

Testing IFrame Rendering in Web Browsers

Dahyeon Park

UNIST

Ulsan, South Korea

tryness@unist.ac.kr

Yuseok Jeon

Korea University

Seoul, South Korea

ys_jeon@korea.ac.kr

Seongil Wi

UNIST

Ulsan, South Korea

seongil.wi@unist.ac.kr

Mijung Kim[✉]

UNIST

Ulsan, South Korea

mijungk@unist.ac.kr

Abstract

Iframes are widely used in modern web development, but their separation from the parent page can lead to iframe-specific rendering bugs, which occur only when content is embedded within an iframe. These bugs may degrade user experience and, in some cases, lead to serious security vulnerabilities. However, existing rendering bug detection techniques are largely ineffective since their oracles compare rendering outputs only within the same context and fail to capture discrepancies across different rendering contexts.

This paper presents ICHECK, an automated testing technique for iframe rendering in web browsers. Specifically, given an HTML document, ICHECK introduces a cross-context oracle that detects inconsistencies by comparing the same HTML content in both top-level and iframe contexts. To further assist debugging, ICHECK performs layout-aware input reduction by redefining the effective search space of bug-inducing elements. As a result, ICHECK helps developers more effectively and efficiently identify and diagnose iframe-specific rendering bugs. Evaluated on the most popular browsers, Chrome and Firefox, ICHECK detected 21 previously unknown bugs, all of which were confirmed. On average, ICHECK uncovered 3.8× more bugs than state-of-the-art rendering bug detection work. In addition, ICHECK effectively reduced HTML inputs by 93% in DOM nodes while achieving up to 10.1× faster reduction time compared with the state-of-the-art delta debugging approaches.

CCS Concepts

• Software and its engineering → Software testing and debugging.

Keywords

Iframe Rendering, Rendering Bug Oracles, Rendering Bug Detection, Rendering Bug Debugging, Browser Rendering Bugs, Web Browsers

ACM Reference Format:

Dahyeon Park, Yuseok Jeon, Seongil Wi, and Mijung Kim. 2026. Testing IFrame Rendering in Web Browsers. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26)*, October 12–16, 2026, Munich, Germany. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3832783.3834417>

1 Introduction

<iframe> is an HTML element used to embed another HTML document into the parent page [38]. An iframe in an HTML page creates

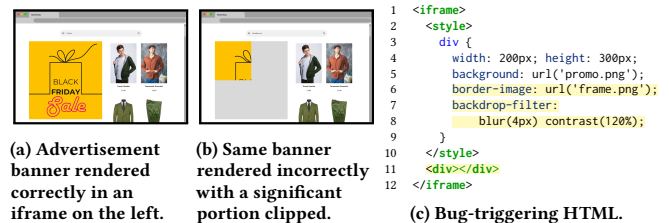


Figure 1: Example of an iframe rendering bug

a new document context separate from the parent page and results in an isolated rendering context. For this reason, iframes are widely used to integrate third-party resources, including advertisement modules, social media widgets, and video players like YouTube, into the host page. Additionally, complex web services often use iframes to modularize functionality, allowing each feature to be developed and maintained independently. Reflecting this widespread usage, iframes appear on approximately 40% of the top one million websites [3].

While the isolation feature of iframes provides significant benefits, it also introduces challenges that can lead to rendering bugs. In particular, the content of an iframe should be rendered independently but visually embedded within the parent page. This dual requirement—logical separation with visual coherence—complicates the implementation of browser rendering engines and has led to many rendering bugs in popular browsers like Chrome. These iframe rendering bugs typically manifest as visual inconsistencies such as clipped, misaligned, or escaping content in an iframe. Figure 1 illustrates an example of clipped content caused by an iframe rendering bug. An iframe bug affects the visibility of third-party content, as shown by the yellow advertisement banner on the left in the figure. For another example, when a payment interface embedded in an iframe is not properly rendered, users may be unable to complete transactions properly, which might cause business loss. All these iframe rendering bugs not only degrade the user experience, but can also lead to severe consequences such as business loss and security risks (security is discussed in more detail in §2).

Despite the prevalence of iframes in modern web development, prior work on rendering bug detection [34, 35, 50] has overlooked iframe-specific bugs. Existing techniques primarily focus on other types of rendering bugs, such as rendering regression bugs [34], JavaScript-specific render-update bugs [35], and general rendering bugs that do not involve iframes [50]. These techniques leverage differential testing by defining their own differential oracles for their target bug types. They compare visual outputs rendered in the same



This work is licensed under a Creative Commons Attribution 4.0 International License. ASE '26, Munich, Germany

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2882-2/2026/10

<https://doi.org/10.1145/3832783.3834417>

rendering context (*i.e.*, top-level¹ vs. top-level). However, those oracles are ineffective for detecting iframe rendering bugs because iframes are rendered in a different rendering context than top-level documents. As a result, these oracles fail to capture inconsistencies that arise between top-level and iframe contexts, where the same content can be rendered differently due to iframe-specific behaviors. This limitation points to the need for a new differential oracle that is specifically designed for iframe scenarios by comparing how the same iframe content is rendered across different contexts.

Once differential testing identifies an inconsistent rendering of a given HTML document, debugging the root cause of the inconsistency remains a significant challenge [13, 31]. One possible debugging aid would be delta debugging that supports HTML input format and isolates failure-inducing inputs through iterative input reductions [25, 36]. However, applying delta debugging for rendering bugs is costly because browser rendering operations are expensive (as discussed in §3), and each reduction step requires repeated rendering, leading to high computational overhead. As a result, the debugging process becomes inefficient. Moreover, existing delta debugging techniques rely on predefined search strategies [40, 48, 52] that optimize how the input space is traversed during reduction. Despite these optimizations, they still require exploring the entire input space. When each evaluation involves expensive browser rendering, this exhaustive exploration further amplifies the overhead. These limitations motivate the need for a new HTML input reduction technique tailored to browser rendering bugs—one that redefines the effective search space by focusing only on HTML elements relevant to the observed visual inconsistency.

In this paper, we present ICHECK, an automated testing technique for detecting and debugging iframe-specific rendering bugs in web browsers. Our key insight is that existing techniques [34, 35, 50] are fundamentally limited by comparing rendering outputs only within the same rendering context, thereby missing inconsistencies that arise across contexts. To address this gap, we introduce a *cross-context differential oracle* that systematically compares the rendering of identical HTML content in top-level and iframe contexts, enabling the detection of a previously overlooked class of rendering bugs.

Beyond detection, we propose a layout-aware input reduction technique that differs from traditional delta debugging approaches that support HTML input format [25, 36]. Prior work focuses on optimizing how to traverse the input space [40, 48, 52] using syntactic [36] or structural [25] information. In contrast, our approach *redefines what needs to be explored by leveraging rendering semantics*. Specifically, we bridge the gap between visual output and HTML structure by mapping regions of visual inconsistency in rendered images to corresponding DOM elements, and use this information to prune semantically irrelevant parts of the input. This cross-domain reduction strategy—combining visual layout analysis with DOM structure—enables a significant reduction of the search space while preserving bug-inducing behaviors.

Together, these contributions form the first end-to-end framework for iframe rendering bug detection and debugging, demonstrating that incorporating rendering semantics into both oracle

design and input reduction fundamentally improves the effectiveness and efficiency of browser testing.

We evaluated ICHECK on the two most popular web browsers, Chrome and Firefox. Our results show that ICHECK is significantly more effective in detecting iframe rendering bugs than state-of-the-art techniques (R2Z2 [34], Metamong [35], and Janus [50]) by detecting 3.8 times more bugs on average. ICHECK also detected 37 bugs (21 previously unknown and 16 known), including one newly discovered vulnerability that was assigned a CVE. All unknown bugs have been confirmed by developers, and four have been fixed. Additionally, our layout-aware HTML input reduction approach effectively reduces the number of HTML elements by 93% in DOM nodes while achieving up to 10.1× faster reduction compared with state-of-the-art delta debugging approaches (HDD [25] and Perses [36]).

In summary, this paper makes the following contributions:

- **Cross-context differential oracle for iframe rendering bugs.** We propose a differential oracle that compares identical HTML content across top-level and iframe contexts, enabling the detection of iframe-specific rendering bugs that are missed by existing same-context approaches.
- **Layout-aware input reduction via search space redefinition.** We propose an HTML input reduction technique that leverages rendering semantics to map visual inconsistencies to DOM elements, thereby redefining the effective search space and pruning irrelevant elements while preserving bug-inducing behavior.
- **A fully automated end-to-end framework from bug detection to debugging.** We implement ICHECK, the first integrated framework for both detecting and debugging frame-specific rendering bugs in web browsers, and show that it significantly outperforms state-of-the-art techniques on real-world browsers on both bug detection capability and reduction efficiency.

