly, we present the *render-update* oracle that can detect the *render-update* bugs without the false positive (§4.3). The *render-update* oracle consists of two components: 1) integrator, and 2) render checker. The integrator exploits yielding, which is the one of the HTML standard rules. It merges the HTML file and the mutation primitives to construct the reference HTML file, which is used to get the reference rendering result of the test page. The render checker compares the rendering result of test page and its reference rendering result to determine whether the browser triggers the *render-update* bug. If they are different, METAMONG considers it as a *render-update* bug. It is worth noting that the *render-update* oracle can be easily adopted to the other DOM fuzzer to detect the *render-update* bugs.

4.1 Overview

The overall design and workflow of METAMONG are shown in Figure 5. METAMONG consists of two phases: 1) bug triggering phase, and 2) bug detection phase. In the bug triggering phase, METAMONG first generates and opens an HTML file (i.e., h_{test}) on the browser to build the initial-page (i.e., p^{init}_{test}) and produce its corresponding rendering output (i.e., initial-render) denoted as r^{init}_{test} 1. In this example, the initial-page p^{init}_{test} has a `<body>` node as a root and a `<p>` node as a child. The `<p>` node has the `id` attribute and its value is `"before"` so the style of the node `<p>` is calculated as `background: plum`. The



(a) DOM Node Insertion. **(b) DOM Node Attribute Insertion.** **(c) DOM Node Focus.**

Figure 6: The examples of DOM mutation primitive.

browser draws the r^{init}_{test} where the text "`Example`" is drawn inside of the plum rectangle. Then, it uses the mutation API generator to make the mutation primitives, which can trigger the *render-update* ②. Metamong executes the mutation primitives on the page p^{init}_{test} to trigger the *render-update* bug ③. After that, the browser updates p^{init}_{test} to the updated-page (i.e., $p^{update}_{test'}$) and performs the *render-update* to update r^{init}_{test} to the updated-render (i.e., $r^{update}_{test'}$) ④. In this example, p^{init}_{test} becomes $p^{update}_{test'}$–the `id` attribute value of the <p> node changes to "`after`" and its style also changes to `background: aqua`. At the same time, the browser performs the *render-update* and generates $r^{update}_{test'}$ where the text "`Example`" is still drawn within the plum background.

In the bug detection phase, the *render-update* oracle uses the integrator to combine the HTML file h_{test} with the mutation primitives to generate the reference HTML file (i.e., h_{ref}) ⑤. Then, the *render-update* oracle opens h_{ref} on the browser to get its reference render (i.e., r^{init}_{ref}), which is the correct rendering output of $p^{update}_{test'}$ ⑥. In this example, before the browser reaches to the `<script>` tag, the page has the <body> node as a root and the <p> node as a child with the `id` attribute "`before`". When the browser reaches to the `<script>` tag, the browser executes the mutation primitive `target.setAttibute('id', 'after')`. It changes the `id` attribute value from "`before`" to "`after`" so the style of the <p> node is calculated as `background: aqua`. The browser then conducts the render-initial to draw the r^{init}_{ref} where the text "`Example`" is drawn inside of the aqua rectangle. Finally, it leverages the render checker to determine the *render-update* bug by comparing two rendering outputs, $r^{update}_{test'}$ and r^{init}_{ref} ⑦. If $r^{update}_{test'}$ is not same as r^{init}_{ref}, Metamong determines this case as the *render-update* bug ⑧. In this example, the text "`Example`" should be drawn inside of the aqua rectangle (i.e., r^{init}_{ref}) but the browser fails to update the background color to aqua. In this respect, we can determine that the browser triggers the *render-update* bug.

4.2 Page Mutator

It is important that the *render-update* bug can be triggered only when the browser performs the *render-update* on the page. To make the browser perform the *render-update*, the page should be updated (or modified) via javascript. To be specific, the workflow of page

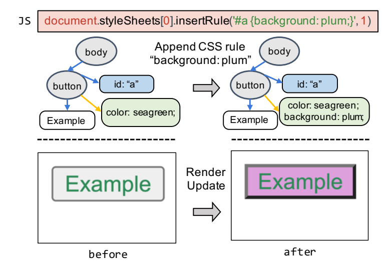


Figure 7: An example of CSS mutation primitive.

mutator is

$$\begin{aligned} &j \leftarrow GenerateMutation() \\ &p^{update}_{test'} \leftarrow PageUpdate(p^{init}_{test}, j) \\ &r^{update}_{test'} \leftarrow RenderUpdate(r^{init}_{test}, p^{update}_{test'}) \end{aligned} \tag{3}$$

where (i) $GenerateMutation$ randomly generates the mutation primitives j, (ii) $PageUpdate$ mutates the initial-page p^{init}_{test} to the updated-page $p^{update}_{test'}$ by executing the mutation primitives j (in javascript) and (iii) $RenderUpdate$ updates the rendering output r^{init}_{test} to the $r^{update}_{test'}$, which is the corresponding rendering output of $p^{update}_{test'}$. As the browser draws the page based on its DOM tree and CSS style, we employ two page-mutation methods: 1) DOM tree mutation; and 2) CSS style mutation.

