Visual-Aware Testing and Debugging for Web Performance Optimization

Xinlei Yang[1*], Wei Liu[1*], Hao Lin[1], Zhenhua Li[1]
Feng Qian[2], Xianlong Wang[1], Yunhao Liu[1], Tianyin Xu[3]
[1]Tsinghua University, China [2]University of Minnesota, USA [3]UIUC, USA

ABSTRACT

Web performance optimization services, or web performance optimizers (WPOs), play a critical role in today's web ecosystem by improving page load speed and saving network traffic. However, WPOs are known for introducing *visual distortions* that disrupt the users' web experience. Unfortunately, visual distortions are hard to analyze, test, and debug, due to their subjective measure, dynamic content, and sophisticated WPO implementations.

This paper presents Vetter, a novel and effective system that automatically tests and debugs visual distortions. Its key idea is to reason about the *morphology* of web pages, which describes the topological forms and scale-free geometrical structures of visual elements. Vetter efficiently calculates morphology and comparatively analyzes the morphologies of web pages before and after a WPO, which acts as a differential test oracle. Such morphology analysis enables Vetter to detect visual distortions accurately and reliably. Vetter further diagnoses the detected visual distortions to pinpoint the root causes in WPOs' source code. This is achieved by *morphological causal inference*, which localizes the offending visual elements that trigger the distortion and maps them to the corresponding code. We applied Vetter to four representative WPOs. Vetter discovers 21 unknown defects responsible for 98% visual distortions; 12 of them have been confirmed and 5 have been fixed.

CCS CONCEPTS

• **Information systems → Web interfaces**; • **Software and its engineering → Software testing and debugging**.

KEYWORDS

Web Performance Optimization; Web Page Distortion; Visual-Aware Testing and Debugging

ACM Reference Format:
Xinlei Yang, Wei Liu, Hao Lin, Zhenhua Li, Feng Qian, Xianlong Wang, Yunhao Liu, Tianyin Xu. 2023. Visual-Aware Testing and Debugging for Web Performance Optimization. In *Proceedings of the ACM Web Conference 2023 (WWW '23), May 1–5, 2023, Austin, TX, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3543507.3583323

[*]Co-primary authors. Zhenhua Li is the corresponding author.

WWW '23, May 1–5, 2023, Austin, TX, USA

ACM ISBN 978-1-4503-9416-1/23/04.
https://doi.org/10.1145/3543507.3583323

1 INTRODUCTION

The ever-growing demand for fast, reliable, and resource-efficient web browsing has been driving the active development of web performance optimization services [3, 23, 41, 47, 59], or web performance optimizers (WPOs). Deployed as server/client-side plug-ins or WAN/LAN proxies by mobile ISPs, content providers, and corporations, WPOs automatically perform a wide range of optimizations (*e.g.,* image transcoding, JavaScript/CSS minification, and HTML/-text compression) to save the page load time (PLT) and/or network traffic. Many WPOs also offer add-on services like caching, web security, and advertisement filtering.

WPOs are highly effective and popular. Google AMP has speeded up the loading process of 5 billion web pages by 2.5× on average [23, 45]. Google Flywheel [3] and Baidu TrafficGuard [47] save mobile traffic for tens of millions of users by over 1/2 and 1/3, respectively. New WPOs such as Fawkes [51], SipLoader [48], and Vroom [66] can further reduce user-perceived page load time significantly. Besides, selective traffic manipulation [49, 56] and symbolic execution [46] have been proposed to prevent privacy violations.

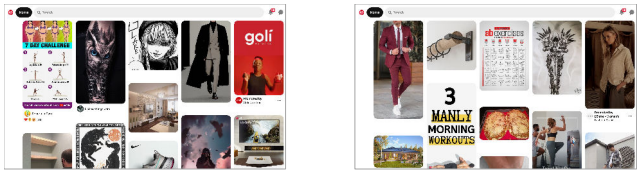
Despite the many compelling features and measurable benefits, WPOs could induce *visual distortions*, which significantly impair users' web experience [17, 21, 22, 70, 71]. Figure 1 shows an example of visual distortions. Compared to the original landing page of Bild.de (a popular German news media), the "optimized" version by Ziproxy [41] (a classic WPO for traffic compression) is severely distorted, leading to unacceptable user experience. In fact, many users choose to opt out of WPOs simply because of visual distortions [18, 30, 36]. This is particularly common for non-technical users who cannot explain the distortions [8, 20]. In addition, our interviews with developers of both Google Flywheel and Baidu TrafficGuard confirm that visual distortions are a key challenge of their services and have created major obstacles to adoption.

Unfortunately, visual distortions incurred by WPOs are under investigated. The knowledge gap hinders the development of practical solutions for detecting, debugging, and fixing visual distortions. To fill this gap, we conduct the first in-depth study on visual distortions introduced by WPOs. With informed consent and a well-established IRB, we recruit 18 users (from different age and gender groups) to help recognize visual distortions on the landing pages of the Alexa top and bottom 2,500 websites (among the top 1M) after being optimized by two widely used WPOs (*i.e.,* Ziproxy and Compy [42]). The experiments reveal three major findings.

- Although visual distortions seem to be subjective, for most (93%) web pages, the inspectors have the same opinions.
- Ziproxy and Compy incur visual distortions to 3.3% and 6.1% of the 5,000 landing pages respectively, which is significant enough to affect user experience and discourage WPO deployments.

(a) Original Page (b) Optimized Page

Figure 1: A real-world example of web page distortion.



(a) Page A (b) Page B

Figure 2: The landing page of Pinterest.com varies significantly in two different loads due to the dynamic contents.

- Visual distortions are more severe (5.6% for Ziproxy and 9.0% for Compy) in landing pages of less popular (the bottom 2,500) websites, as these websites include more non-standard or even incorrect web contents that confuse WPOs' optimizations.

Driven by the prevalence and severity of distortions, we aim to develop an effective approach to systematically address the issue. However, our experience reveals significant challenges. First, automatic detection of distortions is nontrivial. An intuitive approach is to directly compare the image snapshots of the original and optimized pages, which however cannot address *dynamic contents* that vary significantly (exemplified in Figure 2). An alternative is to compare the key data structures (*e.g.,* DOM, render, and CSSOM trees) of web pages, which also falls short for a lack of visual hints, missing layout information, and being over-general. Worse still, even when distortions are successfully detected, it is hard to locate their root causes in the source code, as WPOs typically have sophisticated implementations for accommodating complex resources (*e.g.,* images and videos), various protocols (*e.g.,* HTTP/3 and HTTPS), and diverse languages (*e.g.,* HTML5 and WebAssembly).

In this paper, we present Vetter, a system for automatically testing and debugging visual distortions from the perspective of how modern web pages are generated. Today, very few web pages are written from scratch in an ad-hoc manner; instead, most web pages are programmatically generated by several mainstream web frameworks (*e.g.,* Angular [25], React [35], and Vue [73]). These frameworks follow the standard Model-View-ViewModel (MVVM) design pattern that decouples a web page's layout (View) from the logic and data (Model). As a result, while the logic and data of a dynamic web page vary between different loads, only the scales or concrete contents of visual elements are changed accordingly; visual elements' topological forms and scale-free geometrical structures do not. Such invariants are referred to as a web page's *morphology*.

Vetter is built based on a key insight that visual distortions on a web page (no matter how dynamic it is) are highly relevant to changes in its morphology. As shown in Figure 2, in two different loads of Pinterest's landing page, pictures are both placed in rectangles (topological form) and aligned vertically (scale-free geometrical structure) despite significant pixel-level changes, so the optimized page was never deemed as visually distorted by the users.

However, comparing the morphologies of two web pages is costly—the time complexity is $O(n!)$ for web pages containing n visual elements. To address this, we discover the intrinsic *hierarchy* in a web page's morphology, which derives from the nested properties of the markup languages (*e.g.,* HTML and XML) used by mainstream web frameworks to define layouts. Vetter utilizes this to minify a web page's morphology, thus greatly reducing the complexity to $O(n^3)$, which only takes tens of milliseconds in practice.

Vetter further provides debugging support for visual distortions that are automatically detected (or manually identified). We find that solely analyzing the visually distorted elements (detected during testing) is ineffective, because many of them are the results of the "chain reactions" of neighboring visually distorted elements instead of the defects in a WPO's source code.

To debug visual distortions effectively, Vetter performs *morphological causal inference* that strategically manipulates the loading process of these elements (*e.g.,* replacing one visually distorted element's resource with its original one, or changing the loading sequence) under the guidance of morphological hints (*i.e.,* the graphical and structural information of visually distorted elements), and meanwhile checks whether the distortion can be resolved. Given the results, Vetter can efficiently infer the causal relationship between the WPO's source code and visually distorted elements, thus greatly and safely reducing the search space for debugging.