2 Background

2.1 Security Threats of IFrame Rendering Bugs

In addition to harming user experience and causing business loss mentioned in Introduction (§1), iframe rendering bugs can also introduce serious security vulnerabilities. In particular, certain bugs allow untrusted content to escape beyond the boundary of an iframe and appear on the outer host page. This can lead to spoofing attacks, such as phishing, where malicious content mimics trusted interface elements. This attack exploits a *frame escape bug* that causes untrusted content to be integrated with the host page, making it appear legitimate and potentially deceiving users.

Threat Model. Consider an attacker who creates a malicious advertisement web page that exploits a frame escape bug and successfully registers it on a public advertising platform such as Google Ads [21]. A third-party website developer who is unaware of this threat embeds this malicious content via an iframe as part of their site’s advertisement integration [1]. Due to the rendering bug, the malicious content escapes the boundaries of the iframe and appears seamlessly integrated with the host page’s layout. This visual deception can mislead users into believing the malicious content originates from the trusted site. By exploiting this misperception, the attacker can launch spoofing-based attacks such as phishing,

¹A top-level HTML document refers to the main HTML document that is not embedded in any other HTML document; it serves as the root context for rendering the page [41].



Figure 2: UI spoofing attack using an iframe escape bug. An attacker embeds a fake interface into an iframe and aligns it over the legitimate page, causing the user to unintentionally click a maliciously repositioned button.

clickjacking, and user interface (UI) spoofing. Similar attack scenarios have been reported in the CVE database², and major browser vendors (e.g., Chrome and Firefox) have applied security patches in response to such cases.

Attack Scenario. Figure 2 illustrates a UI spoofing attack scenario that exploits a frame escape bug. Figure 2a presents the account settings page of a trusted website that contains “Update” account button on the left and “Delete” account button on the right. The box labeled “Advertisement” indicates the designated location of an iframe intended to display content from a public advertising platform. Figure 2b depicts the malicious content crafted by the attacker who replicates the legitimate UI but intentionally swaps the visual positions of the “Update” and “Delete” buttons. This spoofed content is rendered outside the iframe’s boundary and aligned to overlap the original button area on the host page, as illustrated in Figure 2c, making the spoofing visually indistinguishable to the user. As a result, when the user attempts to click the “Update” button (believing it to be in its expected position), the click actually triggers the “Delete” action, which is placed beneath the spoofed UI and unintentionally deletes the account.

2.2 Differential Oracles for Rendering Bugs

Differential testing operates under the premise that different implementations of the same specification should behave consistently; thus, any difference among outputs may indicate the presence of a bug. Under this premise, differential testing effectively detects a bug without requiring ground truths, which are difficult to obtain automatically. Due to this advantage, differential testing has been widely adopted in domains with multiple implementations of a common specification, such as compilers [4, 5, 12, 20, 46], deep learning libraries [9, 10, 44], symbolic execution engines [18], and hardware systems [17].

Differential testing is particularly effective for detecting rendering bugs, which often manifest as semantic errors, such as visual discrepancies in the rendered output, rather than as system crashes or explicit error messages. Since HTML rendering involves complex interactions among HTML elements, CSS styles, and JavaScript (JS) code, it is difficult to automatically specify the expected output for a given HTML page. Due to this challenge, all existing work performs differential testing to detect rendering bugs [34, 35, 50], such as

rendering regression bugs [34], JavaScript-specific render-update bugs [35], and general rendering bugs [50].

2.3 Debugging Support for Rendering Bugs

Debugging rendering bugs involves identifying which parts of the HTML input are responsible for the observed visual inconsistency. Although no prior work exists specifically targeting the debugging of browser rendering bugs, applying delta debugging [48] can be an option as discussed in §1. Delta debugging isolates the minimal set of failure-inducing parts of an input by iteratively removing parts of the input and checking whether the inconsistency with and without those removed parts is reproduced. Delta debugging relies on predefined search strategies—such as systematic partitioning [48] or heuristic [52] and probabilistic exploration [40]—to traverse the input space and identify failure-inducing input components.

To improve the effectiveness of delta debugging on structured inputs, prior work has extended delta debugging to formats such as XML, HTML, and abstract syntax trees (AST) by leveraging their hierarchical structure. Hierarchical delta debugging (HDD [25]) and Perses [36] leverage tree-based [25] or grammar-based [36] representations to preserve syntactic validity during iterative reductions. However, despite these advances, they remain agnostic to rendering semantics (e.g., rendering layouts) and are not designed to account for visual inconsistencies in browser rendering.

3 Challenges

Lack of Test Oracles for IFrame Rendering Bugs. A fundamental challenge in detecting iframe rendering bugs is the lack of suitable test oracles for these bugs. Existing test oracles for rendering bugs have limited ability to detect iframe-specific bugs. These oracles typically compare the visual outputs of the same top-level document rendered in different browsers [50], versions [34], or pages [35]. This design reflects an implicit assumption: rendering bugs manifest as inconsistencies within the same rendering context.

However, when the same HTML content is embedded in an iframe, different browsers may produce the same incorrect output—particularly if the bug is triggered by the embedding context rather than by differences in browser implementations. In such cases, the existing oracles fail to detect the iframe-specific bug because they treat matching outputs as correct even when both are incorrect. This illustrates the lack of iframe-aware test oracles and motivates a new technique that accounts for different rendering contexts.

Inefficiency for Debugging Rendering Bugs. There are two main challenges for debugging rendering bugs. First, even though existing delta debugging techniques [25, 36] can be applied for debugging HTML inputs, they become particularly expensive in the context of rendering bugs because each iteration requires rendering the HTML page in a web browser. Rendering is a computationally expensive process, as the browser must execute a complex pipeline including DOM construction, layout calculation, compositing, and rasterization. Consequently, each rendering operation incurs significant cost. For example, prior work [50] reports that for detecting visual inconsistencies in a browser, rendering operations account for 87.03% of the total runtime, and image comparison accounts for 12.01%, leaving only 0.97% for all other tasks.

²<https://nvd.nist.gov/vuln/detail/CVE-2014-1748>, <https://nvd.nist.gov/vuln/detail/CVE-2014-3201>, <https://nvd.nist.gov/vuln/detail/CVE-2022-28286>, <https://nvd.nist.gov/vuln/detail/CVE-2023-7013>

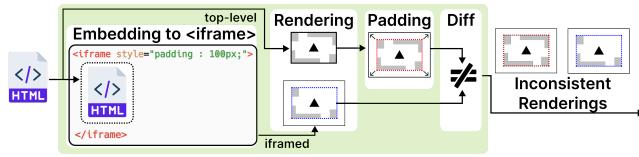


Figure 3: Overview of Bug Detection phase.

Second, as discussed in §2.3, delta debugging relies on predefined search strategies [40, 48, 52] to traverse input space. While those strategies differ in how they explore the search space, they treat all parts of the HTML uniformly, which may result in unnecessary exploration. Considering the high overhead of rendering operations discussed in the first challenge, unnecessary exploration leads to a large number of expensive rendering operations, making the process inefficient in practice.

A practical debugging approach, therefore, requires a more efficient input reduction strategy that reduces the number of expensive rendering operations. To achieve this, an effective technique should redefine the effective search space, rather than merely optimizing how it is explored as in prior work [40, 48, 52]. In particular, for rendering bugs, many parts of an HTML input are semantically irrelevant to the observed visual inconsistency. Thus, a reduction approach should identify and focus only on the elements that contribute to the inconsistency, while pruning irrelevant portions of the input.

4 Approach

In this section, we present our approach, ICHECK, which detects iframe rendering bugs and reduces HTML input that exhibits such bugs for further debugging support. ICHECK consists of two main phases: Bug Detection (§4.1) and Input Reduction (§4.2).

In Bug Detection phase, ICHECK takes an HTML test input and checks whether it triggers a rendering inconsistency between the different contexts (*i.e.*, top-level vs. iframe contexts). Once an inconsistency is detected, ICHECK assists in debugging through Input Reduction phase. In this phase, ICHECK redefines the effective search space by retaining only those DOM elements visually and structurally close to the inconsistency, and yields a reduced HTML input that isolates a bug-inducing subset of elements.

4.1 IFrame Rendering Bug Detection

In this phase, ICHECK leverages differential testing to determine whether the test HTML input triggers iframe rendering bugs (See §5.1 for details on input generation). Specifically, ICHECK renders the same HTML content in two different contexts (*i.e.*, top-level and iframe) and compares the resulting rendering outputs. Figure 3 presents an overview of the bug detection phase, which consists of four steps: *embedding*, *rendering*, *padding*, and *diff*.

ICHECK first embeds the input HTML into an iframe to prepare for rendering in both top-level and iframe contexts. ICHECK then executes the rendering in both contexts and captures screenshots of the resulting renderings. To ensure that potential visual differences of these renderings are due to rendering behavior rather than layout discrepancies, in the padding step, the image of the top-level context is padded to equal dimensions of the iframe context. Finally,

ICHECK compares the padded rendering with the iframed rendering to identify visual inconsistencies between the two contexts. If an inconsistency is detected, ICHECK outputs the pair of these rendered images, which are used as input to the subsequent input reduction phase. The following describes each step in detail.