DOM Mutation Primitive. Metamong currently has three DOM mutation operations: (1) DOM node insertion/deletion; (2) DOM node attribute insertion/deletion; (3) DOM event; For DOM node insertion, Metamong randomly selects where to insert and then generates the DOM node. Then, it uses `insertAdjacentElement (position, element)` DOM API to insert the node in the selected position. Similarly, for DOM node deletion, Metamong randomly selects and deletes the one of the DOM nodes via `remove()` DOM API. The example of DOM node insertion is described in Figure 6a. Before mutation, the page has the <body> node as a root and the child node (i.e., the `<button>` node). The `<button>` node has the `id` attribute with its value "`a`", the text node `Example`, and the CSS style `color: seagreen`. Here, Metamong selects the <body> node, generates the `<fieldset>` node, and inserts it inside the <body> node after its last child node (i.e., `<button>` node). After mutation, the browser performs *render-update* to draw the `<fieldset>` node below the text `Example`.

For DOM node attribute insertion, METAMONG randomly selects the one of DOM nodes and inserts the attribute with the value via `setAttribute(name, value)`. Similarly, for DOM node attribute deletion, METAMONG randomly selects the one of DOM nodes and deletes the one of selected node's attribute names via `removeAttribute(name)`. The example of DOM node attribute insertion is described in Figure 6b. METAMONG selects the `<button>` node and inserts the `hidden` attribute with the value "`hidden`". After mutation, the browser performs *render-update* to erase the `<button>` node and its text node as well.

METAMONG currently has three DOM event operations: (1) DOM node focus, (2) DOM node scrolling, and (3) window resizing. For DOM node focus, METAMONG randomly selects and focuses one of the DOM nodes via `focus()` DOM API. For DOM node scrolling, METAMONG randomly selects and moves the scroll via `scrollTo(x, y)` DOM API. For window resizing, METAMONG randomly changes the size of the browser window via `resizeTo(width, height)` DOM API. An example of DOM node focus is shown in Figure 6c. METAMONG selects and focuses the `<button>` node so the browser performs *render-update* to thicken the outer edge of the `<button>` node.

CSS Mutation Primitive. METAMONG currently has one CSS mutation operation: CSS style insertion/deletion; For CSS style insertion, METAMONG randomly generates the CSS rule and selects the index into which the CSS style rule is to be inserted via `insertRule(rule, index)` API. Similarly, for CSS style deletion, METAMONG randomly selects and deletes one of the CSS style rules via `deleteRule(index)` API. An example of CSS style insertion is described in Figure 7. METAMONG selects one as the index and inserts the CSS style rule "`p {background: plum;}`". After mutation, the browser performs *render-update* to paint the background of the text "`Example`".

4.3 Render-Update Oracle

Overview. The *render-update* bug is a semantic bug, which is triggered if the browser incorrectly performs the *render-update* when changing the page through the javascript. Due to the incorrect *render-update*, it generates the incorrect rendering output. In order to detect the *render-update* bug, we build the *render-update* oracle based on the following assumption:

> **Assumption: Initial-render is the reference (or answer) rendering result of the page.** The updated-render should be the same as the reference render (i.e., initial-render) if their pages are the same.

This assumption is based on the following two properties of the browser rendering process: i) each page has exactly one rendering output, and ii) the browser only performs the render-initial phase, not the *render-update* when generating the initial-render. Based on these properties, if the initial-render and the updated-render are generated from the same page, they should be the same. If they are different, this implies that at least one of them is incorrect. In this paper, the initial-render is always correct based on our assumption because we focus on finding the *render-update* bug, which can be only triggered in render-update phase, not in render-initial phase.

To this end, the *render-update* oracle determines a *render-update* bug when the updated-render and the initial-render are different.