We apply Vetter to four representative WPOs (Ziproxy, Compy, Fawkes, and SipLoader). When using the Alexa top 2,500 websites as the training set and bottom 2,500 websites (among the top 1M) as the test set, Vetter can efficiently (costing 62 ms per page on average) detect visual distortions with high precision (95%) and recall (91%). Further, Vetter finds a total of 21 previously unknown defects in the four WPOs, most of which stem from 1) WPO developers' undue reliance on the correctness of the original web request/response headers, 2) the WPO-amplified dependency violations among web contents, and 3) the lack of support for emerging web techniques. We fix them through either source code correction or lightweight middleware pre-parsing, and thus clear up almost all (98%) of the visual distortions, which are more than detected (91%) as many undetected ones have been automatically resolved by our fixes.

All the 21 discovered defects along with our suggested fixes have been reported to the developers of the four WPOs, among which 12 defects have been confirmed and 5 suggested fixes have been officially adopted. The remaining ones are either under beta tests or under code review.

In summary, this paper makes the following contributions.

- We conduct the first study on visual distortions incurred by WPOs and release a large open dataset involving 5,000 websites.
- We develop Vetter, a visual-aware testing and debugging system that automatically and effectively detects and debugs visual distortions based on the morphology of web pages.
- Vetter has detected 21 unknown defects in four widely used WPOs; to date, 12 of them are confirmed and 5 are fixed.
- The code and data involved in this work are released at https://github.com/Web-Distortion/Vetter (detailed in Appendix A.5).

2 BACKGROUND AND MOTIVATION

2.1 The Dilemma of WPOs

In the web ecosystem, WPOs are extensively developed and deployed as client-side plug-ins, server-side middleware, or standalone proxies, by mobile ISPs [32, 65] and content providers [3, 47], to save page load time and network traffic, to provide versatile add-on services like web content caching [3], encrypted network communication [46], and advertisement filtering [63].

A WPO can help optimize the web browsing experience before (*offline* phase) and/or during (*online* phase) user access. In the offline phase, the WPO typically pre-loads the web page to analyze the included resources, and then executes corresponding optimization routines, *e.g.,* transcoding/compressing images, re-organizing the resource loading sequence [48, 66], and rewriting HTML files [48, 51]. Afterwards, the optimized page is usually cached on the web server for serving later accesses. In addition, some WPOs also perform online optimizations during web page loading. They typically run on a standalone proxy for simpler or lighter tasks, *e.g.,* resource pre-fetching [66], TCP pre-connection [3], and advertisement blocking [47]. This is because "heavy" optimizations like analyzing and rewriting HTML files could induce non-negligible latency penalty.

Despite the benefits, WPOs have received plenty of user complaints when working in the wild due to the side effects. By extensively reviewing the negative comments posted on relevant online communities [70, 71] and customer-support websites [10, 17, 22], we find the side effects include latency penalties, bandwidth bottleneck [75], functional anomalies, visual distortions, and so forth. In particular, most users are concerned about visual distortions, such as layout displacement and content loss on the optimized web pages [21, 22, 70, 71]. Worse still, some users even doubt that the WPO has incurred security/privacy threats such as unauthorized advertisement injection [8] and undesired web page redirection [20].

2.2 Understanding Visual Distortions

To quantify the realistic prevalence and severity of visual distortions incurred by WPOs, we apply two typical WPOs, *i.e.,* Ziproxy (a classic HTTP forwarding proxy for traffic compression and acceleration) and Compy (an open-source implementation of Google Flywheel) to the Alexa top and bottom 2,500 (thus a total of 5,000) websites (among the Alexa top 1M) on Dec. 9th, 2021. Specifically, we deploy the latest versions of Ziproxy and Compy on two separate VM servers rented from AWS EC2, each with a dual-core CPU @2.3 GHz, 2 GB of memory, and 100 Mbps access bandwidth. Also, we develop an automated web browsing exerciser (or simply exerciser) and let it work on a typical Windows-10 PC with a quad-core CPU @3.4 GHz, 16 GB of memory, and 100 Mbps access bandwidth.

For each of the 5,000 websites, the exerciser sequentially visits its landing page with and without the two deployed WPOs using Chrome v79 web browser, respectively. Once the page is fully loaded (with cold browser cache every time), the exerciser takes a snapshot of the screen display. As a result, we obtain three screenshots for each website—one for the original landing page and the other two for the optimized versions (produced by Ziproxy and Compy).

With the screenshots of the 5,000 websites' landing pages, we next determine whether these optimized pages are visually distorted. Here a challenge is that there is *subjectivity*, to some extent, in users' perceptions of visual distortions. To deeply understand visual distortions from real users' perspectives, we conduct a crowdsourcing study by recruiting 18 users with different ages and genders. They help recognize and categorize visual distortions on the 5,000 optimized landing pages produced by the two WPOs. More details of the crowdsourcing study are described in Appendix A.1.

Table 1: Visual distortions occurring to the Alexa top 2,500 websites' landing pages. "#" is the number of distorted pages.

Distortion Symptom	# by Ziproxy	# by Compy	Total
Content Loss	0	63	63
Image Display Error	11	0	11
Text Confusion	13	16	29
Layout Disorder	0	3	3
All	24	82	106

Measurement Findings. Based on the collected dataset, we have multi-fold findings on WPO-incurred visual distortions in terms of their prevalence, severity, and key characteristics.

First, although visual distortions seem to be subjective problems, in most cases (93%) the recruited users can in fact reach a consensus. For the remainder (7%), visual distortions are determined by majority voting; if there is a tie, we would participate to break it.

Besides, both Ziproxy and Compy bring visual distortions to a nontrivial portion (0.96% and 3.28%) of the top 2,500 websites' landing pages. We list the specific symptoms and their quantities in Table 1 based on the opt-in users' feedback. We find that Ziproxy and Compy can vary greatly regarding the occurrence of a certain symptom (we delay the detailed explanation to §4.2). While the portions both look small (<10%), note that even a single visual distortion can incur a direct, negative impact on the user experience, making users unsatisfied or vigilant and thus stop using the WPO. Also note that we only test the landing page of each website, which typically contains quite a number of pages. That is, there might well be much more distortions undiscovered for the 5,000 websites.

By contrast, the bottom 2,500 websites' landing pages suffer even more visual distortions: 5.64% for Ziproxy and 8.96% for Compy. This is understandable: less popular websites usually include more non-standard or even incorrect web contents that can more easily trigger the side effects of WPOs' optimization routines. Our detailed analysis in §4.1 also confirms this.

The study was performed under a well-established IRB. No personally identifiable information was collected.

2.3 Challenges

To combat visual distortions induced by WPOs, we have explored several potential solutions to detecting and debugging them. Our experience reveals significant technical challenges to automatically detecting and debugging visual distortions using traditional metrics.

Specifically, to detect visual distortions, an intuitive approach is using computer vision (CV) metrics to compare the rendering results of the original and optimized pages. Unfortunately, this approach works poorly on dynamic pages whose rendering results differ from one load to another (detailed in Appendix A.2).

Comparing the key data structures of web pages is also ineffective. For example, a web page's render tree which contains visible web elements is inaccessible outside the browser. The CSS Object

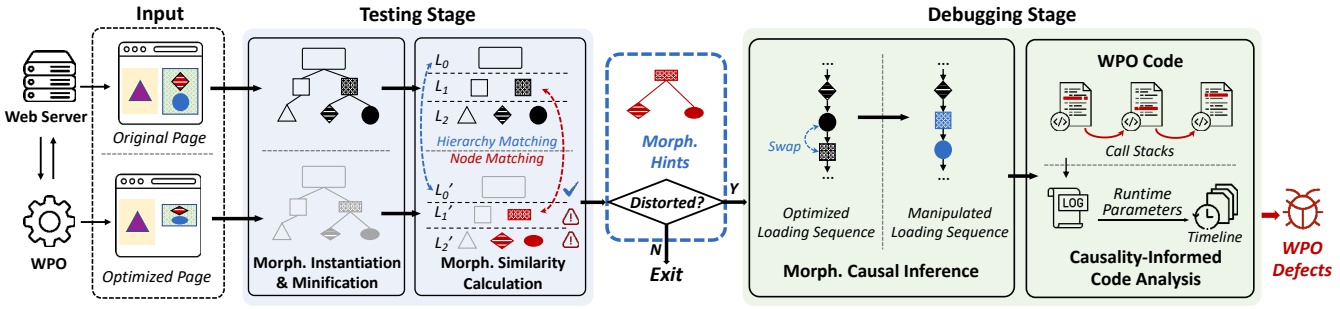


Figure 3: Architectural overview of Vetter.