4.1.1 Embedding. In this step, ICHECK creates a new HTML document that embeds the HTML input into an iframe (dashed box in “Embedding to <iframe>” of Figure 3). This embedding prepares the HTML for rendering in a different context and enables a comparison against its top-level rendering. Note that ICHECK applies CSS padding around the <iframe> element (solid box in “Embedding to <iframe>” of Figure 3), which is essential for exposing iframe escape bugs discussed in §2.1. By applying sufficient padding (e.g., padding: 100px;), ICHECK ensures that any escaped content remains within the viewport³ and can be captured during the diff step.

4.1.2 Rendering and Padding. In the rendering step, the original HTML input is rendered in the top-level context, while the embedded HTML generated in the previous step is rendered in an iframe context. The visual outputs from both renderings are then captured as images. In the padding step, ICHECK adds margins around the top-level rendered image, matching the 100px padding applied to the iframe during embedding. This alignment ensures that both renderings cover the same content region, so any differences observed in the diff step reflect genuine rendering inconsistencies rather than image size or layout mismatches.

4.1.3 Diff. Finally, ICHECK determines whether the HTML input triggers an iframe rendering bug by comparing two images rendered in the top-level and iframe contexts. Specifically, ICHECK converts both images into perceptual hash (also called pHash) values using a perceptual hashing algorithm [47], which compares images based on their visual similarity rather than exact pixel-by-pixel equality. This pHash-based approach is commonly used in existing rendering bug detection techniques [34, 35, 50]. If the difference between the pHash values of two images exceeds a predefined threshold, ICHECK classifies the rendered images as inconsistent renderings and passes them to the debugging phase (§4.2).

In summary, our bug detection phase provides two key advantages. First, by comparing renderings across different contexts, ICHECK overcomes the limitations of existing oracles that support the comparison only within the same context, thus failing to detect iframe-specific rendering inconsistencies. Second, by incorporating padding into the rendering process, ICHECK is able to detect rendering inconsistencies that occur both within the iframe content and beyond its boundary using only a single image comparison.

4.2 Layout-aware Input Reduction

In this phase, ICHECK performs layout-aware DOM tree reduction to reduce the search space of the input that is unlikely to contribute to the rendering inconsistency. Unlike existing delta debugging techniques [25, 36] that operate purely on syntactic structure without considering rendering behavior, our approach leverages not

³In web development, the viewport is the visible area of a web page within the browser window. It defines the rectangular region through which the user views the content of the page [42].

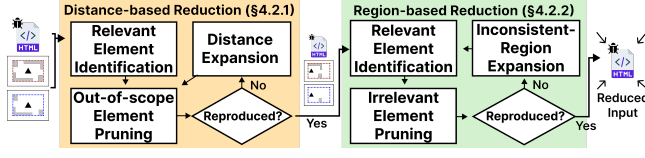


Figure 4: Overview of Input Reduction phase.

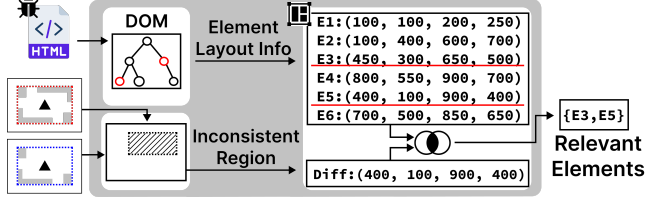


Figure 5: Illustration of the relevant element identification procedure.

only DOM hierarchy but also rendering semantics (*i.e.*, visual layout information) for pruning.

ICHECK performs a two-stage DOM tree reduction as shown in Figure 4, consisting of a *distance-based* and *region-based* strategy. The key intuition behind both strategies is that DOM elements closer to the inconsistent region—either structurally in the DOM tree or visually in the rendered output—are more likely to be responsible for the rendering inconsistency.

4.2.1 Distance-based Reduction. This strategy leverages the structural relationships in the DOM tree by retaining elements that lie within a small edge distance from the nodes identified as relevant to the inconsistency. It consists of two sub-procedures: relevant element identification and out-of-scope element pruning.

Relevant Element Identification. ICHECK first identifies the DOM elements that are relevant to the visual inconsistency detected in Bug Detection phase. To bridge the gap between the visual information in the rendered images and the underlying HTML’s DOM structure, ICHECK leverages the rendering semantics of DOM elements. This allows ICHECK to link the visual inconsistency with the corresponding DOM elements.

Figure 5 illustrates this procedure. ICHECK identifies the regions of visual inconsistency between the two rendered images. Specifically, ICHECK computes a set of *inconsistent regions*, each represented as a bounding box that captures an area where visual discrepancies are detected. For example, in Figure 5, the shaded box in the bottom left illustrates an inconsistent region that corresponds to a visual difference between the two given images. This bounding box helps localize areas of interest and redefine the scope to only those DOM elements that are likely responsible for the inconsistency.

ICHECK next obtains the DOM tree and collects the layout information of all DOM elements. Every DOM element is represented by a bounding box defined by four coordinates (x_1, y_1, x_2, y_2) , where (x_1, y_1) is the top-left and (x_2, y_2) the bottom-right corner. Using these bounding boxes, ICHECK identifies relevant elements by checking for any intersections with the inconsistent regions.

Out-of-scope Element Pruning. After identifying the relevant elements, ICHECK applies a pruning strategy to remove DOM nodes

Input: DOM Tree of original HTML, $DOM_{original}$
Input: Relevant Elements Set, RE $//\{E_3, E_5\}$
Output: Reduced HTML, $HTML_{reduced}$

```

1:  $Distance_{threshold} \leftarrow 0$ 
2: while  $True$  :
3:    $RetainingNodes \leftarrow \emptyset$ 
4:   for  $Node \in Traverse(DOM_{original})$  :
5:      $Distance \leftarrow GetDistanceAtTree(DOM_{original}, Node, RE)$ 
6:     if  $Distance \leq Distance_{threshold}$  :
7:        $RetainingNodes \leftarrow RetainingNodes \cup Node$ 
8:    $DOM_{reduced} \leftarrow DOM_{original}$ 
9:   for  $Node \in Traverse(DOM_{original})$  :
10:    if  $Node \notin RetainingNodes$  :
11:       $DeleteAndAttach(DOM_{reduced}, Node)$ 
12:    $HTML_{reduced} \leftarrow Serialize(DOM_{reduced})$ 
13:    $Result \leftarrow CheckReproducibility(HTML_{reduced})$ 
14:   if  $Result == True$  :
15:     break
16:   else :
17:      $Distance_{threshold} \leftarrow Distance_{threshold} + 1$ 
18: return  $HTML_{reduced}$ 

```

Algorithm 1: Algorithm for distance-based tree reduction.

unlikely to affect the rendering inconsistency. Specifically, it retains only those elements whose structural distance—the number of edges in the DOM tree separating two nodes—to any relevant element is less than or equal to a predefined threshold; And, all others nodes are pruned. This preserves nearby context (*e.g.*, parents, children, or siblings) needed for the bug to manifest. As a result, ICHECK reduces the DOM tree substantially yet still maintains the conditions required to reproduce the inconsistency. After pruning, ICHECK verifies whether the reduced HTML input still reproduces the visual inconsistency. If the inconsistency persists, the reconstruction is considered valid, and ICHECK proceeds to the *region-based reduction* stage for further reduction.

Distance Expansion. If the inconsistency is lost, it indicates that essential nodes are mistakenly pruned. In this case, ICHECK incrementally increases the structural distance threshold by one and reintroduces the corresponding nodes until the visual inconsistency reappears. This ensures that the reduced DOM tree retains all nodes necessary to reproduce the bug while being substantially smaller than the original HTML input.

Algorithm 1 describes the entire workflow of the distance-based tree reduction. It computes the structural distance between each node and the relevant elements (Line 5), and retains those within the predefined threshold (Lines 6–7). Then, the algorithm prunes all other nodes (Lines 8–11). After pruning, the reduced DOM is then serialized into HTML and checked for bug reproducibility (Lines 12–13). If the bug is reproduced, the process terminates (Lines 14–15), otherwise, the distance threshold is increased (Lines 16–17) and the procedure is repeated until reduced HTML occurs visual inconsistency. Finally, the algorithm returns the reduced HTML that preserves the bug (Line 18).

