USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Detecting and Diagnosing Errors in Serving Archived Web Pages

Jingyuan Zhu, *University of Michigan;* Huanchen Sun and Harsha V. Madhyastha, *University of Southern California*

https://www.usenix.org/conference/nsdi26/presentation/zhu-jingyuan

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

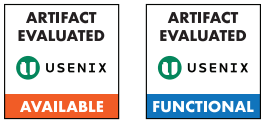
May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology



Detecting and Diagnosing Errors in Serving Archived Web Pages

Jingyuan Zhu[1] Huanchen Sun[2] Harsha V. Madhyastha[2]
[1]*University of Michigan* [2]*University of Southern California*

Abstract—**Web archives crawl and save copies of pages from the web, enabling users to interact with web pages in the form they existed in the past. Prior to serving any archived page, an archive rewrites the page's source so that users' browsers fetch the page's resources from the archive, not from the servers which originally hosted them. But, on many modern pages, an archive's edits to crawled scripts result in a loss of fidelity, i.e., an archived copy fails to accurately mimic the original page even when the archive had crawled all resources on the page.**

To help the developers of archival systems identify and fix the bugs which result in incorrect rewrites of crawled pages, we present FIDEX. First, FIDEX enables accurate identification of the pages on which an archive violates fidelity. It does so by tracking and comparing the execution of scripts between when a page is crawled and when its copy is loaded. In comparison to existing methods which compare the two loads using either screenshots or the errors reported by the browser, FIDEX reduces the false positive rate from around 70% to less than 10%. Second, on every page on which it identifies a loss of fidelity, FIDEX pinpoints which subset of the archive's edits to the page are erroneous. Leveraging this input to fix bugs in the most widely used archival system, we reduced the fraction of archived pages which violate fidelity from 15% to 9%.

1 Introduction

Content on the web is notoriously short-lived. A page's content today often differs from what it was yesterday or last week [35, 39, 64]. Over longer timescales of a few months or years, pages are deleted or moved, and domains expire [40, 51, 57].

A plethora of web archives [11, 13, 20, 22] (e.g., Internet Archive and Library of Congress) aim to address this ephemerality of web content. Web archives crawl and save copies of web pages, so that a user can lookup a historical copy of any page. Users consult archived page copies for a variety of reasons: when they encounter dead links on the web [12, 26], to conduct longitudinal analyses of the web [29, 58], and even as evidence in legal cases [2, 9].

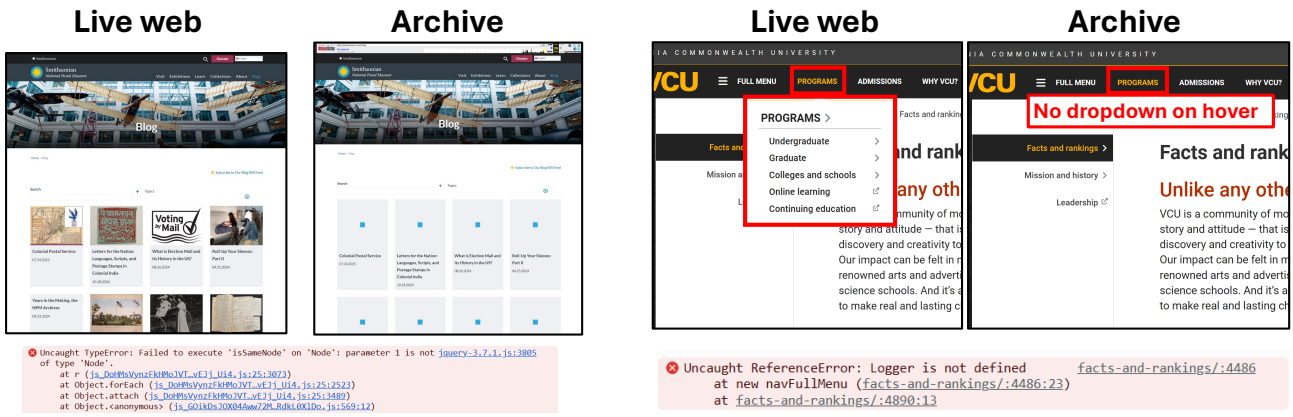
Web archives support this wide range of use cases by striving to serve archived pages with high fidelity, i.e., when a user loads an archived copy of a page, the copy must look (i.e., content and layout) and function (e.g., menus, buttons) like the original page did when this copy was crawled. Several solutions have been developed to address some of the challenges: pages may use JavaScript to fetch resources, so archives crawl them with browsers [4, 5]; some content may load only when the user interacts with the page, so archives emulate all such interactions [7]; since the URLs of resources on a page can be non-deterministic, when serving a page to a user, archives can match the control flow to that encountered during crawling [36]. Yet, even if a state-of-the-art archival system applies all of these measures, we estimate that roughly 13% of archived pages are either missing some content (e.g., images are not displayed despite being archived) or fail to preserve functionality that ought to work on an archived copy (e.g., navigation sidebars). Figure 1 shows two examples.

This loss of fidelity is a result of the modifications that an archive must make to crawled pages, so that users' browsers fetch any page's resources from the archive. On modern web pages, in addition to rewriting resource URLs which are statically embedded in a page's source, an archive must also instrument scripts so that resource URLs which are dynamically constructed get rewritten at runtime. Unfortunately, due to the wide variety of JavaScript code prevalent across the web, an archive's edits to crawled scripts often affect their execution in unanticipated ways.

To fix the bugs in archival systems which result in incorrect rewrites of archived scripts, the standard playbook would be to rely on any error encountered when loading an archived page both as evidence of a bug [54, 62] and as a starting point for diagnosing the root cause [28, 48]. However, such an approach is not fruitful for two reasons. On the one hand, most errors – 56% in our corpus – are not indicative of user-visible problems; many errors occur in code that either has no impact on the page's appearance (e.g., analytics) or powers functionality which will not work on archived pages (e.g., logins). On the other hand, an archive's edits often cause scripts on a page to execute incorrectly without resulting in any browser-reported errors.

In this paper, we present FIDEX (Fidelity violation Exposer) to help the developers of archival system address these challenges. Developers will still need to determine *how* to fix any identified bug, but FIDEX frees them from having to identify *when* there is a bug and diagnose *what* to fix. To this end, we make two contributions.

First, FIDEX accurately identifies when an archived copy fails to match the page originally crawled, without relying on errors reported by the browser. Existing archival systems use screenshots to compare the two loads [17]. However, on modern pages which have dynamic components such as animations and carousels, comparing screenshots across loads raises many false alarms, e.g., in our corpus, screenshots indicate a loss of fidelity on 69% of archived pages. To accurately ensure that pages are visually and functionally equivalent, FIDEX instead monitors the execution of scripts both when a page is crawled and when its copy is later loaded.



(a) (b)

Figure 1: **Screenshots when loading pages from (a) postalmuseum.si.edu and (b) vcu.edu from the web (left) and from an archived copy (right). In (a), images for blog posts are missing even though they were crawled. In (b), hovering on the menu bar fails to reveal the dropdown. At the bottom, we show the cause: the runtime errors that result from the archival system's rewriting of scripts.**

Second, FIDEX's fine-grained visibility into the execution of scripts enables it to accurately hone in on the erroneous rewrites that result in poor fidelity. For any runtime error observed when loading an archived page, we show how to determine whether the line of code where the error occurs causally precedes the code responsible for any of the components missing on the page. Even when an archive's incorrect rewrite of a page does not result in any errors, we show that the negative impact of that rewrite can be inferred from violations of web API specifications.

We have used FIDEX to check the fidelity offered by pywb [23], the most mature and widely used system for serving archived pages. We test pywb's ability to serve US government web pages that were archived during the presidential transitions in 2016 and 2020, pages archived from art websites, and pages drawn at random from a wide range of site rankings. When FIDEX identifies a fidelity violation, it is wrong on less than 10% of pages and it is able to pinpoint the root cause on 64% of pages. Importantly, we fixed a handful of bugs which FIDEX identified as the root cause of pywb's incorrect rewrites on a majority of affected pages. Doing so reduced the fraction of pages for which FIDEX reports a violation of fidelity from 15% to 9%. Our source code and datasets are at https://github.com/USC-NSL/FidEx.