To be specific, the *render-update* oracle is

$$\begin{aligned}
&h_{ref} \leftarrow Integration(h_{test}, j) \\
&p_{ref}^{init} \leftarrow BuildPage(h_{ref}) \\
&r_{ref}^{init} \leftarrow InitRender(p_{ref}^{init}) \\
&result \leftarrow RenderCheck(r_{ref}^{init}, r_{test'}^{update})
\end{aligned} \tag{4}$$

where (i) *Integration* merges the HTML file h_{test} with the mutation primitives j to generate the reference HTML file h_{ref}; (ii) *BuildPage* builds the page with h_{ref}; (iii) *InitRender* renders the page p_{ref}^{init} to draw the reference (or answer) render r_{ref}^{init}; and (iv) *RenderCheck* checks whether $r_{test'}^{update}$ is equal as r_{ref}^{init} and determines the *render-update* bug if they are different.

Integrator. The integrator generates the reference HTML file, which is used to get the reference rendering result of the updated-page $p_{test'}^{update}$. To do so, it exploits **yielding**, one of HTML rules defined in HTML standard [10]:

> **Yielding:** When the browser encounters the javascript while parsing (or loading) an HTML file, it blocks the parsing, executes the javascript, and then resumes to parse the HTML file.

By using this property, we can get an initial-render (i.e., reference render) of the page, which is the same as $p_{test'}^{update}$. That is, the integrator can make the reference HTML file that can be used to build the same page as $p_{test'}^{update}$ without the *render-update*. To be specific, if we execute the javascript right before the browser finishes building the page, we can make the browser block the rendering process and change the page. After the page changes, the browser performs the render-initial so that we can get the reference render of the page without triggering *render-update*.

To do so, the integrator wraps the primitives with `<script>` tag and then appends it to the test HTML file to generate the reference HTML file. By doing so, when the browser loads the reference HTML file, the browser loads the HTML file and executes the mutation primitives to update the page. Then, it finishes building the page and performs the render-initial to draw the reference render of the page.

Render Checker. After the integrator produces the reference render r_{ref}^{init}, the *render-update* oracle leverages the render checker to determine *render-update* bug. The render checker compares two rendering outputs, $r_{test'}^{update}$ and r_{ref}^{init}, and determines as *render-update* bug if they are different. To compare the rendering outputs, we adopt the same image comparison algorithm (i.e., phash [13]) and configuration used by R2Z2 [49] because the *render-update* bugs are very similar to the rendering bugs hunted by R2Z2. It is worth noting that other image comparison algorithms also can be used to detect *render-update* bugs. Finally, if the render checker determines there is a *render-update* bug, it generates the bug report.

5 IMPLEMENTATION

We implemented METAMONG on top of R2Z2 [49] to detect *render-update* bugs from modern browsers. The prototype of METAMONG

Table 1: The number of *render-update* bugs detected by each oracle.

Browser	# of Bugs	R2Z2	LQC	Render-update Oracle
Chrome	15	0 (0%)	3 (20.0%)	15 (100%)
Firefox	13	0 (0%)	9 (69.2%)	13 (100%)
Total	28	0 (0%)	12 (42.9%)	28 (100%)

can fuzz Chrome and Firefox browsers. For the HTML generator, we leveraged `Domato` fuzzer [7], a state-of-the-art grammar-based DOM fuzzer. We modified `Domato` fuzzer to generate the HTML file (i.e., an initial test page) without animations and Javascript. In order to implement the page mutator, we selected DOM/CSS APIs changing the DOM tree and CSS style from the grammar of `Domato` fuzzer. We modified R2Z2's fuzzing implementation to implement the integrator of *render-update* oracle. For the render checker, we adapt R2Z2's image comparison implementation and used the same image comparison algorithm and configuration of R2Z2. In terms of the implementation complexity, Metamong is implemented with 2300 lines of Python.

6 EVALUATION

We evaluate Metamong by answering the following research questions:

- **RQ 1.** Can Metamong detect the *render-update* bugs?
- **RQ 2.** Which mutation primitive is effective to find *render-update* bugs?
- **RQ 3.** How many bugs has Metamong found?

For evaluation, we used a 24-core server running Ubuntu 20.04, with Intel Xeon(R) Gold 5118 (2.30GHz) CPUs and 512GB of RAM.

6.1 Effectiveness of Render-Update Oracle

6.1.1 Recall of Render-Update Oracle. In order to show the effectiveness of *render-update* oracle in detecting bugs, we compared the *render-update* oracle against two different oracles, R2Z2 [49] and LQC [12]. To be specific, we first selected the popular web browsers, Chrome and Firefox. Then, we collected the *render-update* bugs which were already reported in Chrome and Firefox bug trackers. From the Chrome bug tracker, we searched the *render-update* bugs that were reported from 2021 to 2022 and obtained 15 *render-update* bugs [14]. From the Firefox bug tracker, we searched the *render-update* bugs that were reported from 2020 to 2022 and obtained 13 *render-update* bugs [15]. After we collected the *render-update* bugs, we simplified bugs for evaluation and ran three oracles to check whether they can detect these *render-update* bugs. Then, we setup the experiment environment for each oracle. To run R2Z2, we provided the similar experiment setup as described in its paper. We first selected a beta version of each bug as **B** and a stable version released at the time of **B** as **A**. Then, we selected the independently-developed browser as the reference browser (i.e., **R**), where the version of **R** is the closest of **B**. To run LQC, we used the same experiment setup as Metamong.