4.2.2 Region-based Reduction. While the distance-based strategy can effectively reduce the DOM tree, it may still retain some nodes that are not directly involved in the rendering bug. In particular, not

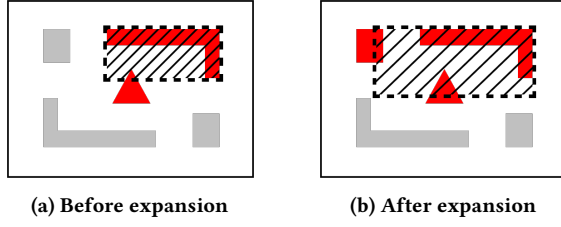


Figure 6: Relevant elements (red) identified before & after expanding inconsistent region (shaded box).

all ancestors, siblings, or children preserved within structural distance affect visual inconsistency. To further reduce the search space by pruning such irrelevant nodes, ICHECK performs the region-based reduction strategy, which retains only the elements visually associated with the inconsistent regions in the rendered output.

Relevant Element Identification. In the region-based strategy, ICHECK applies the same relevant element identification procedure as described in the distance-based strategy (§4.2.1). However, this procedure must be performed again using the reduced HTML returned from the distance-based strategy because pruning in that stage can change the DOM hierarchy and layout, thereby shifting element positions and changing the inconsistent regions.

Irrelevant Element Pruning. After re-identifying relevant elements on the reduced input, ICHECK prunes the DOM tree by retaining only those elements intersecting with the updated inconsistent region. ICHECK then verifies whether the reduced input still reproduces the visual inconsistency, as in the distance-based strategy.

If the inconsistency still occurs, the inconsistent region may be larger than necessary and may include elements that are not essential for reproducing the bug. To further prune the DOM tree, ICHECK gradually shrinks the inconsistent region and repeatedly performs region-based reduction based on the updated region. At each iteration, ICHECK recalculates the relevant elements, performs irrelevant element pruning, and checks whether the bug is still reproduced. This process continues until the smallest possible region that preserves the inconsistency is found.

Inconsistent Region Expansion. If the inconsistency is lost, it indicates that the inconsistent region is too small to retain essential nodes. In this case, ICHECK incrementally expands each inconsistent region to restore relevant elements that may have been missed. In the first n iterations (default: $n = 10$), the inconsistent regions grow toward the viewport boundaries using quadratically accelerated scaling, providing fine-grained control and gradually including nearby elements with increasing precision. The width and height of the expanded region at iteration i (i.e., w_i and h_i , respectively) are calculated as:

$$w_i = w_0 \cdot \left(\frac{w_{\text{viewport}}}{w_0} \right)^{\left(\frac{i}{n} \right)^2}, \quad h_i = h_0 \cdot \left(\frac{h_{\text{viewport}}}{h_0} \right)^{\left(\frac{i}{n} \right)^2}$$

where w_0 and h_0 are the width and height of the initial inconsistent region.

Once the region covers the full viewport (i.e., $i > n$), ICHECK expands the region exponentially beyond the viewport as follows:

$$w_i = w_{\text{viewport}} \cdot 2^{i-n}, \quad h_i = h_{\text{viewport}} \cdot 2^{i-n}$$

After each expansion iteration, ICHECK recalculates the relevant elements, re-applies irrelevant element pruning, and checks whether

the bug is reproduced, as illustrated in Figure 6. If the relevant element set encompasses the entire DOM, it means no further reduction is possible without losing the inconsistency; therefore, ICHECK terminates the reduction and returns the result from the previous distance-based strategy.

5 Evaluation

To evaluate ICHECK, we investigate the following research questions (RQs):

- **RQ1:** How does ICHECK’s oracle perform compared to the state-of-the-art oracles for detecting iframe rendering bugs? (§5.2)
- **RQ2:** Can ICHECK find real-world iframe rendering bugs? (§5.3)
- **RQ3:** How efficiently does ICHECK’s input reduction technique reduce inputs? (§5.4)
- **RQ4:** Do individual components of our input reduction approach affect the effectiveness of reduction? (§5.5)
- **RQ5:** How do the bugs detected by ICHECK manifest in real browsers? (§5.6)

5.1 Experimental Setup

Implementation. We implemented ICHECK in Python 3. For browser automation, we used the Playwright [24] to control both Chromium and Firefox in headless mode. To detect rendering inconsistencies, we computed perceptual hashing (pHash) using the ImageHash library [2]. All DOM manipulations during the Input Reduction were conducted via Playwright’s evaluation interface.

Input Generation. For input generation, we adopted Metamong’s input generation module [35], which builds on Domato [14] to produce structurally diverse HTML inputs. This module ensures that no crashes occur during testing by stripping JS code and injecting controlled rendering updates, thereby making the inputs more suitable for testing browser rendering bugs.

Browsers. We ran our experiments on two most popular browsers: Chrome  and Firefox . Their browser engines (i.e., Chromium and Gecko) account for 96% of global usage [39].

Environment. All experiments were conducted on a server with two Intel Xeon Gold 6248R CPUs (24 cores each) and 512 GB RAM, running Ubuntu 22.04 LTS.

5.2 RQ1. Effectiveness of Test Oracles

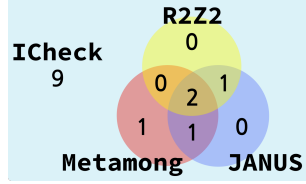
We compared ICHECK’s oracle with the three state-of-the-art oracles: *Regression Oracle* from R2Z2 [34], *Render-update Oracle* from Metamong [35], and *Visual Delta Consistency Oracle* from Janus [50]. We selected these oracles because they are the state-of-the-art approaches specifically designed to find browser rendering bugs.

We prepared 100,000 HTML inputs using the input generation method described in §5.1. All inputs were rendered inside an iframe during evaluation. We used  v134 (Mar. 2025) and  v134 (Jan. 2025) as the primary browsers. To perform cross-version comparisons required by Regression Oracle, we additionally used  v131 (Dec. 2024) and  v130 (Sep. 2024), following the three-month release gap used in the R2Z2 study.

We ran each oracle on the same browsers under the same time budget and input conditions. To ensure a fair comparison, all other factors including rendering environment, screenshot capture, and visual inconsistency checks remained unchanged. We then counted

Table 1: The number of iframe rendering bugs in Chrome and Firefox detected by each tool’s oracle (RQ1).

Tool (Oracle)	Chrome v134	Firefox v134	Total
ICHECK (Our Cross-context Oracle)	10	4	14
R2Z2 (Regression Oracle)	2	1	3
Metamong (Render-update Oracle)	3	1	4
Janus (Visual Delta Consistency Oracle)	3	1	4

**Figure 7: Venn diagram of bugs discovered by ICHECK and baselines (RQ1).**

the number of distinct iframe rendering bugs each oracle found. Thanks to our input reduction technique, we could efficiently deduplicate bug-triggering inputs and accurately count distinct bugs.

Table 1 summarizes the iframe rendering bugs found by each tool’s oracle. Figure 7 shows the overlap among oracles. ICHECK detected all five bugs found by other oracles and discovered a total of 14 bugs, demonstrating that our oracle extends the coverage of existing oracles.

We further analyzed why the prior work failed to detect many iframe rendering bugs. The regression oracle of R2Z2 found only three bugs in our evaluation. This is because the oracle compares rendering outputs solely across browser versions (e.g., Chrome v131 vs. Chrome v134). If a rendering bug persists across versions, the oracle produces identical outputs, resulting in 11 false negatives.

The render-update oracle of Metamong missed 10 of the 14 bugs detected by ICHECK. We observed these missing iframe rendering bugs displayed no differences between the pre- and post-render update states. Since the oracle detects discrepancies solely by comparing these two outputs, it is incapable of detecting iframe rendering bugs that do not produce differences between two states.

The visual delta consistency oracle of Janus also missed 10 bugs due to the same failure reason of Metamong’s oracle. This oracle detects the visual delta inconsistency induced by rendering updates across different browsers (i.e., Chrome v134 vs. Firefox v134). Since missed bugs are unaffected by the rendering update, the visual deltas concluded that before and after the update were identical in each browser, resulting in no visual delta inconsistency across browsers and leading to 10 false negatives.

In summary, the baselines missed many iframe rendering bugs because their oracles compare visual outputs only between same-level documents (i.e., iframe vs. iframe) rendered in different browsers or different versions. In contrast, ICHECK successfully found all bugs missed by the existing oracles by comparing rendering results across different contexts (i.e., top-level vs. iframe).

Table 2: ICHECK’s bug detection capability for 16 known bugs (RQ2).

Known Bugs (Browser)	Detect?	Known Bugs (Browser)	Detect?
CVE-2014-3201 (Chrome v37)	✓	Issue #40157616 (Chrome v88)	✓
CVE-2022-28286 (Firefox v93)	✓	Issue #40801723 (Chrome v95)	✓
CVE-2023-7013 (Chrome v118)	✓	Issue #40209826 (Chrome v96)	✓
Issue #41477467 (Chrome v76)	✓	Issue #40804814 (Chrome v96)	✓
Issue #40668575 (Chrome v79)	✓	Issue #40809900 (Chrome v96)	✓
Issue #40709677 (Chrome v84)	✓	Issue #40248909 (Chrome v108)	✓
Issue #40150448 (Chrome v85)	✓	Issue #40944025 (Chrome v119)	✓
Issue #40158873 (Chrome v87)	✓	Issue #365919656 (Chrome v134)	✓

Table 3: Distinct unknown bugs detected by ICHECK (RQ2).