2 Background

2.1 Operation of web archives

For any page on the web, an archive may collect many snapshots over time, in order to maintain a record of changes made to that page. Each time, the archive captures all resources (e.g., HTML, CSS, scripts, and images) on the page, and stores them into one or multiple bundled Web ARChive (WARC) [19] files so that they can be loaded independently

When a user wishes to view an archived page, they specify the URL of that page and choose the particular snapshot of interest to them. For example, a user can visit https://web.archive.org/web/20201104102000/https://www.nytimes.com/ to view the Wayback Machine's copy of the NYTimes home page captured on the morning of 4 November, 2020, the day after the US presidential election. To ensure that the user's browser fetches the page's resources from the archive and not from the original servers that hosted them, a web archive can employ one of two methods.

Need for rewriting. Similar to many web record and replay tools [31, 47], a user's browser can load any page from an archive via a trusted proxy. The browser can request any resource using its original URL. The proxy relays the request to the archive after rewriting the URL on-the-fly to refer to the copy of the resource hosted on the archive. This approach enables the archive to serve crawled resources unmodified.

However, relying on a trusted proxy to enable access to web archives is not a practical option for three reasons. First, not all users are capable of setting up an appropriate proxy configuration. Second, even if they can, users have to explicitly accept the archive's certificate for TLS interception. Third, not every environment (e.g., within a corporate network) permits the use of such a proxy configuration.

Hence, popular web archives such as the Wayback Machine instead rewrite crawled resources. Web archives employ two kinds of rewriting techniques.

Static rewriting. Traditionally, web archives have rewritten all elements within a page's source that could result in a request to another resource. Common targets include attributes such as the `src` in `<img>` tags, hardcoded URLs in JavaScript's fetch requests, and URLs defined using the `URL()` function in CSS. Such modifications are made *before* resources are served to clients.

Dynamic rewriting. With the emergence of dynamically fetched content via JavaScript (JS), static rewriting is insufficient to account for resource URLs which are determined at runtime. Hence, web archives now also perform dynamic *client-side* rewriting, as illustrated in Figure 2.

```
// Override fetch method at start of page load
let originalFetch = fetch;
fetch = function (url) {
    Code to change url to archived format ...
    return originalFetch(url);
}
```

(a)

```
{
  // "wombatAssignFunction" returns an overridden
       document object
  let document = wombatAssignFunction("document");
  (async function navigate(){
      let response = await fetch("/new-url.json");
      response = await response.json();
      document.location.href = response.url;
  })()
}
```

(b)

Figure 2: **Example of dynamic rewriting of resource URLs. Code inserted by a system like pywb [23] is in green background. Overridden objects and methods are in blue bold text.**

- First, the archive includes a script such as `wombat.js` [25] in the page's main HTML. This script, which executes before any other scripts on the page, overrides all JS functions and element methods that are capable of initiating requests. For example, in Figure 2(a), the `fetch` method is overridden so that, before an AJAX request for a resource is initiated, its URL can be modified to the corresponding archived copy.
- Some DOM native objects such as `window` or `document` cannot be overridden. To tackle this, web archives wrap the code in every JS file within a local scope (i.e., between "{}") so that all global DOM objects are changed into local objects which can be overridden. Additionally, a code snippet is inserted at the beginning of each script to facilitate access to the 'localized' versions of these objects. For example, in Figure 2(b), the setter of `document.location.href` is overridden so that its value will be prefixed with the web archive's hostname.

2.2 Fidelity issues due to dynamic rewriting

Given the vast diversity of JS code on the web, rewriting every page's source as described above often leads to a variety of unexpected errors. During a user's load of an archived page, a syntax error or runtime error introduced by overriding JS objects and methods either prevents or halts the browser's execution of the script in which this error occurs. Even if there are no overt errors, the archive's rewrite of scripts can alter the flow of execution, e.g., by causing the values of variables to diverge. As a result, any content/functionality for which the code that does not get executed was responsible is rendered missing/broken.

To show that this problem is not specific to any one archival system, Figure 1 presents two examples where the archived copy of a page fails to mimic the original page. These pages are replayed incorrectly by pywb, the Wayback Machine, and ReplayWeb.page [24], another mature open-source framework for loading archived pages. In all cases, client-side dynamic rewriting results in unexpected runtime errors, which preempt the execution of JS code responsible for constructing the rest of the page.

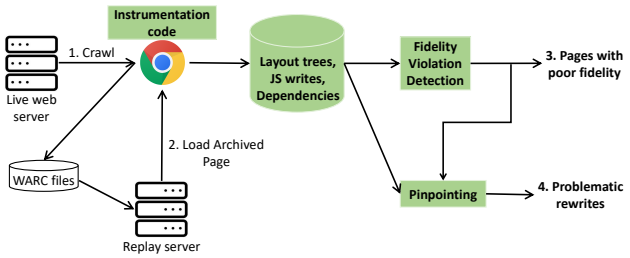


Figure 3: **A web archive's workflow augmented with FIDEX, whose components are highlighted in green.**

Consider the example in Figure 1(a). On this page, images are lazily loaded by a script which modifies the `src` attribute of every image after the page's `load` event fires. On any load of the Wayback Machine's copy of this page, the event handler which runs this script fails to get registered because of an error introduced by the archival system's override of the `document` object to rewrite its APIs. Specifically, due to incorrect rewrites, the overridden and native versions of the `document`'s `documentElement` attribute are compared, which results in an uncaught exception because they are objects of different types. This runtime error prematurely halts the execution of the script which registers the above-mentioned event handler.

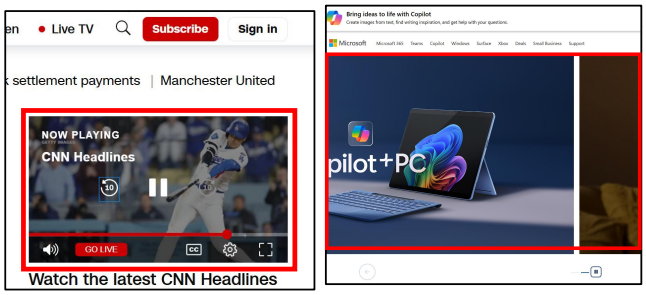
3 Overview

To help web archives detect and fix their incorrect rewrites of scripts, we enhance their workflow with FIDEX. As illustrated in Figure 3, FIDEX augments an archive's browser-based crawler. Both when an archive crawls a page from the web and when it loads its local copy to check the quality, the FIDEX-augmented browser records fine-grained information characterizing the page load. FIDEX compares the logs gathered during the two loads to identify a mismatch between the original page and its archived copy. On every page on which it identifies a loss of fidelity, FIDEX pinpoints the erroneous rewrites which need to be fixed.

3.1 Goals

We design FIDEX with three objectives in mind.

- **Accurate detection:** First, we strive to flag fidelity violations with high accuracy, prioritizing the need to minimize false alarms. Whenever an archived page fails to precisely mimic the original page, an assessment of the severity of the differences is ultimately subjective; it depends on what page content/functionality users and developers care about. We aim to inform this assessment by highlighting the differences and estimating their potential impact, e.g., the size and position of the elements which are missing on the archived page.



(a) (b)

Figure 4: **(a) An animation on cnn.com can cause screenshot to differ between live page and archived copy. (b) Image carousel on microsoft.com can cause page layout to also differ.**

- **Accurate diagnosis:** Second, FIDEX should pinpoint only the rewrites which indeed cause observed fidelity violations. In doing so, we hope to simplify the task of human developers to identify the bugs in archival software that need to be fixed.
- **High-throughput crawling:** Lastly, we aim to preserve an archive's ability to crawl pages at high throughput. Therefore, FIDEX's modifications to browser-based crawlers must impose low overhead.

3.2 Challenges

Fidelity violations vs. page dynamism. A common approach to identify differences between loads of a live page and its archived copy is to compare screenshots [17]. However, on pages such as Figure 4(a) which include videos and GIFs, the screenshot can vary across loads based on when the screenshot is captured.