Model (CSSOM) tree is over general as it only describes the presentation or formatting style of web elements. The Document Object Model (DOM) tree only represents the logic (rather than visual) relations of web elements; it can be changed greatly by inserting multiple empty inner elements (*e.g.,* `<div>`) into the HTML file, yet the rendering result remains unchanged.

Worse still, even if we managed to detect visual distortions, it is still challenging to debug them as WPOs often have sophisticated implementations, *e.g.,* Ziproxy has ~20K lines of code and employs 40+ third-party modules and auto-generated functions. Meanwhile, the available information for diagnosis is very limited. The runtime logs of WPOs crucial to debugging are often hard to fetch, especially for those WPOs that only work in the offline mode (as their optimizations are performed ahead of the page loading process).

3 DESIGN AND IMPLEMENTATION

3.1 System Overview

To address the challenges in §2.3, we present Vetter, a system for automatically testing and debugging visual distortions.

The main idea is to rethink about visual distortions from the perspective of how modern web pages are generated. We observe that most of the modern web pages are programmatically generated by mainstream web frameworks (*e.g.,* Angular [25], React [35], and Vue [73]), rather than manually written from scratch in an ad-hoc manner. All these frameworks adopt the standard Model-View-ViewModel (MVVM) design pattern that separates a web page's layout (View) from its logic and data (Model), so as to make the web page easy to develop, test, and maintain. Consequently, while the logic and data of a dynamic web page vary between different loads, only the scales or concrete contents of the visual elements are changed accordingly. On the other hand, the visual elements' topological forms and scale-free geometrical structures scarcely ever change; such invariants are termed a web page's *morphology*.

Morphology provides an effective vantage point to understand visual distortions—visual distortions occurring to a web page (no matter how dynamic it is) are highly relevant to changes in its morphology. This conforms to our daily web browsing experience—for a dynamic web page (like Pinterest.com), a user does not expect or can even notice the changes of individual visual elements between different page loads. However, if its morphology changes drastically, the user would easily notice and feel uncomfortable. In theory, it is possible for a web page to swap its own morphology upon different loads, but such a behavior is rare in practice.

Vetter is built on the morphology insight. In order to detect visual distortions, Vetter extracts and compares the morphologies of web pages; further, Vetter uses the morphological differences between the pages as important hints to pinpoint the root causes. Figure 3 depicts Vetter's major components and workflow. Vetter takes the original web page and its optimized version generated by a WPO as the input, and performs the following testing and debugging steps:

- *Morphology Instantiation and Minification* (§3.2). To instantiate a web page's morphology, we employ *scene graph*, a classic data structure in computer graphics for representing 2D/3D scenes [19], to capture both the abstract graphical information and spatial structures among the web page's constituent objects. However, comparing the scene graphs of two web pages incurs prohibitively high computational overhead. To address this, Vetter leverages the intrinsic hierarchy in a web page's morphology to minify the scene graph into a *morphological segment tree* (MST), so as to facilitate the remaining steps.
- *Morphological Similarity Calculation* (§3.3). Once two pages' MSTs are constructed, Vetter calculates their morphological similarity in two stages to determine visual distortions. First, Vetter adopts *hierarchy matching* enhanced by *memorization algorithms* to quickly perform coarse-grained, level-by-level matching between two MSTs. Second, it zooms in on each matched level to perform fine-grained, node-by-node matching. In this way, the morphological similarity between two pages can be calculated with $O(n^3)$ time complexity while retaining high accuracy. Meanwhile, Vetter records the graphical and structural information of visually distorted elements (marked red in Figure 3) as morphological hints to inform debugging.
- *Morphological Causal Inference* (§3.4). When diagnosing a visual distortion, we find that focusing on all the visually distorted elements (detected in the above testing step) is expensive and unnecessary, as many of them are just the "chain reaction" results of neighbouring visually distorted elements. Therefore, Vetter strategically manipulates the loading process of these elements (*e.g.,* changing the loading sequence as depicted in Figure 3) using the morphological hints; meanwhile, it invokes the testing steps again to check if the distortion still exists after the manipulation. By repeating the above-described trials until the distortion is resolved, Vetter can infer the causal relationship between the distortion and the visually distorted elements, thus effectively and safely ruling out the distractions.

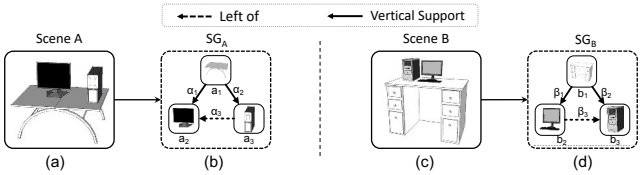


Figure 4: Two typical scenes and their scene graphs.

- *Causality-Informed Code Analysis* (§3.5). Having obtained a visual element (called a *critical element*) that directly causes the distortion, Vetter then extracts specific functions that have ever modified the parameters of the critical element at run time from the function call stacks recorded in the previous step. These functions are thus highly related to the distortion, but may still not be the problematic ones. To help WPO developers efficiently locate the problematic functions, Vetter further picks out the runtime parameters (and their values) of these functions from the WPO-recorded running logs, and organizes them into a timeline, from which developers can easily notice invalid parameters and undesirable call paths, thereby quickly pinpointing the root causes at the source code level.

3.2 Morphology Instantiation & Minification

Effectively instantiating the concept of morphology is vital to the design of Vetter. Through extensive literature review, we notice that *scene graph* [19], a widely-used data structure in computer graphics for representing graphic elements in a scene and their spatial relations, is an ideal choice. Figures 4(a) and (c) depict two scenes which both contain three elements—a monitor, a computer, and a desk. Their corresponding scene graphs are shown in Figures 4(b) and (d), where each node represents a certain element and each edge denotes a kind of structural relations (*e.g.,* "left of" and "vertical support"). Since scene graph can well represent the 3D graphic elements and their spatial relations, we believe that it is expressive enough to describe (2D) web page elements' topological forms and scale-free geometrical structures, *i.e.,* a web page's morphology.

Scene Graph Construction for Web Pages. To construct scene graphs for web pages, a critical problem is that the graphic elements and their structural relations on a web page are not given by the web server or client. Currently, there are mainly two types of solutions: 1) intuitive CV-based web page segmentation [9], and 2) underlying data structure-based page segmentation [11]. Sadly, the former is subject to the inaccuracy of pattern recognition, and the latter involves highly complicated rules that are not actionable in practice.

Thankfully, we note that when rendering a web page, mainstream web browsers such as Chrome, Firefox, and Safari all adopt the SkPaint utility [68] to draw graphic elements on the web page's canvas. These graphic elements correspond to all the web objects, and thus are ideal for scene graph construction. Besides, they are fully accessible to outsiders rather than only the web browser.

Further, by carefully analyzing the browser's SkPaint API invocation logs of the Alexa top and bottom 2,500 websites, we observe a highly skewed invocation pattern: nearly 99% of the invocations merely relate to 12 SkPaint APIs, among which only five (*i.e.,* `drawTextBlob`, `drawRect`, `drawPath`, `drawImageRect` and `drawRRect`) will add a visible graphic element (*i.e.,* text, image, rectangle, rounded rectangle, line, and customized shape) to the web page; the other seven (*i.e.,* `restore`, `save`, `saveLayer`, `concat`, `drawPaint`, `clipRect` and `clipRRect`) do not involve actual rendering operations and thus do not affect the web page's appearance.

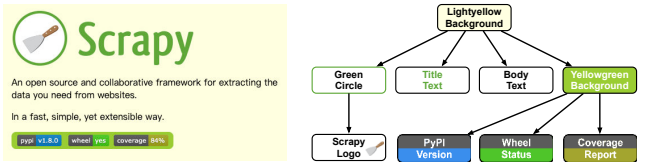


Figure 5: A typical web page and its corresponding morphological segmentation tree (MST).

After filtering out useless SkPaint API invocations, we can use the remainder to build the scene graph for a web page. Specifically, we extract graphic elements together with their major properties (*i.e.,* topological form) from the really useful SkPaint API invocations. Such graphic elements act as the nodes in the scene graph. Further, we need to construct edges that represent the structural relations between different nodes. In practice, there exist multiple structural relations including 1) containment, 2) intersection, 3) contact, 4) adjacency, 5) above/below, 6) left/right, 7) superposition, and so on. Unfortunately, considering all these relations would make the scene graph (*i.e.,* the morphology of the web page) overly complicated for efficient storage and subsequent processing.