Bug disclosure	Chrome	Firefox	Total
Not reported	3	3	6
Reported	14	1	15
└ Confirmed	14	1	15
└ Fixed	‡4	0	4

‡ One of the fixed bugs is an iframe escape vulnerability.

5.3 RQ2. Effectiveness of Bug Detection

To evaluate the bug detection capability of ICHECK, we conducted two experiments: (1) detecting known bugs and (2) discovering previously unknown bugs.

Detecting Known Bugs. We built a benchmark dataset of known bugs by collecting all the reported iframe rendering bugs from CVE database [37] and Chrome/Firefox bug repositories [15, 26], using the keywords *iframe* and *rendering*. We only included bugs for which a Proof-of-Concept (PoC) HTML input was available in the corresponding report. As a result, we obtained 16 known bugs. For each bug, we tested the PoC HTML input using our oracle to determine whether the bug could be detected.

Table 2 presents the detection results for the 16 known bugs. The third column indicates whether ICHECK detected the corresponding bug with ✓ denoting detection. In all 16 cases, ICHECK successfully detected the bug.

Discovering Unknown Bugs. We ran ICHECK on the versions of Chrome v136 and Firefox v137, using 300,000 HTML inputs. In addition, we included results from previous experiments (§5.2) on Chrome v134 and Firefox v134 with 100,000 inputs. In total, ICHECK tested 400,000 inputs and detected 4,957 visual inconsistencies. Then, we applied our input reduction technique to obtain the reduced HTML input and manually analyzed these HTML inputs to count the number of distinct bugs. As a result, ICHECK detected 21 distinct iframe rendering bugs in total.

Table 3 summarizes 21 distinct bugs, and Table A.1 in Appendix (Supplemental Material) provides detailed information for each bug. Among 21 bugs, six were not reported through official bug reports (and therefore are not included in Table 2), but had already been fixed internally by the time we attempted to report. We reported the remaining 15 bugs to their corresponding vendors. All were confirmed by developers. Among them, four were fixed by the browser vendor, all in Chrome. Notably, one of the fixed bugs was a security

Table 4: Comparison results of ICHECK and state-of-the-art delta debugging techniques (RQ3).

(a)  Chrome (with 445 HTML inputs)									
Tool	Reduction Time			# of Avg. DOM Nodes			Byte-size of HTML File		
	Total	Avg. (s)	Time Ratio (×)	Original	Reduced	Reduction Rate (%)	Original (KB)	Reduced (KB)	Reduction Rate (%)
ICHECK	2h 49m	22.8s	1.00		11.3	92.63		1.46	96.29
Perses	30h 15m	225.9s	9.92		8.9	94.20		0.78	98.02
HDD + ddmin	27h 49m	206.2s	9.05	153.1	7.1	95.38	39.39	0.15	99.62
HDD + probdd	22h 50m	165.9s	7.29		7.2	95.32		0.16	99.59
HDD + Wprobdd	24h 21m	178.2s	7.83		7.2	95.32		0.25	99.37

(b)  Firefox (with 197 HTML inputs)									
Tool	Reduction Time			# of Avg. DOM Nodes			Byte-size of HTML File		
	Total	Avg. (s)	Time Ratio (×)	Original	Reduced	Reduction Rate (%)	Original (KB)	Reduced (KB)	Reduction Rate (%)
ICHECK	1h 49m	33.2s	1.00		12.7	92.00		1.11	97.13
Perses	7h 39m	140.0s	4.22		7.3	95.42		0.50	98.71
HDD + ddmin	17h 7m	312.8s	9.43	158.3	6.2	96.11	38.69	0.06	99.84
HDD + probdd	18h 20m	335.1s	10.10		6.2	96.10		0.16	99.59
HDD + Wprobdd	15h 7m	276.4s	8.33		6.2	96.10		0.05	99.87

bug—an iframe escape vulnerability—for which we received a CVE (CVE-2025-0448) and a security bounty reward.

As shown in our results, ICHECK is effective at detecting visual inconsistencies and can even uncover security-critical rendering bugs that are recognized and addressed by major browser vendors.

5.4 RQ3. Effectiveness of Input Reduction

To evaluate the effectiveness of our layout-aware input reduction technique, we compared ICHECK with the state-of-the-art delta debugging techniques that support HTML input format, Perses [36] and hierarchical delta debugging (HDD) [25]. Additionally, to cover different search strategies, we combined HDD with the state-of-the-art delta debugging search strategies, ddmin [48], probdd [40], and Wprobdd [52]. We did not combine them with Perses because it performs its own predefined input reduction procedure.

To conduct this experiment, we first collected inputs exhibiting visual inconsistencies from the 100,000 HTML inputs that were used in RQ1 (§5.2). Among these inputs, we obtained 1520 inputs that manifest visual inconsistencies: 1,149 from  v134 and 371 from  v134. From these 1,520 inputs, we excluded inputs that could not be processed by the baseline tools due to syntactic irregularities introduced from Domato (*i.e.*, input generation tool), which prevent parsing into valid DOM trees by the ANTLR parser used in Perses and HDD. After filtering out these noisy inputs, we finally obtained 642 inputs: 445 from  v134 and 197 from  v134. We ran the baselines to perform input reduction on this dataset. To evaluate the performance, we measured (1) reduction time (*i.e.*, the time required to complete the reduction), (2) DOM reduction rate (*i.e.*, the proportion of DOM nodes removed), and (3) byte-size reduction rate (*i.e.*, the proportion of bytes removed from the input HTML file).

Table 4 presents the comparison results between ICHECK and the baselines. The results show that ICHECK consistently achieves

significantly faster reduction time than the baselines, while achieving comparable reduction rates. On average, ICHECK removed more than 92% of DOM nodes, which is slightly lower than the reduction rates achieved by Perses or HDD-based combinations, but the overall difference in # of average DOM nodes remains small (2.4 to 4.2 nodes in  and 5.4 to 6.5 nodes in ). In contrast, the reduction time is significantly faster, with ICHECK completing the reduction 7.29 to 9.92 times and 4.22 to 10.1 times faster than the baselines in  and  respectively. This indicates that ICHECK’s input reduction technique for redefining the effective search space is effective in reducing the search space itself. This results in quickly producing compact, bug-inducing inputs, enabling efficient debugging support.

In terms of the byte size of the input HTML file, ICHECK reduces the original input’s byte size by 96.29% in  and 97.13% in . Although these reductions are significant, the baseline reducers achieve higher byte-size reduction rates, ranging from 98.02% to 99.62% in  and from 98.71% to 99.87% in . This difference comes from the different reduction objectives and granularities of the techniques. ICHECK performs layout-aware reduction at the granularity of DOM elements by pruning rendering-irrelevant regions based on visual inconsistencies, whereas the baseline reducers perform finer-grained tree-based [25] or grammar-based minimization [36], including the reduction of CSS properties and HTML attributes. Thus, the approaches reduce different aspects of an HTML input and are complementary.

For example, consider Figure 8 that illustrates how much ICHECK simplifies an original HTML input. The original HTML input contains 344 DOM nodes (Figure 8a), resulting in a highly complex rendering that developers must manually inspect without automated support. In contrast, ICHECK reduces the input to only 10 DOM nodes (Figure 8b) while preserving the bug. This would significantly reduce the time required for debugging. Moreover, compared to the baselines, ICHECK can reduce the input reduction time up to

Table 5: Comparison of the same delta debugging techniques with and without reduced HTML input by ICHECK (RQ3).

(a) Chrome (with 445 HTML inputs)							
Tool	ICHECK Reduced HTML			Original HTML			Time Ratio (×)
	Avg. DOM Nodes (#)	Reduction Rate (%)	Total Time	Avg. DOM Nodes (#)	Reduction Rate (%)	Total Time	
Perses	8.2	94.63	14h 25m	8.9	94.20	30h 15m	2.12
HDD + ddmin	7.0	95.43	19h 34m	7.0	95.38	27h 49m	1.39
HDD + probdd	7.0	95.43	14h 52m	7.2	95.32	22h 50m	1.51
HDD + Wprobdd	7.0	95.42	18h 48m	7.2	95.32	24h 21m	1.26

(b) Firefox (with 197 HTML inputs)							
Tool	ICHECK Reduced HTML			Original HTML			Time Ratio (×)
	Avg. DOM Nodes (#)	Reduction Rate (%)	Total Time	Avg. DOM Nodes (#)	Reduction Rate (%)	Total Time	
Perses	7.4	95.35	4h 7m	7.3	95.42	7h 39m	1.62
HDD + ddmin	6.3	95.99	13h 4m	6.2	96.11	17h 7m	1.25
HDD + probdd	6.4	95.98	11h 18m	6.2	96.10	18h 20m	1.54
HDD + Wprobdd	6.4	95.99	12h 1m	6.2	96.10	15h 7m	1.20

9.92 times faster. As the number and complexity of HTML inputs increase, the efficiency advantage of ICHECK becomes even more pronounced.