To compare multiple loads of a page in a more robust manner, prior work [33, 34, 60] has therefore proposed extracting and comparing a structured representation of the page's appearance. However, no representation of the content in each visual component suffices to accurately account for the dynamism on modern web pages. For example, consider the page in Figure 4(b) which contains an image carousel. If we represent each bounding box by the the resource which provides its content [33, 34], then two loads will appear to differ if the image shown in the carousel at the time of comparison differs. If we instead use a more coarse-grained representation – e.g., that *some* image is displayed within the bounding box [60] – then, an archived page may match the original page even if the carousel is rendered dysfunctional due to the archive's incorrect rewrite of the page.

Runtime errors $\not\Rightarrow$ fidelity violations. Whenever there is a loss of fidelity, it would be natural for developers to focus on the errors (e.g., failed resource fetches or runtime exceptions) observed only when loading the archived copy. This would be akin to prior work on fault localization (FL) [65], which leverages crash stack traces [54] or bug reports [62] to pinpoint the root cause of a fault. However, in our setting, such an approach can lead developers astray as errors that occur when loading an archived page often have no impact on the page; many scripts either do not visibly affect the page (e.g., scripts related to analytics or performance monitoring) or power functionality that will not work on archived pages (e.g., logins, live chat, and reCAPTCHA).

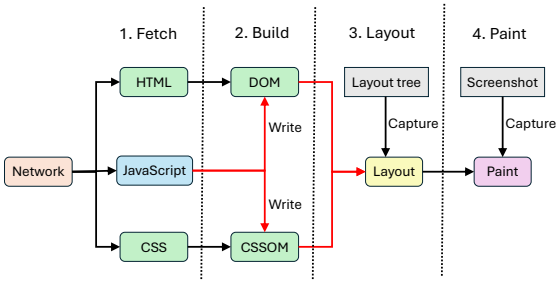


Figure 5: **Illustration of browser's critical rendering path.**

Fidelity violations $\not\Rightarrow$ runtime errors. A final challenge is that many archived pages violate fidelity without the browser reporting any errors. Scripts on the page might catch and handle errors without making them externally visible. Or, incorrect rewriting can lead to divergence in the control flow without triggering any exceptions. Prior work [28, 38, 48, 61] attempts to locate such silent errors by identifying code which is executed only on test cases which fail, but not on successful tests. However, such methods are ill-suited for our context. If we compare the correct load of a page from the web to the incorrect load of its copy from an archive, the difference between the two loads will correspond to *all* of the archive's instrumentation of the page's code, not just the subset responsible for the loss of fidelity.

3.3 Guiding observations and approach

Our approach to address the aforementioned challenges is guided by how modern browsers render pages (Figure 5). After the browser fetches a page's resources over the network, it constructs the DOM and CSSOM trees. It then combines them into a layout tree [16] which determines the positions and dimensions for every visible element. The browser finally paints pixels for each element in the layout tree.

Viewing a page load as in Figure 5 makes it apparent why using a screenshot or the page's layout is insufficient to compare two loads of a page. The DOM and CSSOM trees, though initialized from the page's HTML and CSS stylesheets, are continually updated by JavaScript code on the page. Therefore, when comparing an archived copy to the original page, we cannot compare any particular snapshots captured when loading the two versions; that comparison is prone to *when* the snapshots were captured.

Given the dynamism on modern pages, we declare that an archive preserves fidelity when serving its copy of a page if, for *any* screenshot captured when the page was crawled, the archived copy will *eventually* produce an identical screenshot. We assume that the archive ensures that sources of

non-determinism encountered when crawling a page are replayed when serving the page [36]. For any archived page, we separately evaluate the fidelity of its initial load and of each of the interactions (e.g., hover, click) on the page.

To evaluate whether any archived page satisfies the criterion defined above, FIDEX tracks the JavaScript writes which affect the page's visual appearance. We do so both when crawling a page and when the crawled copy is later served. On the one hand, it is problematic if any of the JS code which contributed to the page's rendering when it was crawled is never executed on the archived copy; whether the page's layout matches up is immaterial. On the other hand, even if the page's end state differs, we can reliably infer that fidelity was preserved if the code that contributes to the page's visual appearance is identical in both loads.

Tracking the execution of scripts also enables FIDEX to aid the developers of archival systems in two ways. First, by establishing how the execution of every script on a page impacts the page's rendering, it can precisely determine which runtime errors causally lead to observed fidelity issues. It, thus, frees developers from having to inspect irrelevant errors. Second, FIDEX's comparison of how the scripts on a page are executed prior to and after rewriting enables it to identify discrepancies which contribute to differences in the page's appearance. Consequently, FIDEX can shed light on incorrect rewrites which are otherwise not apparent from browser-reported errors.

4 Design

To realize the approach described above, we need to answer several questions. What information should FIDEX collect during page loads? How should it use this information to accurately and comprehensively identify fidelity violations? What methodology should FIDEX employ to nail down which of the errors observed when loading archived pages are worth digging into? How can FIDEX uncover problematic rewrites which do not result in any errors? We describe our answers to these questions next.

Since FIDEX focuses on detecting and diagnosing fidelity issues caused by incorrect JS rewriting, we first need to identify and exclude archived pages which violate fidelity for other reasons: incomplete crawling (e.g., because of not mimicking user interactions), non-determinism in the page, implementation limitations in the archival system (e.g., lack of support for WebSockets), etc. To filter out such pages, FIDEX compares each page's live load to its load through the archive's proxy mode, which replays the page without applying any JS rewrites. If this comparison reveals differences, the cause for the loss of fidelity is unrelated to rewriting, and developers can evaluate these pages independently without relying on FIDEX.

4.1 Identifying fidelity violations

FIDEX represents the outcome of any page load as the page's layout tree along with the set of JS writes executed on each node in the tree. The layout tree is an internal browser structure that cannot be directly accessed. But, we approximate it by running JavaScript that iterates through the DOM tree to identify all visible elements, and collects relevant rendering details. We gather every element's dimensions and computed CSS styles, such as animation properties. For any element containing content, such as `<img>`, `<p>`, and `<video>`, FIDEX also records attributes that affect the browser's composition and painting of the element.

```
let changed = False;
// Impl 1: Change element's class upon message recv
window.addEventListener("message", function(event) {
    changeElementClass(element, event);
    changed = True;
});
// Impl 2: Fallback if no message within 10 seconds
window.setTimeout(function() {
    if (!changed)
        changeElementClass(element, null);
}, 10000);
```

Figure 6: **Example inspired by the code found on skype.com in Dec 2024, wherein a fallback implementation runs if the primary one fails, both resulting in the same layout outcome.**

FIDEX captures JS writes by overriding a) browser Web APIs that modify DOM elements (e.g., Node's `appendChild`, Element's `setAttribute`) and b) property setters for `CSSStyleDeclaration` and `ClassList`. FIDEX associates each write with the element on which the operation is invoked, all descendant elements in the layout tree, and any elements included in the argument list. FIDEX records only those JS writes which modify the associated element's layout information.

A tricky, but important, detail is to determine what constitutes a unique *write*. On the one hand, in different loads of the same page, the JS code on the page might execute the same API with different arguments. For example, a carousel might periodically switch the displayed image by modifying the `src` attribute of an image tag. On the other hand, we cannot identify a write simply by the API invoked. This will merge multiple invocations of popular APIs (e.g., `appendChild`) into a single write. FIDEX balances these considerations by representing every unique write by the combination of a) the API invoked and b) the call stack at the time of invocation.

4.1.1 Comparing JavaScript execution

Given the layout trees from the loads of a live page and its archived copy, and the set of JS writes captured for every element in the trees, FIDEX faces two options for how to compare the two loads.

- On the one hand, if we only match JS writes, we find that we falsely flag pages wherein, to maximize compatibility across browser environments, web developers employ multiple implementations (like in Figure 6) to achieve the same functionality. In such cases, loads of a live and

archived page might execute different code, but produce the same layout tree.

- On the other hand, even if the JS writes for some elements differ, say we consider an archived page as equivalent to the original live page if the layout details of every visual element match. Then, we risk failing to detect instances of broken dynamism. For example, as previously discussed in §3, on a page which contains an image carousel, the carousel may be dysfunctional on the archived copy, even though the page's layout matches the live page's at the moment of capture.