Morphological Segmentation Tree (MST). To address this, we carefully study the visual structures of the Alexa top and bottom 2,500 websites, and observe that almost all their landing pages exhibit a certain form of *hierarchy* in their appearance. Take Figure 5(a) as an example, on the rendered web page lie a total of nine graphic elements, where the largest element (Lightyellow Background) contains all the other eight elements. Further, Green Circle contains Shovel Logo, and Yellowgreen Background contains PyPI Version, Wheel Status, and Coverage Report. In fact, the hierarchy among graphic elements is not an incidental phenomenon but a matter of course, recalling the nested properties of the markup languages (*e.g.,* HTML and XML) for defining web pages' layouts.

Guided by the above, we minify a page's scene graph into a *morphological segmentation tree* (MST) by focusing on the intrinsic *hierarchy* among graphic elements, which can be fully captured by the *containment* relation. By only considering this relation, we can naturally simplify the original graph into a tree structure, which is named as *morphological segmentation tree* (MST). For example, Figure 5(b) depicts the MST for the web page shown in Figure 5(a), where each node represents a graphic element and each edge denotes the containment relation between two elements.

Apparently, the above minification process has a caveat: if containment cannot fully represent the relations between web elements (involving 1.6% web pages in our dataset), false negatives may be induced in distortion detection. To address this, we also use the other structural relations together with the containment relation for fine-grained matching between web elements (*cf.* §3.3). In addition, there exist "infinite-scroll" web pages (*e.g.,* social media newsfeeds [2]) that seem to contain infinite contents and thus frustrate our constructing complete MSTs. Fortunately, we observe that such pages are in fact never loaded in one shot. For a typical infinite-scroll page, a small portion of contents are first loaded (the

initial page load), and then more contents are continuously loaded as users scroll down (subsequent content loads). Given this, we construct the MST of a web page based on the contents of the initial page load. Note that according to our measurement study, almost all (>98%) infinite-scroll pages have highly similar morphologies among different loads. Thus, it is almost always sufficient to detect distortions using MSTs constructed from the initial page load.

3.3 Morphological Similarity Calculation

With the constructed MSTs (denoted as MST_A and MST_B) of a web page A and the optimized page B, we next calculate their similarity to compare the pages' morphologies. Recall that a page's morphology refers to the visual elements' topological forms and scale-free geometrical structures. Thus, we should first match the visual elements in the two pages, *i.e.,* the nodes in MST_A and MST_B.

Hierarchy Matching & Node Matching. Perfectly matching the nodes in two MSTs is known to bear $O(n!)$ time complexity when there are n nodes in each MST, which is infeasible in practice. To address this, we first leverage MSTs' hierarchical information to perform coarse-grained, level-by-level *hierarchy matching* between MST_A and MST_B by comparing the structural relations among different nodes in the same level. Moreover, we use a memorization algorithm to accelerate the matching process.

Once a pair of levels in the two MSTs are matched, we then conduct fine-grained, node-by-node matching within the two levels. Specifically, we find the best matching scheme between two levels with the highest average similarity (calculated based on the topological forms and structural relations of the nodes) among all pairs of matched nodes. We use the classic Hungarian algorithm [6] to solve this problem with $O(k^3)$ time complexity, where each level contains k nodes. The algorithmic details are in Appendix A.3.

Morphological Similarity Calculation. Finally, after all the nodes are matched, we calculate the similarity between the two MSTs (termed as MorphSIM) as the average similarities between all pairs of matched nodes. Ideally, we can directly determine the occurrence of visual distortions by using MorphSIM. However, in practice we find that solely relying on MorphSIM brings a low (4.7%) false positive (FP) rate yet a median (20.3%) false negative (FN) rate (*cf.* §4.1), as the morphology-wise comparison is not sensitive to small pixel-level changes. By contrast, making joint use of the widely used CV metrics including SSIM [77], SIFT [50], and pHash [76] (*i.e.,* the CV-hybrid approach) often yields a median (16%) FP rate and a low (10%) FN rate (also *cf.* §4.1). Thus, the two methods in fact well complement each other. Given this, we make combinatory use of MorphSIM-based and CV-based method to decide whether there are distortions by using them as machine learning features for training various machine learning classifiers including Decision Tree, Random Forest [43], Logistic Regression, Naive Bayes, SVM, SGD-Classifier [14], and RBF Neural Network [5]. Finally, Random Forest excels with the average F1 score >93.0%.

3.4 Morphological Causality Inference

To help WPO developers effectively analyze a visual distortion, our idea is to leverage the extracted morphological information of the optimized web page as critical hints for determining the causal relationships [44] between a visually distorted element and the distortion, so as to rule out distractions and reduce the search space of problematic code. In general, since a WPO usually modifies a web page's resources and their loading sequences to achieve performance optimizations, we gradually restore the modified resources and sequences to the original ones to see whether the distortion is resolved. If so, the "real culprits" of the distortion are among the most recently restored resources/sequences.

Specifically, we first extract the resources (including HTML/JavaScript/CSS files, fonts, images, and videos) related to the visual distortion based on the browser execution logs we capture during the page's loading process. We track the CSS rules in the stylesheet resources that correspond to all the visually distorted elements; meanwhile, we record the call stack information of DOM-related JavaScript API invocations (*e.g.,* `appendChild`, `removeChild`, and `setAttribute`) that process the visually distorted elements. Based on the above information, we can find out all the suspicious resources that potentially incur the visual distortion.

In practice, we typically extract tens of (41 on average in our dataset) suspicious resources for a single visually-distorted web page, while only a small portion (4%) of them are the real culprits. To narrow down the search space, we gradually replace the optimized resources with their original versions (one at a time), and invoke the visual-aware testing steps (*cf.* §3.3) after each replacement to check if the distortion has been resolved. If so, we infer that the truly problematic resources that lead to the distortion are among the recently restored ones, without manipulating the remaining resources. Otherwise, we further resort to restoring the web page's resource loading sequence using the the classic sequence alignment algorithm [52], which turns out to be pretty efficient in practice.

3.5 Causality-Informed Code Analysis

Having identified the visually distorted elements and their corresponding resources (termed *critical elements/resources*) that have direct causal relationships with the distortion, Vetter quickly locates the WPO's functions that process the critical elements/resources based on call stacks recorded at run time (using runtime profilers like gdb). For example, when a JPEG image is missing from the optimized page produced by Compy (and thus is identified as a critical element), Vetter uses the call stack information to pinpoint that the image has been processed by 1) `proxyResponse` which extracts the the image format from the `Content-Type` field in the response header, 2) `AddTranscoder` which informs the transcoder of the image format, and 3) `Transcode` which transcodes the image file according to its format.

To help developers locate the root causes in a more fine-grained manner, Vetter also records the runtime logs of the WPO function calls (generated through automatic code instrumentation), which include the functions' runtime parameters and entry timestamps. With these, Vetter further organizes this critical information along with the call stacks as a timeline, so as to clearly depict the in-situ situations of the WPO when processing the critical elements. For the above example of Compy, Vetter organizes the call stacks of the above three functions together with their runtime parameters. Based on this diagnostic information, we easily discover that the function `AddTranscoder`'s input parameter `Content-Type` is set as "PNG", which is apparently inconsistent with the actual image

format (JPEG), causing errors during the transcoding process and thus the content loss (detailed in §4.2).

3.6 Implementation

Vetter contains three major components: WPO Runtime Logger, Distortion Detector, and WPO Debugger. The three components are implemented with a total of 2,400+ lines of code (LoC). WPO Runtime Logger records the WPO's function call stacks and runtime logs. This component is built upon gdb, Go Execution Tracer [31], and OpenTelemetry [33]. Distortion Detector records the page's resources and their loading sequence using Mahimahi [58]. It also records the SkPaint API invocations with the Skia `web_to_skp` tool [24] during page loading to construct the MSTs. Finally, WPO Debugger uses the puppeteer library [34] to monitor and manipulate the page loading process for debugging visual distortions.

4 EVALUATION

4.1 Visual-Aware Testing Performance

We evaluate Vetter's efficacy and overhead of testing visual distortions with the dataset collected in §2 (*i.e.,* the 5,000 web pages' original version and two optimized versions produced by Ziproxy and Compy). We compare Vetter with five distortion detection approaches, which are based on three common CV metrics (*i.e.,* SSIM, SIFT, and pHash), MorphSIM (*cf.* §3.3), and the CV Hybrid metric.

Setup. For the three CV-based approaches and the MorphSIM-based approach, if the similarity calculated using the corresponding metrics between the optimized and the original web pages is below a pre-determined threshold, an optimized web page will be determined as visually distorted. To find out appropriate thresholds, we try different threshold values and examine the approaches' detection performance (measured by F1 score) on Alexa top 2,500 web pages (referred to as the training set). As a result, we respectively set the threshold values for SSIM, SIFT, pHash, and MorphSIM as 0.95, 0.99, 0.91 and 0.46, which are able to maximize their F1 scores.