Additionally, as shown in Table 4, ICHECK significantly improves reduction time, while the baseline tools achieve slightly better reduction rates (approximately 2.7%p in Chrome, 4.1%p in Firefox). Although the difference is not significant, these higher reduction rates can also be achieved more efficiently by applying ICHECK’s already-reduced inputs to the baseline tools, as shown in Table 5. When the baseline reducers are applied with the reduced HTML input by ICHECK, the total reduction time (including the time spent in ICHECK) is improved by 1.2× to 2.12× compared to using the baseline reducers alone.

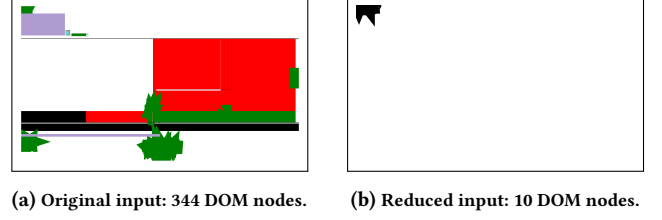
The results of RQ3 indicate that ICHECK is not only effective as a standalone reduction technique but also serves as a useful pre-processing step that accelerates existing input reduction methods. In practice, developers can use ICHECK to quickly obtain a reproducible, compact input, and optionally apply additional reducers if a smaller input is required, achieving both efficiency and high reduction quality.

5.5 RQ4. Ablation Study

We evaluate the effectiveness of each input reduction component by running ICHECK with only that component enabled. To this end, we compare ICHECK with the following two ablation variants:

- **ICHECK-DIST**: applies only the distance-based reduction procedure (§4.2.1)
- **ICHECK-REGION**: applies only the region-based reduction procedure (§4.2.2)

We evaluated each ablation variant on the 642 inputs used in RQ3 (§5.4) and measured both the reduction time and reduction rate. These measurements allow us to assess the contribution of individual debugging components to the overall effectiveness of reducing HTML inputs that exhibit rendering inconsistencies.

**Figure 8: Example of a reduced HTML input.****Table 6: Results of ICHECK’s ablated variants (RQ4).**

Browser	Tool	Reduction Time		Avg. DOM Nodes		
		Total	Avg. (s)	Before (#)	After (#)	Reduction Rate (%)
Chrome	ICHECK	2h 49m	22.8s		11.3	92.63
	└ ICHECK-DIST	0h 46m	6.2s	153.1	32.4	78.83
	└ ICHECK-REGION	2h 53m	23.4s		14.9	90.26
Firefox	ICHECK	1h 49m	33.2s		12.7	92.00
	└ ICHECK-DIST	0h 24m	7.2s	158.3	24.9	84.26
	└ ICHECK-REGION	2h 9m	39.3s		27.5	82.62

Table 6 shows the comparison results between ICHECK and its ablated variants. The results reveal a clear trade-off between reduction effectiveness and efficiency across the variants.

ICHECK-DIST achieves fast reduction (e.g., Chrome: 6.2s/input, Firefox: 7.2s/input) by pruning nodes based solely on DOM tree distance. However, this strategy results in relatively low reduction rates (e.g., Chrome: 78.83%, Firefox: 84.26%), as it may retain nodes that are structurally close but irrelevant to the visual inconsistency.

In contrast, ICHECK-REGION leverages visually inconsistent regions to guide pruning, leading to higher reduction rates (e.g., Chrome: 90.26%, Firefox: 82.62%). However, this improvement comes at the cost of significantly higher reduction time (e.g., Chrome: 23.4s/input, Firefox: 39.3s/input).

These results indicate that the two strategies are complementary: ICHECK-DIST provides efficiency, while ICHECK-REGION improves reduction precision. By integrating both strategies, ICHECK effectively balances this trade-off, achieving both high reduction rates (e.g., Chrome: 92.63%, Firefox: 92.00%) and reasonable reduction time (e.g., Chrome: 22.8s/input, Firefox: 33.2s/input). Overall, these findings confirm that integrating proposed reduction techniques provides the most effective and comprehensive input reduction.

5.6 RQ5. Case Studies for Detected Bugs

In this section, we describe two representative previously unknown bugs that ICHECK detected during the experiments.

Case Study 1: CVE-2025-0448. Figure 9 shows the iframe escape vulnerability that ICHECK detected in Chrome. As shown in Figure 9b, a gray rectangle was rendered outside the blue dotted iframe boundary. This is a security violation of the expected visual containment. This bug is not limited to the gray box; it enables arbitrary images to be rendered outside the iframe, allowing the adversary to spoof the host page’s UI as discussed in §2.1.

Figure 9c presents the minimized PoC input that triggered this bug. Specifically, it was triggered by a select tag (Line 6) with an extreme transformation style (Line 4). Notably, this bug appeared

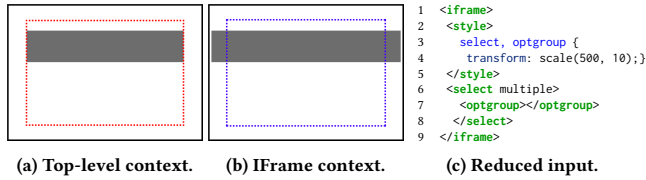


Figure 9: Case study #1: iframe escape vulnerability (CVE-2025-0448).

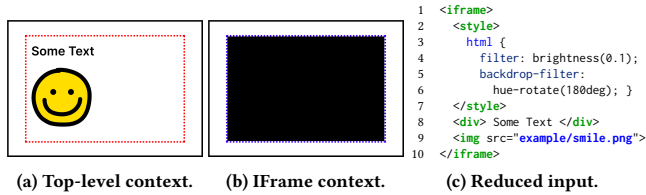


Figure 10: Case study #2: iframe rendering bug (Issue #420066688).

only in the iframe context and not in the top-level context (Figure 9a). The root cause of this bug was a structural flaw in Chrome’s rendering engine: layer clipping failed under certain scale conditions, causing transformed content to escape the iframe. Chrome vendor has patched the issue and provided a bug bounty reward.

Note that the input reduction approach of ICHECK played a crucial role in debugging this bug as well.

The original HTML input contained 425 DOM nodes, but ICHECK reduced it to just 6 DOM nodes (98.58% reduction rate) while preserving the visual inconsistency. This made it significantly easier to locate the transformed select tag responsible for the bug.

Case Study 2: Issue #420066688 in Chrome. Figure 10 presents another rendering bug in Chrome. Figure 10c shows the reduced input obtained via our debugging approach. Applying both the filter (Line 4) and backdrop-filter (Line 5) style attributes to the html tag resulted in the entire iframe content being rendered as a black box (Figure 10b). When the same HTML input was rendered in a top-level context (Figure 10a), however, all text and images appeared correctly. This bug is critical, as it renders the iframe content completely inaccessible and thereby severely degrades the user experience of the affected web page. Importantly, our cross-context oracle successfully detected this inconsistency by comparing the renderings in Figures 10a and 10b.

6 Discussion

Generalizability of ICHECK’s Debugging Technique. Our evaluation focused on iframe-specific rendering bugs since the oracle of ICHECK is explicitly designed to detect iframe rendering bugs. However, our underlying debugging technique is not limited to just iframe-specific bugs. Our layout-aware reduction approach is broadly applicable to rendering bugs in general. Any visual inconsistency can benefit from our work because, as far as HTML input and a pair of inconsistent renderings are available, our technique can effectively reduce the HTML input.

To the best of our knowledge, ICHECK is the first work to introduce an automated debugging technique for rendering-specific bugs in web browsers. Prior studies have primarily focused on defining

oracles for detecting different rendering bug categories [34, 35, 50], but have not addressed how to reduce or localize the inputs once a bug is detected. By filling this gap, our debugging technique extends beyond iframe-specific rendering bugs and provides a general methodology for isolating bug-inducing elements and styles. This generality suggests that ICHECK not only complements existing detection approaches but also establishes a foundation for future research on debugging rendering bugs.

7 Threats to Validity

Generalizability of the Results. Our evaluation was limited to  and , and the results may not generalize to other browsers or rendering engines, such as WebKit. Nevertheless, the proposed cross-context oracle is browser-agnostic: it only compares the rendering of the same input in top-level and iframe contexts and does not rely on browser-specific modifications. Because these rendering contexts are supported across modern browsers, extending ICHECK to additional browsers is technically feasible. However, further experiments are required to confirm whether ICHECK achieves comparable effectiveness on other rendering engines.