In this tradeoff, we choose the latter option for two reasons. First, we empirically observe that the reduction in false positives outweighs the potential increase in false negatives. Second, we find that the risk failing to detect broken dynamism on an archived page can largely be mitigated by waiting for the "working set" of dynamic behaviors to complete before capturing the layout tree (§4.1.2).

Specifically, FIDEX compares a live page to its archived copy as follows. FIDEX traverses the layout trees from both loads from the root. For elements in the same position in either tree, FIDEX considers the two elements as equivalent either a) if both elements include the same layout information, or b) if the set of JS writes associated with these elements is identical. If all elements are matched during the traversal, FIDEX concludes that the live and archived pages are equivalent. Otherwise, FIDEX flags the archived copy as having fidelity issues.

To compare JavaScript executions for interactions on the page, FIDEX further performs this comparison after every user interaction on the page. FIDEX triggers interactions by 1) collecting all event listeners from visible elements, 2) grouping event listeners by their handler functions and the classes of their associated elements, and 3) selecting one interaction from each group and invoking the corresponding event handler.

4.1.2 Timing of comparison

Thus far, we have discussed what information FIDEX records when loading pages and how it compares this information to identify fidelity violations. But, *when* in a page load can FIDEX be certain that it has collected all the information necessary for comparison? This is an important consideration for two reasons. First, some JS-controlled dynamic components (e.g., carousels) kick into action only a few seconds after the browser's `onLoad` event fires. Second, when crawling a page and later testing its archived copy, it is important to trigger all interactions on the page, as some of these result in extra network fetches. FIDEX must compare the two page loads only after the changes associated with the fetched resources have been applied.

Therefore, it is essential for FIDEX to wait until all dynamic components on the page exhibit their intended behavior before collecting the layout tree and the corresponding JS writes. We observe that the key source of delayed dynamism is asynchronous JavaScript execution. For example, carousels are often implemented by adding a callback through `setInterval` or `setTimeout` to modify the corresponding elements. Similarly, interactions that involve additional resource fetches often attach a callback to fetch APIs that interact with the DOM.

To wait to see all dynamism on a page, FIDEX overrides Web APIs that add asynchronous callbacks. The target APIs include timers (e.g., `setInterval` and `setTimeout`), promise fulfill/reject callbacks (e.g., `Promise.then` and `Promise.reject`), and network request callbacks (e.g., `XMLHttpRequest.send`). Whenever any of these APIs are invoked, FIDEX adds the provided callback to its unfinished list. FIDEX removes a callback from the list once it is fired, and waits for the list to become empty. To account for dynamic components which, in order to change the page periodically, are implemented with callbacks that repeatedly add themselves to the event loop, FIDEX adds every callback function to the list at most once. Additionally, FIDEX times out on callbacks associated with promises that never resolve (i.e., remain in a pending state forever).

Since the behavior of parameterized callbacks can potentially differ across invocations, it may seem risky to track each callback only once. However, we empirically observe that such cases are rare. In contrast, it is far more common for a callback to be periodically invoked with different arguments (e.g., a carousel cycling through images). Treating each such invocation as a new instance would prevent FIDEX from ever considering the page load as complete.

4.2 Pinpointing root cause of fidelity violations

Next, we shift our attention from detection to diagnosis.

FIDEX aims to aid developers in their effort to identify the bugs in archival systems which result in fidelity violations. First, FIDEX pinpoints the runtime errors which result in either missing content or broken functionality. Prior work [54, 63] has shown that crash stack traces help developers identify code defects and fix bugs. Second, on pages which violate fidelity without resulting in any errors, FIDEX highlights the rewritten code on the page – a specific line, function, or file – which results in the observed fidelity violation. FIDEX may not always identify the precise origin of divergence between a live and archived page. But, by pointing out problematic rewrites, it will help developers focus their debugging efforts.

4.2.1 Runtime error based diagnosis

Runtime errors in JavaScript are caused by 1) syntactically incorrect script files that cannot be parsed, or 2) uncaught exceptions that halt execution. In the former case, all code in the script that cannot be parsed does not get executed. In the latter case, the browser skips executing the remaining code in the script file where the error occurs. As a consequence,

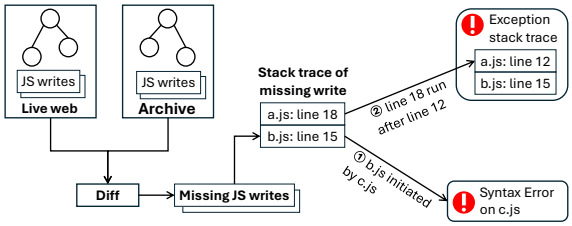


Figure 7: **How FIDEX confirms whether a runtime error led to violation of fidelity. Either syntax errors (case 1) or uncaught exceptions (case 2) can cause missing JS writes.**

the archived page ends up missing many modifications to the layout tree that were made in the load of the live page.

Detecting runtime errors when loading a page is straightforward, e.g., with Chrome, FIDEX just needs to listen for the `exceptionThrown` event from the browser's DevTools protocol. If a Syntax error is thrown only when loading an archived page, it is evident that the script where the error occurs has been incorrectly rewritten. Similarly, a Type error is often thrown on an incorrectly rewritten line of code.

However, not every runtime error caused by rewriting will result in fidelity violations. For example, a syntax error in a script that is fetched but never used has no effect on the page's rendering. Similarly, exceptions thrown in asynchronous events may not halt any JS writes that impact the layout tree.

To validate if a runtime error impacts page fidelity, we could reload the page from the live web and inject the same error in that load. Then, we can check whether the differences between the live page and its archived copy reduce. However, the live page on the web will likely differ from the version originally crawled; the page may even no longer exist. Loading the archived copy in proxy-mode (§2), which is unaffected by the archive's rewrites, is also not always an option since not every archival system supports this. Moreover, reloading every page once for each runtime error observed on the page will impose significant overhead; in our dataset, we observe 2.1 additional errors on average per archived page which violates fidelity.

Therefore, FIDEX attempts to evaluate the impact of every runtime error by correlating it with the JS writes missing in the load of the archived page. Recall that FIDEX's comparison of layout trees from the original and archived versions of a page reveals the JS writes that were executed in the former load but not the latter. For each such missing JS write, FIDEX checks whether that write was not executed because the error prematurely terminated the execution of the script in which the error is thrown. It checks if any frame in the error's stack causally precedes any frame in the write's stack.

As shown in the example in Figure 7, FIDEX tracks two kinds of precedence. On the one hand, within the same scope, an earlier line causally precedes a later line. On the other hand, if a script initiates the fetch of another script, then all lines in the former prior to the fetch precede all the code

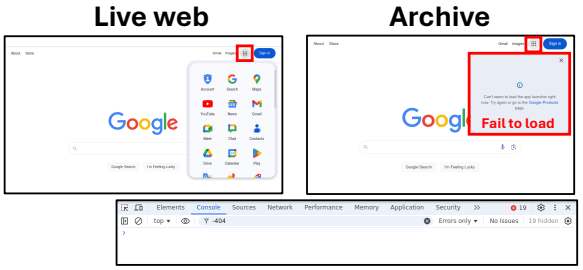


Figure 8: **On the archived copy of Google's homepage, the Google Apps dropdown fails to load when the associated icon is clicked, even though no error messages are shown in the browser console (failed fetches are filtered out).**

```
1  const originalWindow = window;
2  // Override of the getter for property "prop" on
       element "target"
3  function overrideWindowGet(target, prop) {
4      Erroneous overrides ...
5  }
6  window = new Proxy(window, {
7    get: function (target, prop) {
8      const retval = overrideWindowGet(target, prop);
9      if (proxyToObj(retval)!== originalWindow[prop])){
10       console.log("Spec violation for window.get)
11     }
12     return retval;
13 })
```

Figure 9: **Example of FIDEX detecting silent errors. Code highlighted in green is added by FIDEX to check if overridden APIs satisfy their specification.**

in the latter. FIDEX tracks these dependencies when a page is crawled. When it later evaluates the quality of the archived copy, FIDEX transitively applies these stored dependencies to determine which of the new runtime errors causally precede any missing JS write.