As solely relying on any of the CV metrics yields low F1 scores (as shown in §2.3), we further use mainstream classifiers to combine the three CV metrics together, including Decision Tree, Random Forest [43], Logistic Regression, Naive Bayes, SVM, SGD-Classifier [14], and RBF Neural Network [5]. We find that the SGD-Classifier achieves the best performance on the training set. Similarly, we combine the MorphSIM and CV metrics (*i.e.,* Vetter's testing approach) using different classifiers; this time, Random Forest excels.

With these preparations, we compare Vetter with other approaches on Alexa bottom 2,500 (among top 1M) web pages (referred to as the test set). Our rationale behind using top 2,500 (most popular pages) for training and bottom 2,500 (less popular, often non-standard) for testing is to evaluate the robustness of these approaches with two very different sets of web pages. We use the same testbed as that introduced in §2.2, and the crowdsourced results as the ground truth. We mainly focus on testing precision, recall, F1 score and detected number of visually distorted pages when evaluating different approaches' performance in testing visual distortions.

Testing Performance. Table 2 lists the testing performance of Vetter and five other comparative approaches on the test set. As shown, CV-based approaches (*i.e.,* the first four rows in the table) yield unsatisfactory performance, as they induce many FPs when tackling dynamic pages that differ greatly between different loads. On the other hand, since MorphSIM is not sensitive to pixel-level changes, some content loss and distortions cannot be detected, thus leading to a lower recall. In comparison, Vetter makes combined use of CV-based and MorphSIM-based approaches to avoid their defects, achieving the best testing performance. Detailed analysis is presented in Appendix A.4.

Table 2: Testing performance of Vetter and the other detection approaches based on the three CV metrics, the combination of the CV metrics (CV Hybrid), and MorphSIM. "# Dist." denotes the detected number of distorted pages.

Metric	Precision	Recall	F1 Score	# Dist.
SSIM	45%	88%	0.59	713
SIFT	48%	70%	0.57	532
pHash	44%	89%	0.59	738
CV Hybrid	49%	90%	0.63	670
MorphSIM	82%	80%	0.81	356
Vetter	**95%**	**91%**	**0.93**	349

4.2 Visual-Aware Debugging Results

We apply Vetter to four representative WPOs: Ziproxy, Compy, Fawkes, and SipLoader for pinpointing the root causes of the visual distortions they incur when optimizing the 5,000 pages in our dataset. As a result, Vetter successfully unravels a total of 21 previously-unknown defects: 4 in Ziproxy, 4 in Compy, 2 in Fawkes, and 11 in SipLoader. Moreover, the debugging efforts are significantly reduced by Vetter. In detail, for Ziproxy Vetter reduces the search space from ~20K LoC to 560 LoC for each defect on average. Similarly, for Compy, Fawkes, and SipLoader, the search space is reduced to only 16%, 3%, and 6%, respectively.

Concretely, we classify the defects into 11 types as shown in Table 3. In particular, we note that most of the defects root in several misconceptions or wrong assumptions of WPO developers.

Undue Reliance on HTTP Headers. HTTP headers can be improperly configured by web developers. Some WPO developers do not realize the possible misconfigurations, and directly use the headers to decide the optimization logic, thus inducing distortions. For example, Compy checks an image's format solely with the `Content-Type` field in HTTP headers, which can be inconsistent with the actual format and lead to incorrect image transcoding. Also, when compressing text files, Ziproxy adds a new `Content-Encoding: gzip` field to the response header, without deleting the original `Content-Encoding` field, causing text confusion.

Amplified Dependency Violations. The loading sequence of web page resources should obey the complex dependencies among them, so as to assure that the page is loaded properly. For instance, before a JavaScript file's execution, the resources (*e.g.,* images and CSS files) it depends on must be fully loaded. However, some mechanisms of WPOs like resource pre-fetching and script pre-execution manipulate the resources' loading sequence, and thus could cause or amplify dependency violations. In practice, we observe that SipLoader cannot capture all the dependencies during its optimization phase (mainly performed offline), since some dependencies are dynamically generated online during the page loading process. Such

Table 3: Defect types, symptoms and ratios of four WPOs.

Proxy	Defect Type	Symptom	Ratio
Ziproxy	Imprudent image	Image transcoding error	47.9%
	Conflicting fields in HTTP header	Text confusion	43.7%
	Disorder of async JavaScript	Layout disorder	5.4%
Compy	Object header-body inconsistency	Content loss	90.1%
	Insufficient support for new web protocols	Content loss	6.7%
	Incomplete request forwarding	Undesirable typesetting	3.2%
Fawkes	Lack of analysis of JavaScript dependencies	Incorrect DOM manipulations	82.7%
	Insufficient support for control characters	Web page freezing	17.3%
SipLoader	Insufficient support for compression algorithms	Resource file corruption	15.4%
	Incomplete dependency tracking during rewriting	Content loss/layout disorder	52.3%
	URL conversion error	Content loss	32.3%

limitation results in over a half of the visual distortions induced by SipLoader. Similar issues also exist in Ziproxy.

Lacking Support for Emerging Web Techniques. Compy, Fawkes, and SipLoader do not well adapt to today's emerging web techniques, thus causing visual distortions on optimized pages. For instance, SipLoader cannot recognize the resources compressed by Brotli [1], so it directly treats the resources as uncompressed. Besides, Compy cannot handle WebSocket requests, thus impairing some websites' interactive functions like online chat room.

4.3 Defect Fixing

To fix the defects, we provide either source code corrections or auxiliary middleware pre-parsing for the WPOs.

- *Consistency Checking for HTTP Headers.* Given that HTTP headers can often be misleading, we provide consistency checking between the headers and the related resources. In detail, for Compy we identify an image's actual format by sniffing its byte pattern rather than simply believing the headers. For Ziproxy, if there already exists a `Content-Encoding:none` field, we replace its value rather than adding a conflicting new field.
- *Runtime Dependency Tracking.* To prevent the optimized loading sequence from violating resources' dependencies, we build a lightweight middleware to pre-parse the HTML files using a headless Chrome browser [64]. Similar to Prophecy [57], the middleware leverages JavaScript `Proxy` objects to collect the write logs of JavaScript variables during the pre-parsing phase. With the write logs, the middleware merges all JavaScript files into a single inline script where all the JavaScript variables are properly generated based on dependencies, and then sends the rewritten HTML file to the WPO.
- *Adapting to New Web Techniques.* For Compy, we have integrated supports for WebSocket. Also, we check the `Transfer-Encoding` field in SipLoader to recognize Brotli-compressed files, and perform the corresponding compression/decompression on demand.

Impacts on Real-World WPOs. After applying the above fixes to the four WPOs, we find that nearly all (98%) of the visual distortions occurred on the 5,000 web pages in our dataset disappear. Further, to realistically improve the four mentioned real-world WPOs, we have reported our uncovered defects and the suggested fixes to all of them. Although Ziproxy's and Fawkes' developers have not replied yet, Compy's and SipLoader's developers have confirmed a total of 12 GitHub issues [15, 37] reported by us through an anonymous GitHub account named Web-Distortion. More importantly, nearly half of the fixing patches have been upstreamed to the master branch of their code base [16, 38], leading to the first major update of Compy in 2021 and a major upgrade of SipLoader in 2022. For the remaining half, they are under improvement for compatibility/security concerns.

5 RELATED WORK

Visual Distortion Testing for Web Systems. Testing visual distortions of web pages is crucial to the QoE of many web systems. Prior work has proposed several tools [12, 13, 53] towards detecting incorrect rendering of web pages for both web browsers and web applications. Specifically, for browsers, R2Z2 [69] differentiates the same HTML file's rendering results on two browsers to detect and debug incorrect rendering caused by a browser's buggy rendering pipeline. It identifies incorrect rendering using pHash, a CV metric we have extensively discussed in §2.3. Besides, a number of formal methods [54, 60–62] have been devised for verifying the layout algorithm of browsers. For web applications, existing studies [27–29, 40, 74] mainly focus on their cross-browser visual consistency by comparing the page's DOM trees on different browsers.

Differently, Vetter adopts the novel concept of morphology of web pages to address the challenges of complex dynamic web pages, which results in accurate and effective detection.

Web Problem Debugging. To debug web problems, existing tools focus on recording and replaying web pages. Two popular examples are Google's web-page-replay [26] and Telerik's Fiddler [72], which intercept HTTP traffic through DNS redirection or intermediate data forwarding to record and replay web requests/responses. Some other tools [4, 7, 67] record the detailed information of JavaScript executions and replay them for diagnosis purposes. While these debugging tools can help WPO developers uncover common program defects, they cannot well diagnose those related to web pages' visual distortions. Vetter addresses this by strategically inferring the causal relationships between visual elements and distortions with the crucial morphological hints extracted from web pages.