Table 1 describes the number of *render-update* bugs detected by three oracles. LQC identified three *render-update* bugs in Chrome and nine in Firefox (i.e., the recall of LQC is 42.9%). However, it was not able to detect 12 Chrome bugs and four Firefox bugs that were triggered by DOM events (e.g., scroll) because LQC only allows to mutate the DOM tree and CSS style to prevent false positives. On the other hand, R2Z2 failed to detect any of the 28 render-update bugs, resulting in a recall rate of 0.0%. This failure is largely due to the following two reasons. Firstly, R2Z2 is specifically tailored to detect "regression" bugs, and render-update bugs did not fall under this category. Secondly, R2Z2 can only identify *render-update* bugs when the browser interoperability conditions are met (i.e., Chrome and Firefox generate the same rendering result for a given input). In the case of Chrome *render-update* bugs, only two out of 15 were regression bugs, and R2Z2 successfully detected the differences between versions **A** and **B** for two bugs. However, R2Z2 failed to detect two Chrome *render-update* bugs due to the unique design variations between Chrome and Firefox, which made R2Z2 fail to meet the browser interoperability conditions. Furthermore, R2Z2 failed to detect any of the Firefox *render-update* bugs because there were no regression bugs. Lastly, the *render-update* oracle was able to detect all of 28 *render-update* bugs (i.e., the recall of *render-update* oracle is 100%). This demonstrates Metamong's strong practical capability in detecting render-update bugs.

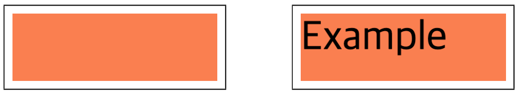
Case Study: Chrome Issue #1222734. This bug is interesting because Metamong was able to detect the bug and did not report the false positive even though the reference render is incorrect according to DOM/CSS specifications. In other words, even if r_{ref}^{init} is incorrect from the perspective of DOM/CSS specifications, it is correct from the perspective of the *render-update* bug. This is because the initial-render can be generated only by render-initial phase and the *render-update* bug can be caused only by the incorrect behavior of render-update phase (i.e., the origin of *render-update* bug). In this respect, the initial-render can be used as the reference rendering result (i.e., r_{ref}^{init}) to determine the *render-update* bug, making Metamong avoid the false positive. Besides, Metamong was able to detect this bug because $r_{test'}^{update}$ is different to r_{ref}^{init}. This case study signifies that even if the reference result is incorrect according to DOM/CSS specifications, Metamong does not generate the false positive and is able to detect the bug if *render-update* triggers a *render-update* bug.

6.1.2 Accuracy of Render-Update Oracle. Ideally, the *render-update* oracle should not trigger the false positive issue. This is because the *render-update* oracle compares two rendering outputs that are generated from two same pages where one is updated and the other is not. As each page has exactly one rendering output, there must be a *render-update* bug if two rendering outputs are different. Our claim can be supported by our evaluation testing Metamong because all of the 21 *render-update* bugs reported by Metamong were confirmed by the developers of each browser. That is, we could not find any false positive case while evaluating Metamong.

The limitation of LQC is it cannot detect *render-update* bugs which can be triggered by DOM events because it employs only DOM node and CSS style mutation primitives to prevent the false positive. In this respect, if LQC is employed to detect *render-update* bugs triggered by DOM events, it will suffer from the false positive issues. For example, Figure 8 illustrates a false positive case of LQC that can be caused by the DOM event (i.e., `scroll`). In this example, Chrome draws the correct rendered result (as shown in Figure 8b) after executing the function `update_page()` which moves the text

```
<!DOCTYPE html>
<script>
  function update_page() { target.scrollTo(0, 50); }
</script>
<body><div id="target" style="height: 100px; overflow: auto;">
  <div style="height: 800px; background: coral">Example</div>
</div></body>
```

(a) PoC HTML code



(b) Actual Result (Correct) (c) Reference Result of LQC (Incorrect)

Figure 8: A false positive example of LQC with `scroll` mutation primitive.