4.2.2 Detect impactful execution divergence

Erroneous rewriting can also lead to fidelity issues without resulting in any runtime errors; Figure 8 shows an example. When exceptions are caught and handled by `catch` blocks or rewritten functions return incorrect values, though execution continues, it diverges from expected behavior.

Discover silent errors. Unlike runtime errors, it is not straightforward to detect errors that are not exported by the browser. For instance, using the browser's debugger to set a breakpoint on every caught exception would significantly slow down crawling of pages. To minimize overhead, FIDEX instruments only those functions that are dynamically rewritten; after all, in our setting, any divergence in the execution of scripts must stem from the archive's rewrites. FIDEX wraps every overridden function in a try-catch block to detect when its incorrect execution results in an exception that is caught outside the function.[1] However, if an overridden function does not result in an exception, it is hard to identify when it executes incorrectly; after all, the purpose of rewriting a function is to modify its behavior.

[1]Typically, archival systems embed into every page a single script (e.g., wombat.js) which contains all the overridden Web APIs.

We observe that dynamic rewrites of web APIs [21] only intend to ensure that any dependent resource fetches are directed to the archive. The overridden APIs must still satisfy their specified interface. For example, during the load of an archived page, the `Window` object needs to be overridden using JS Proxy to ensure that the resources added to `Window` are fetched from the archive. Despite this, the getter for `Window.parent` must satisfy its specification of enabling access to the parent window of the current frame (if available). The only impact of the `Window` object being overridden ought to be that the `Window.parent` getter should return a proxified version of its parent frame, rather than the original.

Hence, as shown in Figure 9, FIDEX discovers silent errors by adding specification checks to overridden functions. FIDEX attaches to every overridden (proxified) object a reference to the corresponding native object. For values returned by rewritten getters, FIDEX checks that the attached native object still follows the API specification. Overrides on setters or fetch-related APIs are harder to verify. We have found some common issues include setting incorrect values (e.g., setter on iframe's `srcdoc` is incorrectly scoped) or returning incorrect values (e.g., fetched JSON files are identified as JS, and incorrectly rewritten). FIDEX also adds corresponding checks for them.

Validate impact of silent errors. Unlike runtime errors, silent errors do not halt execution, and may have no impact on the page. For example, failing to get the correct parent frame will prevent the handler for cross-frame messages from being triggered in reCAPTCHA. But, its failure has no impact on the user.

Therefore, FIDEX has to validate the effect of every silent error by comparing how the page looks with and without the error. FIDEX does so by selectively reverting the overridden function/object in which the error is observed to its original implementation. FIDEX reloads the page after this modification. If the number of differences in the layout tree between the original page and the archived copy reduce, FIDEX can confirm that the silent error is indeed a cause for the violation of fidelity.

5 Implementation

Our implementation of FIDEX includes ∼5.8K LOC in Python and ∼3.1K LOC in JavaScript.

Browser-based crawling. Browser-based crawlers used by web archives, such as Browsertrix [4], have substantial performance drawbacks [37]. Therefore, we implement a relatively lightweight browser-based crawler which uses Webrecorder, a Chrome extension for recording web pages.

Tracking JavaScript writes. FIDEX tracks JavaScript writes by overriding web APIs that modify DOM nodes, styles, and classes. FIDEX uses Puppeteer [15] to automate browser-based page loads, and leverages Puppeteer's `evaluateOnNewDocument` method to inject the overridden implementations right after any document is initialized. By doing so before archival systems override these APIs, FIDEX ensures that it tracks native calls to APIs that adhere to their specifications. To prevent FIDEX's instrumentation from interfering with the archive's rewrites, we only record the target element and arguments when overriding any method, but never modify them.

Relating runtime errors to missing JS writes. FIDEX uses Esprima [8] and BeautifulSoup [3] to parse JavaScript from each stack frame into an abstract syntax tree (AST). FIDEX strips off rewrites added by the archival system to align the AST between the live and archived pages. Then, FIDEX uses static analysis on aligned ASTs to check for precedence relationships between stack frames.

Maintainability and generalizability. To ensure that FIDEX is easy to maintain as web APIs that write to the DOM and CSSOM evolve, we categorize all such APIs into two types: APIs that write by setting an element's attributes, and APIs that write by calling an element's methods. FIDEX implements a general overriding function for each type. To track any new API, we only need to add it to the list of APIs for the overriding function of the corresponding category.

FIDEX will need to be modified to make it compatible with different archival systems as they vary in how they rewrite scripts. It will also be necessary to add new specification checks to detect silent errors. However, most of FIDEX's implementation is independent of precisely how and what a specific archival system chooses to rewrite. For example, instrumenting page loads to capture layout trees and JS writes is only dependent on the browser used. The code for comparing layout trees can also be reused.

6 Evaluation

We evaluate FIDEX from three perspectives: 1) its ability to correctly flag pages that exhibit fidelity issues, 2) its ability to pinpoint the erroneous rewrites that lead to fidelity violations, and 3) its efficiency in all aspects of its operation. Rather than focusing on any one archive (e.g., the Internet Archive's Wayback Machine), we evaluate FIDEX on version 2.9.0 (the latest version in Sep 2025) of pywb [23], the software most widely used by web archives [14] to statically and dynamically rewrite references to resources in crawled pages. pywb is fully open-source, allowing us to instrument it to capture JavaScript execution during both recording and replay. To examine FIDEX's generalizability, we also extend it to work with an alternate system for serving archived web pages, ReplayWeb.page [24].

The key takeaways from our evaluation are:

- Out of 80K pages sampled from sites in the Tranco list, FIDEX finds that the archived copies of 15% of pages violate fidelity, with most of these issues contributing significantly to the page's visual content. FIDEX's precision in identifying fidelity violations is 90%, whereas relying on

	More Errors	Screenshot	Layout tree	FIDEX
Positives	34,200	55,392	14,094	12,019
Negatives	45,800	24,608	65,906	67,981

Table 1: **No. of positives (detected fidelity violations) and negatives classified by each approach out of 80K archived pages.**

screenshots or browser-reported errors for the same purpose results in a false positive rate of at least 65%.

- FIDEX pinpoints the erroneous rewrites on 64% of the pages where it detects fidelity violations. 30% of the pinpointed errors correspond to cases where execution diverges without resulting in any exceptions.
- The top 4 most common errors pinpointed by FIDEX account for $>$ 60% of all pinpointed errors. We worked with the developers of pywb to fix the bugs in pywb which result in these errors. These bug fixes reduced the fraction of archived copies with fidelity violations from 15% to 9%.
- FIDEX reduces crawling throughput by less than 5%, which is lower than the overhead added by taking screenshots when crawling pages.

6.1 Dataset

To gather a representative set of pages from the web, we start by sampling 20K sites from Tranco [50]: 2K from ranks [1–10K], 3K from ranks [10K–100K], 5K from ranks [100K–500K], and 10K from ranks [500K–1M]. For each site, we query the Wayback Machine and select at random 4 non-home pages that exist today. In combination with the home-pages, this gives us 100K pages in total.

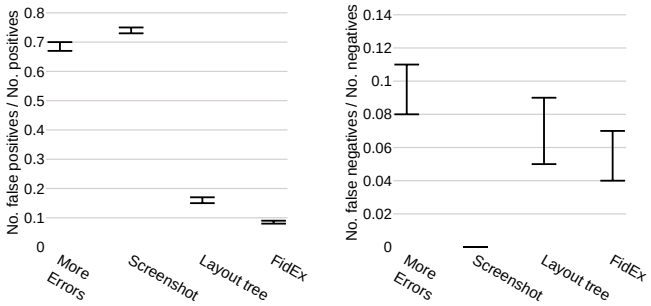
As mentioned in Section 4, we filter archived pages which violate fidelity for reasons unrelated to rewriting errors. For every archived page in our corpus, we load it thrice using pywb's proxy-mode. Both while crawling a page and when we replay its copy, we trigger up to 20 interaction groups on the page.[2] We filter out pages wherein, in any load using proxy-mode, the initial page load or any subsequent interaction does not match the crawled page. We use FIDEX's fidelity violation detection for this comparison. We are left with 83K pages, of which we sample 80K for our study.