6 CONCLUSION

This paper presents Vetter, an automatic testing and debugging system for the visual distortion problem induced by web performance optimizers (WPOs). The problem has long been frustrating the industry by rendering WPOs unreliable or even unusable, but is never addressed due to its elusiveness and difficulty. Based on a special notion of morphology, an inherent and stable visual property of modern web pages, Vetter effectively and efficiently identifies visual distortions on even complex dynamic pages. The morphological insights, coupled with strategical distortion-element causal inference, further help pinpoint the root causes at the WPO source code level. By applying Vetter to four representative WPOs, Vetter locates crucial defects and resolves almost all distortions. In a broader sense, our ideas proposed and lessons learned root in the fundamental design patterns of modern web pages, and thus should also be useful in strengthening the reliability of other web systems like web browsers, web applications, and beyond.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable comments. We thank Zifan Zhang, Tingjun Piao, and Jinlong E for their help in data collection and analysis in the early stage of this research. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB4500703, by NSFC under Grant 62202266, and by Microsoft Research Asia.

REFERENCES

[1] 2016. *RFC 7932: Brotli Compressed Data Format.* Technical Report. RFC Group.
[2] 2023. Twitter. https://twitter.com. (2023). (Accessed on Feb. 8, 2023).
[3] Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, Michael Piatek, Colin Scott, Matt Welsh, and Bolian Yin. 2015. Flywheel: Google's Data Compression Proxy for the Mobile Web. In *Proc. of USENIX NSDI*. 367–380.
[4] Silviu Andrica and George Candea. 2011. WaRR: A Tool for High-Fidelity Web Application Record and Replay. In *Proc. of IEEE DSN*. 403–410.
[5] David S Broomhead and David Lowe. 1988. Multi-Variable Functional Interpolation and Adaptive Networks. *Complex Systems* 2, 3 (1988), 321–355.
[6] Derek Bruff. 2005. The Assignment Problem and the Hungarian Method. *Notes for Math* 20, 5 (2005), 29–47.
[7] Brian Burg, Richard Bailey, Amy J Ko, and Michael D Ernst. 2013. Interactive Record/Replay for Web Application Debugging. In *Proc. of ACM UIST*. 473–484.
[8] Chris Burns. 2015. Android Data Saver Mode for Chrome Might Also Block Ad Images. https://www.slashgear.com/android-data-saver-mode-for-chrome-might-also-block-ad-images-01416563/. (2015). (Accessed on Jan. 20, 2022).
[9] Deng Cai, Shipeng Yu, Ji-Rong Wen, and Wei-Ying Ma. 2003. *VIPS: A Vision-based Page Segmentation Algorithm.* Technical Report. Microsoft.
[10] Christian Cantrell. 2016. Everything You Need to Know about Google's Accelerated Mobile Pages. https://www.smashingmagazine.com/2016/02/everything-about-google-accelerated-mobile-pages/. (2016). (Accessed on Feb. 11, 2022).
[11] Jinlin Chen, Baoyao Zhou, Jin Shi, Hongjiang Zhang, and Qiu Fengwu. 2001. Function-Based Object Model towards Website Adaptation. In *Proc. of ACM WWW*. 587–596.
[12] Shauvik Roy Choudhary, Mukul R Prasad, and Alessandro Orso. 2012. CrossCheck: Combining Crawling and Differencing to Better Detect Cross-Browser Incompatibilities in Web Applications. In *Proc. of IEEE ICST*. 171–180.
[13] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. 2010. WEBDIFF: Automated Identification of Cross-Browser Issues in Web Applications. In *Proc. of IEEE ICSM*. 1–10.
[14] David Cournapeau. 2021. Sklearn Linear Model SGDClassifier. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html. (2021). (Accessed on Mar. 10, 2022).
[15] Barna Csorogi. 2021. Issues of Compy. https://github.com/barnacs/compy/issues. (2021). (Accessed on Mar. 15, 2022).
[16] Barna Csorogi. 2021. Merged Fixes of Compy. https://github.com/barnacs/compy/pull/68. (2021). (Accessed on Mar. 10, 2022).
[17] Drupal. 2020. AMP Display Is Activated for Webform, but Fields Are Not Present in the Generated AMP Page. https://www.drupal.org/project/amp/issues/2825270. (2020). (Accessed on Mar. 13, 2022).
[18] Facebook. 2018. How Do I Turn Data Saver Off If There Is No Option in the Help and Settings? https://www.facebook.com/help/community/question/?id=10154771843914157. (2018). (Accessed on Mar. 9, 2022).
[19] Matthew Fisher, Manolis Savva, and Pat Hanrahan. 2011. Characterizing Structural Relationships in Scenes Using Graph Kernels. In *Proc. of ACM SIGGRAPH*. 1–12.
[20] Google. 2015. Data Saver Causes Erroneus Redirect of www.changiairport.com to Different Site. https://support.google.com/chrome/forum/AAAAP1KN0B0Sf855UX0cy8/. (2015). (Accessed on Mar. 10, 2022).
[21] Google. 2017. Critical AMP Error-Content Mismatch between AMP and Canonical Pages. https://support.google.com/webmasters/forum/AAAA2Jdx3sUp10PgfhYxUI. (2017). (Accessed on Feb. 3, 2022).
[22] Google. 2019. Content Problem with Data Saver On. https://support.google.com/chrome/thread/2303283?hl=en. (2019). (Accessed on Mar. 10, 2022).
[23] Google. 2020. AMP Is a Web Component Framework to Easily Create User-First Websites. https://amp.dev/. (2020). (Accessed on Feb. 12, 2022).
[24] Google. 2022. Skia web_to_skp Tool. https://github.com/google/skia/blob/main/experimental/tools/web_to_skp. (2022). (Accessed on Mar. 24, 2022).
[25] Angular Group. 2022. Angular Platform. https://angular.io/. (2022). (Accessed on Mar. 24, 2022).
[26] Chromium Group. 2017. Web Page Replay. https://github.com/chromium/web-page-replay. (2017). (Accessed on Mar. 11, 2022).
[27] Dharma Group. 2018. Dharma. https://github.com/MozillaSecurity/dharma. (2018). (Accessed on Mar. 21, 2022).
[28] Domato Group. 2018. Domato. https://github.com/googleprojectzero/domato. (2018). (Accessed on Mar. 20, 2022).
[29] DOMFuzz Group. 2018. DOMFuzz. https://github.com/MozillaSecurity/domfuzz/tree/master/dom. (2018). (Accessed on Mar. 20, 2022).
[30] Digital Photography Review Group. 2014. Help! Problems Watching Live TV. https://www.dpreview.com/forums/post/53027375. (2014). (Accessed on Feb. 27, 2022).
[31] Go Group. 2017. Go Execution Tracer. https://blog.gopheracademy.com/advent-2017/go-execution-tracer/. (2017). (Accessed on Mar. 21, 2022).
[32] Juniper Group. 2022. Juniper Networks. https://www.juniper.net/. (2022). (Accessed on Mar. 24, 2022).
[33] Open Telemetry Group. 2022. Open Telemetry. https://opentelemetry.io/. (2022). (Accessed on May 21, 2022).
[34] Puppeteer Group. 2022. Puppeteer. https://pptr.dev/. (2022). (Accessed on May 21, 2022).
[35] React Group. 2022. React. https://reactjs.org/. (2022). (Accessed on Mar. 24, 2022).
[36] Superuser Group. 2016. Where to Turn Off Data Server in Chrome for Desktop. https://superuser.com/questions/1016592/where-to-turn-off-data-server-in-chrome-for-desktop. (2016). (Accessed on Jan. 9, 2022).
[37] SipLoader Group. 2022. Issues of SipLoader. https://github.com/SipLoader/SipLoader.github.io/issues. (2022). (Accessed on Apr. 10, 2022).
[38] SipLoader Group. 2022. Merged Fixes of SipLoader. https://github.com/SipLoader/SipLoader.github.io/pulls?q=is%3Apr+is%3Aclosed. (2022). (Accessed on Apr. 10, 2022).
[39] TestIn Group. 2021. Landing Page of TestIn. https://www.testin.net/. (2021). (Accessed on Feb. 24, 2022).
[40] Wadi Group. 2017. Wadi. https://github.com/sensepost/wadi. (2017). (Accessed on Mar. 20, 2022).
[41] Ziproxy Group. 2021. Ziproxy: HTTP Traffic Compressor. http://ziproxy.sourceforge.net/. (2021). (Accessed on Mar. 10, 2022).
[42] Compy Groups. 2021. HTTP/HTTPS Compression Proxy. https://github.com/barnacs/compy. (2021). (Accessed on Mar. 18, 2022).
[43] Tin Kam Ho. 1995. Random Decision Forests. In *Proc. of IEEE ICDAR*. 278–232.
[44] Brittany Johnson, Yuriy Brun, and Alexandra Meliou. 2020. Causal Testing: Understanding Defects' Root Causes. In *Proc. of ACM/IEEE ICSE*. 87–99.
[45] Byungjin Jun, Fabián E. Bustamante, Sung Yoon Whang, and Zachary S. Bischof. 2019. AMP up Your Mobile Web Experience: Characterizing the Impact of Google's Accelerated Mobile Project. In *Proc. of ACM MobiCom*. 1–14.
[46] Ronny Ko, James Mickens, Blake Loring, and Ravi Netravali. 2021. Oblique: Accelerating Page Loads Using Symbolic Execution. In *Proc. of USENIX NSDI*. 289–302.
[47] Zhenhua Li, Weiwei Wang, Tianyin Xu, Xin Zhong, Xiang-Yang Li, Yunhao Liu, Christo Wilson, and Ben Y Zhao. 2016. Exploring Cross-Application Cellular Traffic Optimization with Baidu TrafficGuard. In *Proc. of USENIX NSDI*. 61–76.
[48] Wei Liu, Xinlei Yang, Hao Lin, Zhenhua Li, and Feng Qian. 2022. Fusing Speed Index during Web Page Loading. In *Proc. of ACM SIGMETRICS*. 1–23.
[49] Xing Liu, Feng Qian, and Zhiyun Qian. 2017. Selective HTTPS Traffic Manipulation at Middleboxes for BYOD Devices. In *Proc. of IEEE ICNP*. 1–10.
[50] David G. Lowe. 2004. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision* 60, 2 (2004), 91–110.
[51] Shaghayegh Mardani, Mayank Singh, and Ravi Netravali. 2020. Fawkes: Faster Mobile Page Loads via App-Inspired Static Templating. In *Proc. of USENIX NSDI*. 879–894.
[52] William J. Masek and Michael S. Paterson. 1980. A Faster Algorithm Computing String Edit Distances. *J. Comput. System Sci.* 20, 1 (1980), 18–31.
[53] Ali Mesbah and Mukul R Prasad. 2011. Automated Cross-Browser Compatibility Testing. In *Proc. of ACM/IEEE ICSE*. 561–570.
[54] Leo A Meyerovich and Rastislav Bodik. 2010. Fast and Parallel Webpage Layout. In *Proc. of ACM WWW*. 711–720.
[55] Donald Michie. 1968. "Memo" Functions and Machine Learning. *Nature* 218, 5136 (1968), 306–306.
[56] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. 2015. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proc. of ACM SIGCOMM*. 199–212.
[57] Ravi Netravali and James Mickens. 2018. Prophecy: Accelerating Mobile Page Loads Using Final-State Write Logs. In *Proc. of USENIX NSDI*. 249–266.
[58] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP. In *Proc. of USENIX ATC*. 417–429.
[59] Opera. 2021. Opera Turbo Mobile Web Proxy. https://www.opera.com/turbo. (2021). (Accessed on Jan. 7, 2022).
[60] Pavel Panchekha, Michael D Ernst, Zachary Tatlock, and Shoaib Kamil. 2019. Modular Verification of Web Page Layout. In *Proc. of ACM OOPSLA*. 1–26.
[61] Pavel Panchekha, Adam T Geller, Michael D Ernst, Zachary Tatlock, and Shoaib Kamil. 2018. Verifying that Web Pages Have Accessible Layout. In *Proc. of ACM PLDI*. 1–14.