Limited JavaScript Behaviors. Our evaluation covers only a restricted range of JavaScript behaviors. Specifically, we removed the general-purpose JavaScript code generated by Domato for JavaScript engine fuzzing and instead injected controlled operations that alter rendering updates, such as removing elements and adding or modifying CSS. Consequently, our evaluation may miss rendering bugs that depend on complex JavaScript logic, asynchronous execution, event-driven interactions, or other dynamic behaviors. We adopted this controlled input generation strategy to reduce crashes and nondeterministic behaviors, allowing observed differences to be more reliably attributed to rendering inconsistencies.

Threshold Selection. The number of detected rendering inconsistencies may depend on the perceptual-hash threshold [47] used by our oracle. We adopted the threshold values used in prior rendering-bug detection approaches, Metamong [35] and R2Z2 [34], rather than selecting them arbitrarily. However, we did not independently calibrate the threshold or conduct a sensitivity analysis for our inputs and experimental environments. Therefore, different threshold values may affect the reported detection results.

8 Related Work

Browser Rendering Bugs. There has been a line of research on identifying browser rendering bugs [34, 35, 50]. Due to the difficulty of defining precise conditions that characterize semantic bugs, prior work has primarily relied on differential testing to detect visual inconsistencies. R2Z2 [34] combines cross-browser and cross-version testing to detect regression rendering bugs. Metamong [35] targets render-update bugs by comparing the rendering of an initial page with that of the updated page after dynamic DOM modifications. On the other hand, Janus [50] introduces visual delta consistency oracle, assuming that identical JavaScript should produce consistent rendering across different browsers. Different from them, ICHECK aims to identify iframe rendering bugs by introducing a novel cross-context oracle that compares the rendering results of identical HTML in top-level and iframe contexts.

Yang *et al.* [45] studied security flaws of iframes/pop-ups in Android WebView that arise from mismatches between browser and WebView environments. However, this work focused on app-level security flaws rooted in UI, APIs, and policies of WebView, rather than on incorrect browser rendering.

Delta Debugging that Supports HTML input format. As discussed in §2.3, delta debugging [48] relies on predefined search strategies to optimize input traversal. The original algorithm, *ddmin* [48], has been extended to improve efficiency and reduction quality. *ProbDD* [40] guides the search process using probabilistic models, and a later variant, *WDD* [52], further improves performance by considering different weights of input elements during partitioning.

Some existing work [25, 36] extends delta debugging to structured input formats such as XML and HTML by leveraging structural information (e.g., DOM trees [25] or grammars [36]). Hierarchical delta debugging (HDD [25]) operates on tree representations (e.g., DOM trees or AST) and removes subtrees while preserving structural validity. Later, incremental variants of HDD have been proposed to reduce the input in coarse granularity [16] and improve the quality of minimized inputs [19]. *Perses* [36] is a grammar-based input reduction technique that uses context-free grammars to ensure that reduced inputs remain syntactically valid throughout the reduction process.

However, different from these delta debugging approaches, our work focuses on redefining the effective search space rather than optimizing how it is explored. While prior techniques traverse the full input space using predefined strategies, our approach leverages rendering semantics to map visual inconsistencies to DOM elements and prune irrelevant parts of the input, thereby reducing unnecessary rendering executions.

Browser Layout Testing and Verification. A line of research has focused on *layout testing* by comparing how different browsers render the same HTML input, particularly in the real-world web applications. These approaches aim to detect cross-browser compatibility issues by identifying visual inconsistency across browsers [6, 7, 22, 32]. Most approaches employ image comparison [6, 7, 22], computer vision [22], or machine learning [32] techniques to highlight rendering differences. *WEBDIFF* [7] detects inconsistencies using pixel- and histogram-based screenshot comparison. *CrossCheck* [6] augments screenshot differencing with large-scale crawling and machine learning. Mesbah and Prasad [22] automate cross-browser testing by executing web applications and comparing their rendered screenshots. *Browserbite* [32] applies image segmentation and feature extraction with machine learning to classify genuine incompatibilities. Different from these techniques, our work does not focus on cross-browser compatibility, but targets iframe-specific rendering bugs that arise even within the same browser.

Another line of work explores *layout verification* by formalizing browser rendering behavior and checking whether a given layout conforms to expected visual constraints [23, 29, 30]. These methods employ constraint solvers to validate layout correctness against formal specifications. Meyerovich and Bodik [23] encode CSS layout computation as grammar-based attribute constraints and solve them in parallel. *Cassius* [30] formalizes CSS rules as arithmetic constraints in a domain-specific language and verifies them with SMT solvers. *VizAssert* [29] expresses accessibility requirements as logical constraints over CSS rendering and checks them using

SMT-based verification. Unlike these prior approaches, our work does not verify layouts against formal specifications. Instead, our work focuses on detecting and debugging bugs that manifest as unintended visual inconsistencies.

DOM Fuzzing. There has been research on DOM fuzzing to identify memory-level bugs [11, 14, 27, 28, 33, 43]. *Domato* [14], *Dharma* [27], *Avalanche* [28], and *Wadi* [33] rely on handwritten context-free grammars to generate and mutate HTML files. *FreeDOM* [43] leverages a context-aware intermediate representation to generate syntactically and semantically valid test inputs. To generate more precise test inputs, *SaGe* [49] extracts grammars from web standards and infers semantic correctness by monitoring runtime semantic errors. *Minerva* [51] detects browser API bugs by revealing inter-API relations and synthesizing test cases guided by these relations.

The HTML inputs generated by prior approaches are structurally complex and involve diverse layout interactions, and are therefore effective in triggering rendering bugs. In our study, we leverage *Domato* [14] in combination with a cross-context oracle to detect iframe rendering bugs.

9 Conclusion

This paper presents *ICHECK*, an automated testing technique for detecting and debugging iframe-specific rendering bugs in web browsers. We identified a key limitation of existing techniques—their reliance on same-context comparisons—which prevents them from capturing inconsistencies across rendering contexts. To address this gap, we introduced a cross-context differential oracle that compares identical HTML content in top-level and iframe contexts, and a layout-aware input reduction technique that leverages rendering semantics to redefine the effective search space. By mapping visual inconsistencies to corresponding DOM elements, our approach prunes irrelevant input and significantly reduces the number of expensive rendering operations required during debugging.

Evaluation results demonstrate the effectiveness of *ICHECK* across multiple browsers. *ICHECK* detects up to 3.8× more bugs than state-of-the-art baselines and detected 37 bugs (16 previously known and 21 previously unknown), including a confirmed CVE with security implications. Additionally, *ICHECK* achieves 93% reduction in DOM nodes on average, while achieving up to 10.1× faster reduction compared with the state-of-the-art approaches. These results show that *ICHECK* not only improves the precision of rendering bug detection but also serves as a practical debugging aid.

10 Data Availability

The data supporting this study are openly available at <https://doi.org/10.6084/m9.figshare.33112562.v1> [8].

Acknowledgements

The authors thank the anonymous reviewers for their insightful comments and suggestions. This work was supported by the National Research Foundation (NRF) (No. RS-2026-25497375, RS-2025-00561150) and the Institute for Information & Communications Technology Planning & Evaluation (IITP) (No. RS-2024-00337414, RS-2020-II201336, RS-2026-25616928, RS-2024-00437306, RS-2023-00259967, RS-2020-II201819), all funded by the Ministry of Science and ICT (MSIT) of South Korea.