6.2 Fidelity violation detection

First, we evaluate FIDEX's ability to detect fidelity violations. We compare it to three baselines.

- *More Errors*. In the first baseline, we log all runtime errors and declare that an archived copy violates fidelity if any additional errors are observed when loading it, compared to the errors seen when the page was crawled.
- *Screenshot*. Akin to Browsertrix's quality assurance [17], we compare the load of an archived page to its original crawl using screenshots. We ensure that our implementation closely mimics the methodology used in Browsertrix's open-source code.

[2]We group interactions by their class names and trigger one each from the first 20 groups; 80% of pages have less than 20 interaction groups.



(a) (b)

Figure 10: FIDEX **versus three baselines, with respect to (a) false positives and (b) false negatives. The whiskers show the false positive/negative rate assessed by either checker.**

- *Layout Tree*. Similar to prior work [33, 34] which uses a structured representation of every page, we compare the layout trees between a crawled page and its archived copy.

In all cases, we compare the original page and its archived copy both after the initial load and after every interaction. For every baseline, we record its chosen representation of the page at the same time when FIDEX does so (§4.1.2).

As shown in Table 1, FIDEX determines that 12K (i.e., 15%) of the 80K archived pages violate fidelity. About two-thirds of these correspond to visual mismatches between the original page and its archived copy. In the remaining, some functionality on the page that ought to work on the archived copy is broken. There are more cases in the former category because, when pywb's rewriting of a page's source breaks some interactions, it typically also affects the initial page load. For instance, many pages include a hamburger icon which, when clicked, displays a menu bar. On pages where pywb's rewrites render such clicks dysfunctional, we find that the icon often does not even get displayed.

The number of pages that FIDEX deems violate fidelity is lower compared to all other baselines because of its better accuracy. Given the lack of any ground truth assessments, we compare the accuracy of all methods as follows. For every method, we sample 100 pages each that it declares to be positives (with fidelity issues) and negatives (without fidelity issues). We manually inspect the sampled archived pages to identify false positives and false negatives. Two of this paper's authors independently cross-checked each sample. For each positive, we check if the archived copy indeed has any missing content or broken functionality. For each negative, we spent at least a minute looking for differences between the archived copy and the original page. We found that the two checkers are largely in agreement in their assessments. The discrepancies are mainly due to minor differences between the live and archived pages that are difficult to notice (e.g., a small button missing at the bottom of the page).

False positives. As shown in Figure 10(a), across all four approaches, FIDEX has the lowest false positive rate of 8–9%. *More Errors* flags differences which do not impact the user, e.g., errors in unused libraries, tracking scripts, and

third-party logins. Whereas, *Screenshot* and *Layout Tree* are unable to account for how dynamic components may be rendered differently in the loads of live and archived pages.

Like Browsertrix, if we record the page's state as soon as the browser's network activity ceases, then FIDEX's false positive rate increases significantly to 26–27%. Thus, the time at which FIDEX records the layout tree and associated JS writes is crucial. However, FIDEX falls short of achieving zero false positives because, on some pages, it takes up to a minute before dynamism kicks in. This is much longer than the timeout that FIDEX sets for each pending promise, to prevent itself from waiting forever. Moreover, on pages which add elements forever (e.g., infinite scrolls), the layout tree can differ based on the time of capture.

False negatives. Since we capture screenshots after FIDEX waits to observe all dynamism, we observe that there are no false negatives with *Screenshot*. *More Errors* has more false negatives than FIDEX because many violations of fidelity do not result in any explicit errors. The 4 false negatives for FIDEX identified by both checkers are primarily caused by the incomplete tracking of layout information in our current implementation. Again, the timing of when FIDEX captures a page's state matters; if we mimic Browsertrix's timing, FIDEX's false negative rate increases to 6–10%.

With *Screenshot* and *Layout Tree*, instead of looking for an exact match between the original and archived pages, we could check if the similarity is greater than a threshold. However, we have not found any good threshold for reducing false positives without a significant impact on false negatives. For example, for *Screenshot*, a similarity threshold of 99% offers the best F1-score and reduces the false positive rate by 10%. But, the false negative rate increases by 5%.

Impact of fidelity issues. Next, we examine the significance of the fidelity violations identified by FIDEX. We compute the area of the visual differences between the live and archived pages by collecting all leaf nodes that differ between the two layout trees. For each of these leaf nodes, we compute the relative location on the page normalized to the page's overall dimensions.

We find that, on the median archived page, the missing elements span a reasonably large area of 48K pixels. We also find that most of the content missing on the archived page appears near the top left corner of the original page. These results highlight the significance of the fidelity issues detected by FIDEX as users are more likely to perceive the impact if the area that differs is large and is closer to the top left of the page. See Appendix A for more details.

Our manual examination of the pages on which fidelity is violated reveals that common visual fidelity issues include images that have been archived but are not rendered, incorrect styling of page content, and missing text. Functionally, the most common problems are navigation bars being rendered dysfunctional, and information that is supposed to pop up when an element is clicked does not.

6.3 Pinpointing of erroneous rewrites

Next, we use FIDEX to pinpoint the root cause for each fidelity violation that it identified. FIDEX pinpoints what it deems to be erroneous rewrites for 70%[3] of these archived pages. Roughly 30% of these are cases where the browser reports no additional errors when loading the archived page.

Validation of pinpointed errors. To validate how many of the erroneous rewrites pinpointed by FIDEX are indeed causes for the observed fidelity violations, we evaluate if the difference between the original page and its archived copy reduces either by selectively introducing the error in the former or eliminating it in the latter. For any erroneous rewrite that leads to a runtime error, we use pywb's proxy-mode to load the archived page without any rewriting but after injecting the error into the script where it occurred. For silent errors, FIDEX already validates them as part of its methodology for pinpointing erroneous rewrites (§4.2.2).

Since there are no existing methods to pinpoint erroneous rewriting in archival systems, we compare FIDEX with the baseline of *More Errors* described in the last section. This baseline simulates how developers would normally debug if the archived copy is problematic: by inspecting every additional error observed in the load of the archived copy.

We find that 92% of the erroneous rewrites pinpointed by FIDEX indeed lead to fidelity violations. In contrast, with *More Errors*, only 44% of the additional errors contribute to any fidelity issues. FIDEX greatly improves the accuracy of pinpointing by reducing the number of errors that developers need to investigate by more than 80%, while shining light on incorrect rewrites that do not result in any errors reported by the browser.

Clustering of pinpointed errors. Although FIDEX is able to pinpoint root causes for thousands of archived copies that violate fidelity, it cannot automatically fix erroneous rewrites. Since it is impractical for developers to look at FIDEX's output for thousands of pages, we group pinpointed errors that result from the same rewriting mechanism. For this, we leverage the error messages associated with runtime errors. We follow JavaScript error reference [10] to get a list of error message templates, and group errors that follow the same template but differ only in the page-specific variable. For silent errors, since FIDEX adds specification checking to overridden methods, it logs a a message which describes the observed problem. We simply cluster such errors based on our customized error messages.

Figure 11 shows a heavy-tailed distribution: the 4 largest error groups account for more than 60% of the 10,509 errors pinpointed by FIDEX. On the other hand, we find that there is a long tail of esoteric issues. For example, 295 groups correspond to erroneous rewrites seen on no more than 5 pages.

[3] We expect this percentage to be less than 100 because, as observed previously, some of the fidelity violations flagged by FIDEX are false positives.

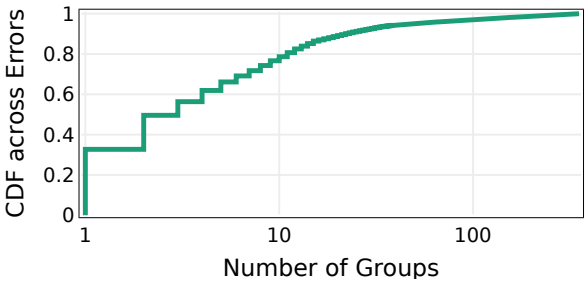


Figure 11: **Number of pinpointed errors for each clustered error group.**

As the web and systems for archiving it evolve, we expect that some of these issues may become more prevalent.