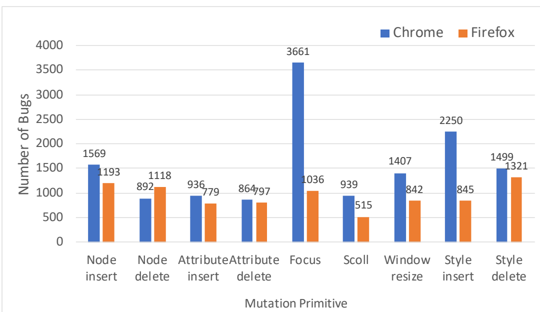


Figure 9: The average number of bugs triggered by each mutation primitive on 100K HTML testcases across three experiments.

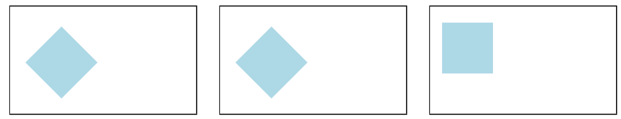
`False Positive` 10 pixels upwards. The problem here is that LQC does not consider the DOM event when generating the reference result. As a result, LQC generates the incorrect reference result (shown in Figure 8c) and triggers the false positive as both results (i.e., Figure 8 and Figure 8c) are different.

6.2 Effectiveness of Page Mutator

The effectiveness of mutation primitives are important as they are the key to trigger *render-update* bugs. To evaluate their impact, we conducted three experiments measuring the number of bugs triggered by each mutation primitive using the same set of 100,000 HTML test cases on both Chrome and Firefox. Figure 9 describes the average number of *render-update* bugs for each mutation primitive on Chrome and Firefox across three experiments. According to this evaluation, `Focus` and `Style delete` are the best mutation primitives triggering *render-update* bugs in Chrome and Firefox, respectively. `Node insert` and `Style delete` are effective mutation primitives on both browsers as they can change many parts of the page. On the other hand, `Attribute delete` and `scroll` are the worst mutation primitives triggering *render-update* bugs in Chrome and Firefox, respectively. In this experiment, we were unable to identify patterns of effective mutation primitives because the number of *render-update* bugs for varies depending on the browser. We leave this as a future work.

```
<!DOCTYPE html>
<script>
  function update_page() { document.styleSheets[0].deleteRule(0); }
</script>
<style>
  #update { offset-path: path('M 0 1 L -1 0'); }
  #update { transform: translatez(0); }
  #child {width: 100px; height: 100px; background: lightblue;}
</style>
<body><dl id="update"><div id="child"></div></dl></body>
```

(a) PoC HTML code.



(b) Before (c) After (Actual) (d) After (Expected)

Figure 10: A case study of Chrome Issue #1365255.

6.3 New Render-Update Bugs Discovered by METAMONG

To discover *render-update* bugs, we ran METAMONG on Chrome 89 and Firefox 85, which were the latest browser versions at the time of performing our experiment. Then we conducted an additional experiment on the latest version of Chrome (i.e., Chrome 108) in order to uncover more *render-update* bugs. During the first experiment, METAMONG found five and four *render-update* bugs in Chrome 89 and Firefox 85, respectively, and total of nine *render-update* bugs were confirmed by each browser vendor. Furthermore, it was confirmed that five of the Chrome and two out of four Firefox bugs were new *render-update* bugs. In additional experiment, METAMONG found 12 *render-update* bugs in Chrome 108, and the Chrome developers confirmed that total of 12 *render-update* bugs were true and new *render-update* bugs.

To summarize, METAMONG has found 17 and four *render-update* bugs (21 in total) and 17 and 2 of them (19 in total) were new *render-update* bugs in Chrome and Firefox, respectively. The list of 21 *render-update* bugs found by METAMONG is shown in Table 2. All of 21 *render-update* bugs were confirmed by the developers of each browser vendor, 19 of them were new *render-update* bugs, and five of them were fixed so far, showing the efffectiveness of METAMONG for detecting *render-update* bugs. In addition, the Chrome developer has shown a positive reaction, mentioning that the design of METAMONG is interesting [9]. It is difficult for us to clearly measure how long it takes to find a certain vulnerability, as we kept running METAMONG and METAMONG's behavior (particularly its mutation) is mostly random. Overall we ran METAMONG for two months, which includes the time to detect and analyze the bug as well as updating METAMONG.

Case Study: Chrome Issue 1365255. This *render-update* bug occurs when Chrome performs *render-update* on the node that has the child node and the `transform` CSS style. Chrome should update the node and its child node when the CSS style of the node is changed. However, due to the performance optimization on `transform` CSS style, Chrome only updates the node and does not update the child node, triggering the *render-update* bug. The snippet of a PoC code is shown in Figure 10a. First, Chrome opens the PoC HTML code and draws the rendering output (Figure 10b).