[62] Pavel Panchekha and Emina Torlak. 2016. Automated Reasoning for Web Page Layout. In *Proc. of ACM OOPSLA*. 181–194.
[63] Behnam Pourghassemi, Jordan Bonecutter, Zhou Li, and Aparna Chandramowlishwaran. 2021. adPerf: Characterizing the Performance of Third-Party Ads. In *Proc. of ACM SIGMETRICS*. 37–38.
[64] Puppeteer. 2020. Headless Chrome Node.js API. https://pptr.dev/. (2020). (Accessed on Feb. 12, 2022).
[65] Riverbed. 2022. Riverbed Networks. https://www.riverbed.com/. (2022). (Accessed on Mar. 24, 2022).
[66] Vaspol Ruamviboonsuk, Ravi Netravali, Muhammed Uluyol, and Harsha V. Madhyastha. 2017. Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *Proc. of ACM SIGCOMM*. 390–403.
[67] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *Proc. of ACM FSE/ESEC*. 488–498.
[68] Skia. 2020. SkPaint Overview. https://skia.org/user/api/skpaint_overview. (2020). (Accessed on Mar. 12, 2022).
[69] Suhwan Song, Jaewon Hur, Sunwoo Kim, Philip Rogers, and Byoungyoung Lee. 2022. R2Z2: Detecting Rendering Regressions in Web Browsers through Differential Fuzz Testing. In *Proc. of ACM/IEEE ICSE*.
[70] Stackoverflow. 2017. Google Chrome Issue with Data Saver: WebApp Not Loading. https://stackoverflow.com/questions/43736942/force-android-chrome-to-not-downsample-images/43742876#43742876. (2017). (Accessed on Jan. 13, 2022).
[71] Stackoverflow. 2018. Disable Chrome's Data Saver Optimization. https://stackoverflow.com/questions/31314119/disable-chrome-s-data-saver-optimization. (2018). (Accessed on Jan. 15, 2022).
[72] Telerik. 2020. Fiddler: The Free Web Debugging Proxy for Any Browser, System or Platform. http://www.telerik.com/fiddler. (2020). (Accessed on Mar. 1, 2022).
[73] Vue. 2022. Vue.Js. https://vuejs.org/. (2022). (Accessed on Mar. 24, 2022).
[74] Wen Xu, Soyeon Park, and Taesoo Kim. 2020. FREEDOM: Engineering a State-of-the-Art DOM Fuzzer. In *Proc. of ACM CCS*. 971–986.
[75] Xinlei Yang, Hao Lin, Zhenhua Li, Feng Qian, Xingyao Li, Zhiming He, Xudong Wu, Xianlong Wang, Yunhao Liu, Zhi Liao, Daqiang Hu, and Tianyin Xu. 2022. Mobile Access Bandwidth in Practice: Measurement, Analysis, and Implications. In *Proc. of ACM SIGCOMM*. 114–128.
[76] Christoph Zauner. 2010. *Implementation and Benchmarking of Perceptual Image Hash Functions*. Ph.D. Dissertation. University of Applied Sciences, Hagenberg.
[77] Wang Zhou, Bovik Alan, Hamid Rahim Sheikh, et al. 2004. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing* 13, 4 (2004), 600–612.

A APPENDIX

A.1 Crowdsourcing Study on Visual Distortions

We distribute our volunteer recruitment requests on a popular crowdsourcing platform [39], where opt-in users need to recognize whether there are visual distortions incurred by Ziproxy/Compy, using the screenshots of the optimized and original landing pages. If the user believes that there is a visual distortion, s/he is further asked to list the specific symptom (*e.g.,* content loss). Eventually, 18 users opted in during Dec. 11–21, 2021. Among them, 7 are male and 11 are female, with ages ranging from 20 to 53. Each user can take any number of tasks and receive the corresponding rewards. The only constraint is that each task should be finished by at least three users, so that majority voting is possible for each task.

A.2 Challenges of Detecting Visual Distortions with CV Metrics

In order to understand the performance of detecting visual distortions using CV metrics, we treat the entire web page as a static snapshot image, and directly compare the original and optimized pages' final rendering results using three widely-used CV metrics, including 1) *structural similarity* (SSIM) [77], 2) *scale-invariant feature transform (SIFT)* [50], and 3) *perceptual hash* (pHash) [76]. Our evaluation results (*cf.* Table 2) show that making both separate and combined use of the three CV metrics yield poor detection results (precision <50%). A deeper analysis shows that such unsatisfactory performance is owing to dynamic web pages whose rendering results differ from one load to another. Specifically, the dynamic visual elements include rotating banners, randomly selected texts/images, visitor counters, and so on. These dynamics can easily disrupt the above-described pixel-by-pixel CV comparisons between two pages' snapshots, incurring many false positives.

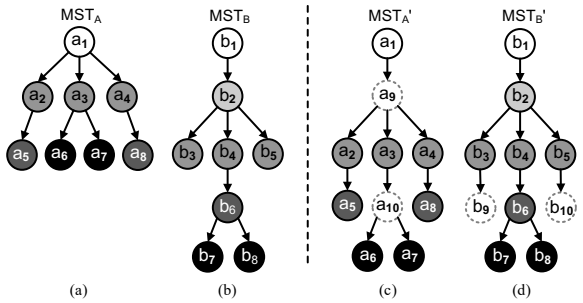


Figure 6: Hierarchy matching between two MSTs by adding virtual nodes to realign them.