6.4 Fixing commonly pinpointed errors

Over the course of several months, we worked with the developers of pywb to fix the bugs in pywb which result in the 4 most common errors pinpointed by FIDEX as the root cause for fidelity violations.

- **Syntax errors:** The most frequent source of fidelity violations is syntax errors triggered by invalid keywords (e.g., `arguments`) in strict mode. Since ECMAScript modules (ESM) [27] operate in strict mode by default, pywb's inability to detect them led to incorrect rewrites. We fixed the problem by detecting ESM and applying a separate rewrite method that avoids syntax errors while preserving the intended rewriting effects.
- **Window parent/top:** The second most common issue involves silent errors where `Window.top` and `Window.parent` fail to return the correct parent frame. Under frameless mode (used by the Wayback Machine), the check for whether the current frame is the top frame always returns true, causing incorrect `top/parent` values. We resolve this by correcting the frame-check logic.
- **Cross-origin frame access:** The third category of common errors arises when rewritten attribute accesses are blocked (silently) in cross-origin frames, such as `srcdoc`. We fix this by removing unnecessary cross-frame attribute accesses in rewrites and relaxing the safety policy to allow legitimate accesses.
- **Reference errors:** Lastly, we address reference errors caused by scoping of global variables after rewrites. We parse each JavaScript file into an Abstract Syntax Tree (AST) and replace all global variable declarations with `var`, ensuring they remain in the global scope.

For each of the 4 target errors, we evaluate the effectiveness of our fix on 400 archived pages affected by that error. As we see in Figure 12, all 4 fixes ensure that pywb is now able to serve more than 60% of the previously affected archived copies without any loss in fidelity. For an additional 11–39%, our fixes resolve the target errors but some new errors prevent full recovery. In 0–12% of cases, our fixes fail

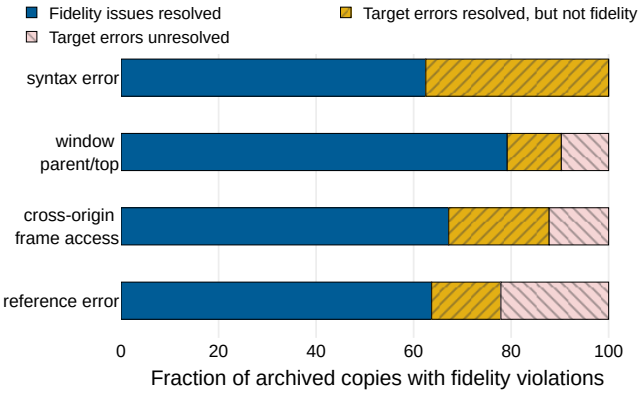


Figure 12: **Breakdown of the impact of our modifications to pywb on 400 low-fidelity archived copies.**

to resolve the target errors, which we attribute to factors such as incomplete fixes (e.g., AST parsing failures), undetected silent errors, or inaccuracies in FIDEX's diagnosis.

We also evaluated the impact of our modifications to pywb on 400 archived copies which it was already able to serve without any issues. FIDEX flagged fidelity violations in 3 of these 400 pages. However, manual inspection revealed that all of these were false positives.

Finally, we evaluated our new version of pywb on our full dataset of 80K pages. The fraction of archived pages on which FIDEX identified a fidelity violation reduced from 15% to 9%. For example, both the problematic cases from Figure 1 are now resolved.

6.5 Efficiency

We evaluate FIDEX's efficiency from three perspectives; we list the takeaways here, and defer the details to Appendix B.

- **Crawling overhead**: Compared to a baseline crawler, FIDEX's tracking of JS writes to the DOM and CSSOM reduces crawling throughput by 5%. However, FIDEX still incurs lower overhead compared to taking screenshots, which degrades throughput by 13% compared to the baseline.
- **Runtime for fidelity violation detection.** For the median page, FIDEX takes a median of 19.8 seconds to determine whether it violates fidelity. This latency, which includes the time to load the archived copy, is similar to the median of 20.4 seconds with screenshot-based detection.
- **Runtime for pinpointing.** FIDEX takes a median latency of only 1 second to identify which of the runtime errors in the load of an archived page are responsible for observed fidelity violations. For silent errors, FIDEX takes a median of 17 seconds to pinpoint the erroneous rewrites if it is able to do so, and 24 seconds if it fails to pinpoint anything.

6.6 Generalizability

Lastly, we evaluate the broader applicability of FIDEX beyond fidelity issues introduced by pywb on pages archived from sites in the Tranco list.

Alternate datasets. We run FIDEX on pages derived from two datasets whose preservation is important.

- The End of Term (EOT) dataset [49] captures a snapshot of every publicly available federal website once every four years. The intent is to document changes in the web presence of the Executive Branch of the US government at the time of presidential transition.
- The Collaborative ART Archive (CARTA) [1] dataset includes at-risk web-based art materials captured collaboratively by the Internet Archive and the New York Arts Resources Consortium (NYARC). Pages in this dataset have rich visual and interactive content.

From either dataset, we randomly sample 2,000 pages that exist on the web today and pass our filtration criteria (§6.1).

FIDEX finds fidelity violations on 12% of the pages from EOT and 11% of the pages from CARTA. Fidelity issues include both missing content (e.g., on the archived copy of the Smithsonian National Postal Museum's blog [18], images for individual blog posts fail to load) and broken functionality (e.g., on NASA's page about climate change [6], switching from the default tab about 'Evidence' to the tabs about 'Causes' and 'Effects' does not work). For pages with fidelity violations, FIDEX is able to pinpoint root causes on 81% and 72% of the pages, respectively, from EOT and CARTA. These numbers are close to the corresponding fractions seen in our dataset from Tranco.

Alternate archival system. To understand whether other archival systems too suffer from fidelity issues, we use FIDEX to examine pages archived with ReplayWeb.page [24], a serverless archival system which runs directly in the browser. We made a number of modifications to our implementation of FIDEX in order to account for the differences between ReplayWeb.page and pywb. These differences include ReplayWeb.page's reliance on service workers to serve resources, its loading of any archived page within an iframe, and different rewriting outcomes on JavaScript.

We use FIDEX to evaluate the fidelity with which ReplayWeb.page serves 2,000 archived pages. FIDEX finds evidence of fidelity violations on 10% of these pages. 42% of the pages served with poor fidelity by ReplayWeb.page were also flagged by FIDEX when these pages were loaded using pywb, but 56% of the problems were unique to ReplayWeb.page. This result implies that ReplayWeb.page has many bugs that are not in pywb.

Because our implementation of FIDEX that is compatible with ReplayWeb.page is not yet as mature as our implementation for pywb, FIDEX is able to pinpoint erroneous rewrites for only 40% of the archived pages with poor fidelity.

7 Related work

Web application testing. To detect cross-browser incompatibility (XBI) of web applications, prior work has employed both DOM and visual comparisons [32, 34, 56], and also compared the relative layouts across browsers [33]. An alternate, but related, line of work has studied automatic detection and localization of HTML presentation failures – discrepancies between the actual rendering of a page and its intended appearance – using image processing and computer vision techniques [43–45, 60]. The primary shortcoming in all of the above work which FIDEX addresses is the ability to accurately compare multiple loads of modern web pages which have JS-controlled dynamic components. Moreover, going beyond detecting a mismatch between loads, FIDEX also helps developers diagnose and fix the bugs in archival systems which cause the mismatch.

Fault localization. To identify the root cause of buggy programs, prior work has largely used two types of approaches. One class analyzes crash stack traces or bug reports to localize problems [53, 54, 62, 63]. Another relies on tests to evaluate program behavior and flag suspicious code, using spectrum-based [28, 38], mutation-based [48, 61], or ML-based [41,42,59] fault localization. In our setting, the former class of techniques would be thrown askew by the fact that many runtime errors are unrelated to fidelity violations. On the other hand, each problematic archived copy effectively serves as a standalone test case, leaving spectrum-based and mutation-based methods without enough behavioral diversity to accurately identify suspicious lines of code.