Table 2: The list of 21 *render-update* bugs found by METAMONG in Chrome and Firefox.

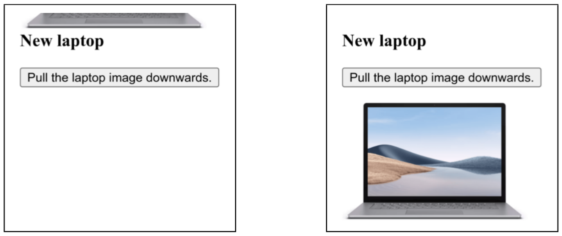
Browser	Issue ID	Correct	Incorrect	New	Fixed	Description
Chrome (89.0.4329.0)	#1154662			✓		The dashed underline is incorrect after removing an element.
Chrome (89.0.4329.0)	#1162740			✓		The text in <dl> moves to the wrong position after removing a CSS rule "content".
Chrome (89.0.4329.0)	#1163006			✓	✓	The <details> moves to a wrong position after removing CSS rules "column" and "position".
Chrome (89.0.4329.0)	#1163031			✓		The height of <dd> is larger than expected after removing a CSS rule "height".
Chrome (89.0.4329.0)	#1164643			✓	✓	The position of <h4> is incorrect after adding a CSS rule "position"
Chrome (108.0.5305.0)	#1364376			✓		The width of <keygen> does not change after removing a CSS rule "border-style".
Chrome (108.0.5305.0)	#1365243			✓	✓	The text is larger than expected after removing a CSS rule "scale".
Chrome (108.0.5305.0)	#1365244			✓		The position of <dialog> is incorrect after removing a CSS rule "backdrop-filter".
Chrome (108.0.5305.0)	#1365252			✓		The size of <th> is incorrect after removing a CSS rule "writing-mode".
Chrome (108.0.5305.0)	#1365255			✓	✓	The border line of <fieldset> is not updated after removing a CSS rule "offset-path".
Chrome (108.0.5305.0)	#1365746			✓		The height of <fieldset> does not decrease after removing a CSS rule "margin-right".
Chrome (108.0.5305.0)	#1366233			✓	✓	The shape of <q> is incorrect after removing a CSS rule "font-weight".
Chrome (108.0.5305.0)	#1366280			✓		The height of <th> is incorrect after removing a CSS rule "margin-left".
Chrome (108.0.5305.0)	#1370936			✓		The border line is incorrect after removing a CSS rule "-webkit-border-end".
Chrome (108.0.5305.0)	#1370962			✓		The size of <table> is incorrect after removing a CSS rule "@font-face".
Chrome (108.0.5305.0)	#1370987			✓		The position of quote is incorrect after removing an element.
Chrome (108.0.5305.0)	#1371003			✓		The location of text is incorrect after adding an element.
Firefox (85.0a1)	#1680232			✓		The line moves to a wrong position after adding a CSS rule "display".
Firefox (85.0a1)	#1683814			✓		The size of <dir> is bigger unexpectly after adding an element.
Firefox (85.0a1)	#1683820					The position of <dialog> is incorrect after adding an element.
Firefox (85.0a1)	#1684290					The position of <label> is incorrect after removing a CSS rule "input".

Then, it executes the function `update_page()` to delete the CSS rule `#update {offset-path: path('M 0 1 L -1 0');}`. The correct behavior is that Chrome removes the CSS rule from the <dl> node and its child node <div> and performs *render-update* to update the rendering output. The correct rendering output is shown in Figure 10d. However, as Chrome does not update the child node <div>, it generates the incorrect rendering output (Figure 10c) introduced by the *render-update* bug.

Figure 11 illustrates a potential negative consequence of this *render-update* bug. Consider a scenario where a website is selling a new laptop, and users can view the laptop image by clicking a button. The intended correct behavior is once the button is clicked, the CSS `offset-path` is removed and the laptop image should be placed below the button (as depicted in Figure 11b). Unfortunately, as illustrated in Figure 11a, Chrome fails to move the laptop image due to the *render-update* bug, disturbing users from viewing the laptop image.