References

- [1] Adam Barth, Collin Jackson, and John C Mitchell. 2009. Securing frame communication in browsers. *Commun. ACM* 52, 6 (2009), 83–91. doi:10.1145/1516046.1516066
- [2] Johannes Buchner. 2017. imagehash. <https://github.com/JohannesBuchner/imagehash>. Accessed: 2025-09-10.
- [3] Builtwith. 2025. IFrame usage statistics. <https://trends.builtwith.com/docinfo/IFrame>.
- [4] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep Differential Testing of JVM Implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1257–1268. doi:10.1109/ICSE.2019.00127
- [5] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) (PLDI '16). Association for Computing Machinery, New York, NY, USA, 85–99. doi:10.1145/2908080.2908095
- [6] Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. 2012. Cross-Check: 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*. 171–180. doi:10.1109/ICST.2012.97
- [7] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. 2010. WEBDIFF: Automated identification of cross-browser issues in web applications. In *2010 IEEE International Conference on Software Maintenance*. 1–10. doi:10.1109/ICSM.2010.5609723
- [8] Data Availability Accessed: 2026. Testing IFrame Rendering in Web Browsers: Replication Package and Experimental Results. doi:10.6084/m9.figshare.33112562.v1
- [9] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. 2022. Fuzzing deep-learning libraries via automated relational API inference. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) (ESEC/FSE 2022). Association for Computing Machinery, New York, NY, USA, 44–56. doi:10.1145/3540250.3549085
- [10] Zizhuang Deng, Guozhu Meng, Kai Chen, Tong Liu, Lu Xiang, and Chunyang Chen. 2023. Differential testing of cross deep learning framework APIs: revealing inconsistencies and vulnerabilities. In *Proceedings of the 32nd USENIX Conference on Security Symposium* (Anaheim, CA, USA) (SEC '23). USENIX Association, USA, Article 414, 18 pages.
- [11] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. 2021. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases.. In *NDSS*. doi:10.14722/ndss.2021.24224
- [12] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 93 (Oct. 2017), 29 pages. doi:10.1145/3133917
- [13] Gerjant Franken, Tom Van Goethem, Lieven Desmet, and Wouter Joosen. 2023. A bug's life: analyzing the lifecycle and mitigation process of content security policy bugs. In *Proceedings of the 32nd USENIX Conference on Security Symposium* (Anaheim, CA, USA) (SEC '23). USENIX Association, USA, Article 206, 18 pages.
- [14] Google. [n.d.]. Domato. <https://github.com/googleprojectzero/domato>.
- [15] Google / Chromium Project 2025. *Chromium Issue Tracker*. Google / Chromium Project. <https://issues.chromium.org/>. Accessed: 2025-09-10.
- [16] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2017. Coarse Hierarchical Delta Debugging. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 194–203. doi:10.1109/ICSME.2017.26
- [17] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. 2021. DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1286–1303. doi:10.1109/SP40001.2021.00103
- [18] Timotej Kapus and Cristian Cadar. 2017. Automatic testing of symbolic execution engines via program generation and differential testing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 590–600. doi:10.1109/ASE.2017.8115669
- [19] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. 2018. HDDR: a recursive variant of the hierarchical Delta debugging algorithm (*A-TEST 2018*). Association for Computing Machinery, New York, NY, USA, 16–22. doi:10.1145/3278186.3278189
- [20] Jiawei Liu, Jinkun Lin, Fabian Ruff, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) (ASPLOS 2023). Association for Computing Machinery, New York, NY, USA, 530–543. doi:10.1145/3575693.3575707
- [21] Google LLC. 2025. Google Ads. <https://ads.google.com/>
- [22] Ali Mesbah and Mukul R. Prasad. 2011. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) (ICSE '11). Association for Computing Machinery, New York, NY, USA, 561–570. doi:10.1145/1985793.1985870
- [23] Leo A. Meyerovich and Rastislav Bodik. 2010. Fast and parallel webpage layout. In *Proceedings of the 19th International Conference on World Wide Web* (Raleigh, North Carolina, USA) (WWW '10). Association for Computing Machinery, New York, NY, USA, 711–720. doi:10.1145/1772690.1772763
- [24] Microsoft Playwright Team. 2025. *Playwright: Fast and reliable end-to-end testing for modern web apps*. Microsoft. <https://playwright.dev/>. Accessed: 2025-09-10.
- [25] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) (ICSE '06). Association for Computing Machinery, New York, NY, USA, 142–151. doi:10.1145/1134285.1134307
- [26] Mozilla Foundation 2025. *Mozilla Bugzilla Issue Tracker*. Mozilla Foundation. <https://bugzilla.mozilla.org/>. Accessed: 2025-09-10.
- [27] Mozilla Security Team. 2015. Dharma. <https://github.com/MozillaSecurity/dharma>. Accessed 2025.
- [28] Mozilla Security Team. 2019. Avalanche. <https://github.com/MozillaSecurity/avalanche>. Accessed 2025.
- [29] Pavel Panchekha, Adam T. Geller, Michael D. Ernst, Zachary Tatlock, and Shoaib Kamil. 2018. Verifying that web pages have accessible layout. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) (PLDI 2018). Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3192366.3192407
- [30] Pavel Panchekha and Emina Torlak. 2016. Automated reasoning for web page layout. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) (OOPSLA 2016). Association for Computing Machinery, New York, NY, USA, 181–194. doi:10.1145/2983990.2984010
- [31] Antu Saha and Oscar Chaparro. 2025. Decoding the Issue Resolution Process in Practice via Issue Report Analysis: A Case Study of Firefox. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*. IEEE Press, 2316–2328. doi:10.1109/ICSE55347.2025.00241
- [32] Natalia Semenchenko, Marlon Dumas, and Tönis Saar. 2013. Browserbite: Accurate Cross-Browser Testing via Machine Learning over Image Features. In *2013 IEEE International Conference on Software Maintenance*. 528–531. doi:10.1109/ICSM.2013.88
- [33] SensePost. 2015. Wadi. <https://github.com/sensepost/wadi>. Accessed 2025.
- [34] Suhwan Song, Jaewon Hur, Sunwoo Kim, Philip Rogers, and Byoungyoung Lee. 2022. R2Z2: Detecting Rendering Regressions in Web Browsers through Differential Fuzz Testing. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. ACM, 1818–1829. doi:10.1145/3510003.3510044
- [35] 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)*. ACM, TBD. doi:10.1145/3611643.3616336
- [36] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perse: syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) (ICSE '18). Association for Computing Machinery, New York, NY, USA, 361–371. doi:10.1145/3180155.3180236
- [37] The CVE Program / MITRE Corporation 2025. *CVE: Common Vulnerabilities and Exposures*. The CVE Program / MITRE Corporation. <https://www.cve.org/>. Accessed: 2025-09-10.
- [38] Anne van Kesteren, Domenic Denicola, Ian Hickson, Simon Pieters, and Philip Jägenstedt. 2024. HTML Living Standard. <https://html.spec.whatwg.org/>. Accessed: [Insert Access Date].
- [39] w3schools. 2025. Browser Statistics. <https://www.w3schools.com/browsers/>
- [40] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. 2021. Probabilistic Delta debugging (*ESEC/FSE 2021*). Association for Computing Machinery, New York, NY, USA, 881–892. doi:10.1145/3468264.3468625
- [41] WHATWG. 2025. HTML Living Standard: Top-level browsing contexts. <https://html.spec.whatwg.org/multipage/iframe-embed-object.html>.
- [42] World Wide Web Consortium (W3C). 2011. CSS 2.1 Specification: Visual formatting model. <https://www.w3.org/TR/CSS2/visuren.html#viewport>. Section 9.1.1: Viewport and canvas.
- [43] Wen Xu, Soyeon Park, and Taesoo Kim. 2020. FREEDOM: Engineering a State-of-the-Art DOM Fuzzer. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) (CCS '20). Association for Computing Machinery, New York, NY, USA, 971–986. doi:10.1145/3372297.3423340
- [44] Chenyuan Yang, Yinlin Deng, Jiayi Yao, Yuxing Tu, Hanchi Li, and Lingming Zhang. 2023. Fuzzing Automatic Differentiation in Deep-Learning Libraries. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) (ICSE '23). IEEE Press, 1174–1186. doi:10.1109/ICSE48619.2023.00105
- [45] Guangliang Yang, Jeff Huang, and Guofei Gu. 2019. Iframes/Popups Are Dangerous in Mobile WebView: Studying and Mitigating Differential Context Vulnerabilities. In *28th USENIX Security Symposium* (USENIX Security 19). 977–994.
- [46] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference*

- on *Programming Language Design and Implementation* (San Jose, California, USA) (PLDI '11). Association for Computing Machinery, New York, NY, USA, 283–294. doi:10.1145/1993498.1993532
- [47] Christoph Zauner. 2010. Implementation and benchmarking of perceptual image hash functions. (2010).
 - [48] A. Zeller and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200. doi:10.1109/32.988498
 - [49] Chijin Zhou, Quan Zhang, Lihua Guo, Mingzhe Wang, Yu Jiang, Qing Liao, Zhiyong Wu, Shanshan Li, and Bin Gu. 2023. Towards Better Semantics Exploration for Browser Fuzzing. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 243 (Oct. 2023), 28 pages. doi:10.1145/3622819
 - [50] Chijin Zhou, Quan Zhang, Bingzhou Qian, and Yu Jiang. 2025. Janus: Detecting Rendering Bugs in Web Browsers via Visual Delta Consistency. In *Proceedings of the 47th International Conference on Software Engineering (ICSE)*. ACM. doi:10.1109/ICSE55347.2025.00013 To appear.
 - [51] Chijin Zhou, Quan Zhang, Mingzhe Wang, Lihua Guo, Jie Liang, Zhe Liu, Mathias Payer, and Yu Jiang. 2022. Minerva: browser API fuzzing with dynamic mod-ref analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) (ESEC/FSE 2022). Association for Computing Machinery, New York, NY, USA, 1135–1147. doi:10.1145/3540250.3549107
 - [52] Xintong Zhou, Zhenyang Xu, Mengxiao Zhang, Yongqiang Tian, and Chengnian Sun. 2025. WDD: Weighted Delta Debugging. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) (ICSE '25). IEEE Press, 1592–1603. doi:10.1109/ICSE55347.2025.00071

Received 2026-03-27; accepted 2026-06-18