A.3 Matching Strategies between MSTs

This part first details the algorithm design of the level-by-level hierarchy matching between two MSTs at the granularity of *node groups*. Here a node group refers to a set of nodes that share the same parent node. Then, we discuss the node matching among the already matched groups' inner nodes.

Hierarchy Matching. To begin with, we are at the root level (Level-0) of both MST_A and MST_B. Here by level we refer to nodes that have the same number of edges along their paths to the root node, *e.g.,* node a_5, a_6, a_7, and a_8 in Figure 6(a) are of the same level. Therefore, a level may contain several *node groups* (*e.g.,* MST_A's Level-2 contains three groups).

For each group in MST_A's root level (obviously there is only one group in the root level), we examine whether it exactly matches MST_B's any group in the root level in terms of their nodes' number and inner structure, *i.e.,* the groups have the same number of nodes and inner structure. Note that the specific process for structure matching between two groups of counterpart nodes will soon be detailed in Node Matching. Naturally, there are two outcomes—we either find two groups that match each other, or we do not. If it is the former case, we can mark them as matched and move to the next group in the current level in MST_A; if all groups at the level have been traversed, we go down to the lower level. Otherwise, we say a *hierarchy mismatch* occurs.

Upon a hierarchy mismatch, we try to find a matched group in MST_B no matter which level the group lies at. To this end, we examine groups in MST_B also in a top-down manner. If we cannot find any matches, we will go back to the specific level in MST_B where the hierarchy mismatch occurs, and directly use a group at the level that best matches the mismatched group in MST_A. Else, if we find a matched group at MST_B's Level-K, we then mark them as matched as well, and *realign* the counterpart matched group in MST_A to Level-K by adding virtual nodes (as shown in Figures 6(c)(d)). By traversing all the groups in MST_A following the above procedure, we can eventually accomplish hierarchy matching.

During the above process, we recursively compare and realign the MSTs' node structures (from top to bottom) to mitigate the negative influence of their different hierarchies. In the worst case, there exists no group of nodes in MST_A that matches any group of nodes in MST_B; assuming MST_A and MST_B both contain $O(n)$ nodes, our hierarchy matching procedure would incur $O(n^2)$ time complexity, which would be pretty high for a large yet realistic n, especially when each operation of hierarchy matching is accompanied by multiple node matching operations (as detailed soon) whose complexity is not included here. To reduce the required comparisons, we further adopt a *memorization algorithm* [55] to accelerate hierarchy matching as follows.

Our idea of memorization algorithm is motivated by a key observation: when we go down to lower levels in the two MSTs, identical comparisons may appear many times. To avoid such repetitive comparisons, when we compare two groups, we insert the comparison results to a hash table. If two groups are identical, they are stored under a same key as an array: $(K, [V_1, V_2])$; here K is the inner structure of either group, while V_1 and V_2 respectively include the two groups of nodes' labels. Otherwise, they are stored under different keys. Thereby, all the repetitive comparisons can be avoided, and we only need to make $O(n)$ comparisons to fulfill hierarchy matching, rather than the original $O(n^2)$ comparisons.

Node Matching between Groups. We now detail the process of node-by-node structure matching between two groups of nodes from two MSTs. This process acts as the basic operation unit invoked by hierarchy matching as described above.

Given two groups of nodes from two MSTs, the first thing is to extract a set of properties for each node, based on which we can measure the similarity among different nodes for structure matching. When constructing the set of properties, we ignore a node's non-morphological properties such as size and position; instead, we focus on the node's topological form and structural relations to the other nodes within the same group. The two kinds of information are both obtained from the logs of the SkPaint APIs (as discussed in §3.2). Recall that we take seven major structural relations between the graphic elements of a web page into consideration. All in all, we make integrated use of all the eight properties (one from the topological form and seven from the structural relations), which together constitute the node's *property set* (PS).

Based on the above, we define the similarity of different nodes through an intuitive similarity function: $SIM_{a,b} = \frac{|PS_a \bigcap PS_b|}{|PS_a \bigcup PS_b|}$, where PS_a and PS_b are the property sets of nodes a and b.

Hence, we conduct structure matching as follows. First, nodes in two groups with identical property sets are matched preferentially, which can be accomplished with $O(n^2)$ time complexity, where each group contains $O(n)$ nodes. Further, to generate the (best) matching with the highest average similarity among the remaining nodes, we convert the two groups of nodes to a bipartite graph—the two groups of nodes constitute two vertex subsets, and each edge between the two vertex subsets is given a weight as specified in the similarity function above. Thereby, finding the matching scheme with the highest average similarity between the two groups of nodes is equivalent to finding the maximum matching in the derived bipartite graph; for the latter, we can leverage the classic Hungarian algorithm [6] to solve it with $O(k^3)$ time complexity, where each group contains k nodes.

A.4 Evaluations of Detecting Visual Distortions

Testing Performance of Different Approaches. As shown in Table 2, the performance of all the CV-based approaches are unsatisfactory, with F1 score <0.65, and precision <50%. By analyzing the results, we find that CV-based approaches induce many false positives (FPs), most of which are related to dynamic pages that differ greatly between different loads. On the other side, the testing recalls of CV-based approaches are reasonable (≤90%), inducing a few false negatives (FNs), which mostly are content loss that causes obvious layout changes but only slight pixel-level differences.

Compared with CV-based approaches, the MorphSIM-based approach substantially improves the testing precision from <50% to 82%, but slightly decreases the recall from 90% to 80%. The results indicate that the MorphSIM-based approach can well distinguish dynamic pages from visually distorted ones, thus bringing remarkable precision improvements compared with CV-based approaches. On the other hand, as MorphSIM is not sensitive to pixel-level changes, some content loss/distortions cannot be detected, thus leading to a lower testing recall.

Given that the CV metrics and MorphSIM well complement each other, Vetter makes a combined utilization of them, and thus achieves the highest testing F1 score (0.93), precision (95%) and recall (91%). Of course, Vetter also incurs false positives and negatives in practice. On the test set, Vetter's FP rate is 1% and FN rate is 9%. By manually examining the false positives, we find that all of them are highly dynamic in terms of not only concrete content but also visual structure. For example, the visual structure of an HTML5 gaming page optimized by Ziproxy changes significantly compared with that of the original page. In this case, MorphSIM between the original and optimized pages falls below the threshold (0.46), leading to a wrong decision.

As to the false negatives, we observe that all of them suffer a small-size content loss. In particular, on the optimized page, the absence of a small visual element leads to a leaf node's missing in its MST, which usually brings little impact on the calculation of both MorphSIM and the CV metrics. Thus, this small content loss can hardly be captured by Vetter. However, when these elements are semantically or functionally important, *e.g.,* a login button, users could easily notice such distortions, thus leading to FNs. Note that such FNs are strongly related to the page-specific semantics, and thus are really hard to detect.

Testing Efficiency of Vetter. We next evaluate the time overhead of Vetter for testing visual distortions. In general, the overhead mainly involves: (1) the page loading process of the original and optimized pages, (2) calculation of the CV metrics and MorphSIM for each page, and (3) delays incurred by machine learning models. We then measure the time overhead of Vetter when testing visual distortions on the test set. When running on a budget VM server with a dual-core CPU @2.3 GHz, Vetter's average testing time of a web page ranges from 1.7 s to 5.2 s, averaging at 3.2 s. In particular, we observe that the time overhead is mainly incurred by the page loading process, which takes 3.1 s on average, while the other two factors together take only 62 ms on average. That is to say, almost

all (98%) the time overhead comes from the loading process or optimization routines of WPO, rather than the Vetter's testing logic. Such performance of Vetter is largely owing to Vetter's minification of a page's morphology (§3.2) and its efficient morphological similarity calculation (§3.3).

A.5 Artifact Appendix

Abstract

The artifacts of Vetter are publicly available at GitHub. To facilitate a better understanding of Vetter, we provide detailed instructions on how to build, deploy, and use Vetter. Please refer to our README file at https://github.com/Web-Distortion/Vetter/ for details.

Scope

The artifacts can be used to reproduce the major results of Vetter.

Contents

The artifacts include the source code of Vetter, the detailed defects of four widely used WPOs (Ziproxy, Compy, Fawkes, and SipLoader) we have found using Vetter, and the crowdsourcing datasets involving 5,000 websites regarding the WPO-incurred visual distortions.

Hosting

Code and data are hosted in the `main` branch of `Vetter` repository.

GitHub Repo. https://github.com/Web-Distortion/Vetter

DOI for the Artifacts. https://doi.org/10.5281/zenodo.7601984