7 RELATED WORK

Browser Layout Testing. Previous research has suggested testing methods to aid web application developers in finding cross-browser incompatibilities in their web applications [29, 30, 43]. They discovered cross-browser incompatibilities in web applications by checking whether two independently-implemented browsers produce different rendering outputs from the same page. They determine the bug if two browsers produce different rendering outputs. Note that, unlike the previous research, METAMONG focuses on detecting *render-update* bugs in web browsers, not in web applications. In addition, METAMONG does not trigger any false positive



(a) Actual Result (Incorrect). **(b) Expected Result (Correct).**

Figure 11: A possible negative consequence of the render-update bug by Chrome Issue #1365255. The laptop image should be placed below the button once clicked, but the image does not move due to the bug.

issue, but the previous research suffers from the false positive issue due to benign cross-browser incompatibilities. Another previous research has used the manually-implemented oracle with image comparison technique to detect HTML presentation failures in web applications [40, 41]. Note that METAMONG does not require manual effort and domain knowledge to build the oracle which is the key contribution of our paper. There is another work, R2Z2 [49], which detects regression rendering bugs from web browsers. It combines cross-browser testing, cross-version testing, and WPT tests to avoid false positive issues. Nonetheless, it still suffers from the false positive issue due to the missing WPT tests. LQC [12] detects *render-update* bugs in web browsers using differential testing. It first updates the page, reloads the updated page, and compares pages before and after reloading. It determines the bug if pages before and after reloading are different. However, this approach cannot detect *render-update* bugs that DOM events can trigger due to the characteristic of page reloading. Note that METAMONG can detect *render-update* bugs triggered by DOM events, meaning better practical ability in bug detection.

Browser Layout Verification. Several works partially formalized the browser layout algorithm [44–46]. They used static analysis techniques to verify the overall implementation of the browser's rendering component. To do so, they translated the HTML and CSS specifications into the formalized rules for the static analysis. However, this approach has the critical limiation–writing the formalized rules not only requires domain knowledge but also is very labor-intensive and error-prone. In this respect, this approach suffers from the false positive issue when the formalized rules are incorrect or insufficient.

Semantic Bug Fuzzing. DiFuzzRTL [35] proposes a new coverage metric to capture the states of an RTL design and detect CPU bugs. R2Z2 [49] combines the cross-browser and cross-version testings to detect the regression rendering bugs from the browser. Several works find semantic bugs in Java Virtual Machine (JVM) implementations via differential testing [24, 27, 28]. Some studies leveraged the fuzzing to find the semantic bugs from the deep learning libraries [32, 51]. [32] automatically infers the relational APIs to find the inconsistencies from the deep learning libraries. [51] leverages the open source to infer the API input parameter types for effective deep learning library fuzzing. PGFuzz [37] proposes a policy-guided fuzzer to detect policy violations from robotic vehicle control programs. FuzzOrigin [38] proposes a static origin tagging mechanism to detect UXSS vulnerabilities in browsers.

8 DISCUSSION

Sufficient Number of Mutation Primitives. METAMONG currently has three DOM and one CSS mutation primitives. Compared to other DOM fuzzers such as `Domato` [7] and `FreeDOM` [53], METAMONG leverages a few number of DOM and CSS APIs. One might think using a small number of APIs extremely limits the bug finding. However, while evaluating the recall of *render-update* oracle (§6.1.1), we observed that all 28 *render-update* bugs can be triggered by METAMONG's mutation primitives. This shows that the current implementation of METAMONG does not limit the *render-update* bug finding.

Lack of Guiding Methods. METAMONG employs `Domato` fuzzer to generate HTML files. `Domato` is a grammar-based generation fuzzer and it does not use guidance methods such as coverage-guiding for testcase generation. Thus, it is possible that the *render-update* bug is hidden deep in the browser's rendering implementation and `Domato` fuzzer cannot generate the HTML file triggering the *render-update* bug. However, the unsuitable guiding method can further disturb finding bugs. For instance, `FreeDOM` guided by coverage triggers 3.8X fewer crashes in DOM fuzzing compared to its generation fuzzing. Nonetheless, we think leveraging a suitable guiding approach would definitely enhance METAMONG's bug-finding capability. We leave this as future work.

9 CONCLUSION

This paper proposed METAMONG, a framework tailored for detecting *render-update* bugs in web browsers without false positives. METAMONG consists of two key components: a page mutator, and a *render-update* oracle. The page mutator generates the mutation primitives changing the DOM tree and CSS styles, to trigger *render-update* bugs. The *render-update* oracle exploits an HTML standard rule, yielding, to detect *render-update* bugs without false positives. With the prototype implementation of METAMONG, it discovered 19 new *render-update* bugs in Chrome and Firefox browsers without false positives, demonstrating its effectiveness and practical ability in finding *render-update* bugs.

10 DATA AVAILABILITY

We disclosed the source code of METAMONG and as well as the data used in this paper at https://figshare.com/s/d3c228e614672f9aa811.

ACKNOWLEDGMENTS

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 the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2023-00209093). 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] has() - css: Cascading style sheets - mdn web docs. https://developer.mozilla.org/en-US/docs/Web/CSS/:has.

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

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

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

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

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

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

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

[9] Issue 1364376: Empty <span> still has height after removing border. https://bugs.chromium.org/p/chromium/issues/detail?id=1364376#c4.

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

[11] Life of a pixel. https://docs.google.com/presentation/d/1boPxbgNrTU0ddsc144rcXayGA_WF53k96imRH8Mp34Y/edit.

[12] Browser layout testing - quickcheck. https://github.com/nathand8/layout-quickcheck.

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

[14] The list of render-update bugs in chrome bug tracker, . https://bugs.chromium.org/p/chromium/issues/list?q=opened%3E2021-1-1%20opened%3C2023-2-28%20component%3ABlink%3EPaint%3EInvalidation%20-status%3DWontFix%20-status%3DDuplicate%20-Type%3DTask%20-Type%3DCompat%20-label%3APerformance-sheriff%20-label%3AClusterFuzz%20-label%3AFindit&can=1/.

[15] The list of render-update bugs in firefox bug tracker, . https://bugzilla.mozilla.org/buglist.cgi?short_desc_type=allwordssubstr&short_desc=%7Binc%7D&classification=Client%20Software&classification=Developer%20Infrastructure&classification=Components&classification=Server%20Software&classification=Other&query_format=advanced&bug_status=RESOLVED&bug_status=NEW.

[16] Inside look at modern web browser (part 3), . https://developer.chrome.com/blog/inside-browser-part3/.

[17] Rendering overview, . https://firefox-source-docs.mozilla.org/gfx/RenderingOverview.html/.

[18] Issue 1214206: Wide-cell percentage widths are distributed incorrectly. https://crbug.com/1214206.

[19] Issue 1132218: tesla.com: Color options not rendered until window resize when compositesvg is enabled. https://bugs.chromium.org/p/chromium/issues/detail?id=1132218.

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

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

[22] 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.

[23] M. Böhme, L. Szekeres, and J. Metzman. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, Pittsburgh, PA, May–May 2022.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] S. R. Choudhary, M. R. Prasad, and A. Orso. X-pert: Accurate identification of cross-browser issues in web applications. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, May 2013.

[32] Y. Deng, C. Yang, A. Wei, and L. Zhang. Fuzzing deep-learning libraries via automated relational api inference. In *Proceedings of the 30th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Singapore, Nov. 2022.

[33] G. A. Di Luna, D. Italiano, L. Massarelli, S. Österlund, C. Giuffrida, and L. Querzoni. Who's debugging the debuggers? exposing debug information bugs in optimized binaries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Virtual, Apr. 2021.

[34] 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.

[35] 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.

[36] 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.

[37] H. Kim, M. O. Ozmen, A. Bianchi, Z. B. Celik, and D. Xu. Pgfuzz: Policy-guided fuzzing for robotic vehicles. In *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*, Virtual, Feb. 2021.

[38] S. Kim, Y. M. Kim, J. Hur, S. Song, G. Lee, and B. Lee. FuzzOrigin: Detecting UXSS vulnerabilities in browsers through origin fuzzing. In *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, Aug. 2022.

[39] D. Lehmann and M. Pradel. Feedback-directed differential testing of interactive debuggers. 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.

[40] 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.

[41] 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.

[42] R. Meng, Z. Dong, J. Li, I. Beschastnikh, and A. Roychoudhury. Linear-time temporal logic guided greybox fuzzing. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, Pittsburgh, PA, May–May 2022.

[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] 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.

[46] 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.

[47] 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.

[48] 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.

[49] S. Song, J. Hur, S. Kim, P. Rogers, and B. Lee. R2z2: Detecting rendering regressions in web browsers through differential fuzz testing. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, Pittsburgh, PA, May–May 2022.

[50] 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.

[51] A. Wei, Y. Deng, C. Yang, and L. Zhang. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, Pittsburgh, PA, May–May 2022.

[52] M. Wu, L. Jiang, J. Xiang, Y. Huang, H. Cui, L. Zhang, and Y. Zhang. One fuzzing strategy to rule them all. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, Pittsburgh, PA, May–May 2022.

[53] 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.

[54] Y. Yang, Y. Zhou, H. Sun, Z. Su, Z. Zuo, L. Xu, and B. Xu. Hunting for bugs in code coverage tools via randomized differential testing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, Montreal, QC, Canada, May 2019.