JavaScript record and replay. Beyond web archives, recording and replaying JavaScripts has been widely used for debugging and performance evaluation [30,31,46,52,55]. In these systems, JavaScript code is not rewritten after crawling. Instead, the browser is configured to run in an environment wherein, even though the browser requests the page's resources using their original URLs, these requests are directed to the local copy. In contrast, web archives serve archived copies of pages after rewriting JavaScript; as we showed, this often affects their behavior.

8 Conclusion

Web archives devote a significant amount of person hours and infrastructure to crawl and preserve pages from the web. It is a shame, then, that the widespread use of JavaScript on the web hampers their ability to serve archived pages with high fidelity. In this work, we showed how to more reliably identify fidelity violations than feasible with existing methods and help inform the diagnosis of the bugs in archival systems that cause these problems.

Acknowledgements

We thank the anonymous NSDI reviewers and our shepherd, Michael Wei, for their feedback. We are also grateful to Sumitra Duncan, Ilya Kremer, Mark Phillips, and Tessa Walsh for their input and guidance. This work was supported in part by grants from the National Science Foundation (CNS-2403432) and the Institute of Museum and Library Services (LG-254829-OLS-23).

References

[1] About the collaborative art archive (CARTA). https://carta.archive-it.org/.

[2] Archive.org's wayback machine is legit legal evidence, US appeals court judges rule. https://www.theregister.com/2018/09/04/wayback_machine_legit/.

[3] beautifulsoup4 · PyPI. https://pypi.org/project/beautifulsoup4/.

[4] Browsertrix crawler. https://github.com/webrecorder/browsertrix-crawler.

[5] Brozzler. https://github.com/internetarchive/brozzler.

[6] Climate change - NASA science. https://science.nasa.gov/climate-change/.

[7] Conifer. https://conifer.rhizome.org/.

[8] Esprima. https://esprima.org/.

[9] Federal circuit takes judicial notice of wayback machine evidence of prior art. https://www.jdsupra.com/legalnews/federal-circuit-takes-judicial-notice-2434231/.

[10] Javascript error reference - JavaScript — MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Errors.

[11] Library of congress. https://www.loc.gov/web-archives/.

[12] More than 9 million broken links on Wikipedia are now rescued. https://blog.archive.org/2018/10/01/more-than-9-million-broken-links-on-wikipedia-are-now-rescued/.

[13] Perma.cc. https://perma.cc/.

[14] Playback - IIPC. https://netpreserve.org/web-archiving/playback/.

[15] Puppeteer. https://pptr.dev/.

[16] Render-tree construction, layout, and paint — articles — web.dev. https://web.dev/articles/critical-rendering-path/render-tree-construction.

[17] Review crawl quality - Browsertrix docs. https://docs.browsertrix.com/user-guide/review/.

[18] Smithsonian National Postal Museum. https://postalmuseum.si.edu/blog.

[19] The WARC format 1.1. https://iipc.github.io/warc-specifications/specifications/warc-format/warc-1.1/.

[20] Wayback machine. https://web.archive.org/.

[21] Web APIs — MDN. https://developer.mozilla.org/en-US/docs/Web/API.

[22] Web archiving - IIPC. https://netpreserve.org/web-archiving/.

[23] webrecorder/pywb: Core python web archiving toolkit for replay and recording of web archives. https://github.com/webrecorder/pywb.

[24] webrecorder/replayweb.page: Serverless replay of web archives directly in the browser. https://github.com/webrecorder/replayweb.page.

[25] webrecorder/wombat: Wombat.js client-side rewriting library. https://github.com/webrecorder/wombat.

[26] Cloudflare and the Wayback Machine, joining forces for a more reliable web. https://blog.archive.org/2020/09/17/internet-archive-partners-with-cloudflare-to-help-make-the-web-more-useful-and-reliable/, 2020.

[27] Javascript modules - JavaScript — MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules, 2025.

[28] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. An evaluation of similarity coefficients for software fault localization. In *Pacific Rim International Symposium on Dependable Computing*, 2006.

[29] N. Brügger and R. Schroeder. *The web as history: Using web archives to understand the past and the present*. UCL Press, 2017.

[30] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *ACM Symposium on User Interface Software and Technology*, 2013.

[31] G. Candea and S. Andrica. WaRR: A tool for high-fidelity web application record and replay. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2011.

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

[33] S. R. Choudhary, M. R. Prasad, and A. Orso. X-pert: Accurate identification of cross-browser issues in web applications. In *IEEE/ACM International Conference on Software Engineering*, 2013.

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

[35] D. Fetterly, M. Manasse, M. Najork, and J. L. Wiener. A large-scale study of the evolution of web pages. *Software: Practice and Experience*, 34(2):213–237, 2004.

[36] A. Goel, J. Zhu, R. Netravali, and H. V. Madhyastha. Jawa: Web archival in the era of JavaScript. In *USENIX Symposium on Operating Systems Design and Implementation*, 2022.

[37] A. Goel, J. Zhu, R. Netravali, and H. V. Madhyastha. Sprinter: Speeding up High-Fidelity crawling of the modern web. In *USENIX Symposium on Networked Systems Design and Implementation*, 2024.

[38] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *IEEE/ACM International Conference on Automated Software Engineering*, 2005.

[39] S. M. Jones, H. Van de Sompel, H. Shankar, M. Klein, R. Tobin, and C. Grover. Scholarly context adrift: Three out of four URI references lead to changed content. *PloS one*, 2016.

[40] W. Koehler. Web page change and persistence—a four-year longitudinal study. *Journal of the American society for information science and technology*, 53(2):162–171, 2002.

[41] X. Li, W. Li, Y. Zhang, and L. Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.

[42] Y. Li, S. Wang, and T. Nguyen. Fault localization with code coverage representation learning. In *IEEE/ACM International Conference on Software Engineering*, 2021.

[43] S. Mahajan and W. G. Halfond. Finding html presentation failures using image comparison techniques. In *ACM/IEEE International Conference on Automated Software Engineering*, 2014.

[44] S. Mahajan and W. G. Halfond. Websee: A tool for debugging html presentation failures. In *IEEE International Conference on Software Testing, Verification and Validation*, 2015.

[45] S. Mahajan and W. G. J. Halfond. Detection and localization of html presentation failures using computer vision-based techniques. In *IEEE International Conference on Software Testing, Verification and Validation*, 2015.

[46] J. W. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic capture and replay for javascript applications. In *USENIX Symposium on Networked Systems Design and Implementation*, 2010.

[47] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *USENIX Annual Technical Conference*, 2015.

[48] M. Papadakis and Y. Le Traon. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability*, 25(5-7):605–628, 2015.

[49] M. E. Phillips, K. K. Phillips, and S. Alam. End of term web archive dataset: Longitudinal web archive of. gov and. mil domains. In *ACM/IEEE Joint Conference on Digital Libraries*, 2023.

[50] V. L. Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. *arXiv preprint arXiv:1806.01156*, 2018.

[51] S. Rhodes. Breaking down link rot: The Chesapeake project legal information archive's examination of URL stability. *Law Library Journal*, 102:581, 2010.

[52] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of javascript benchmarks. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2011.

[53] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *IEEE/ACM International Conference on Automated Software Engineering*, 2013.

[54] A. Schroter, A. Schröter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? In *IEEE International Working Conference on Mining Software Repositories*, 2010.

[55] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2013.

[56] S. Song, J. Hur, S. Kim, P. Rogers, and B. Lee. R2z2: Detecting rendering regressions in web browsers through differential fuzz testing. In *IEEE/ACM International Conference on Software Engineering*, 2022.

[57] D. Spinellis. The decay and failures of web references. *Communications of the ACM*, 46(1):71–77, 2003.

[58] M. Toyoda and M. Kitsuregawa. Extracting evolution of web communities from a series of web archives. In *ACM Conference on Hypertext and Hypermedia*, 2003.

[59] A. Z. Yang, C. Le Goues, R. Martins, and V. Hellendoorn. Large language models for test-free fault localization. In *IEEE/ACM International Conference on Software Engineering*, 2024.

[60] X. Yang, W. Liu, H. Lin, Z. Li, F. Qian, X. Wang, Y. Liu, and T. Xu. Visual-aware testing and debugging for web performance optimization. In *ACM Web Conference*, pages 2948–2959, 2023.

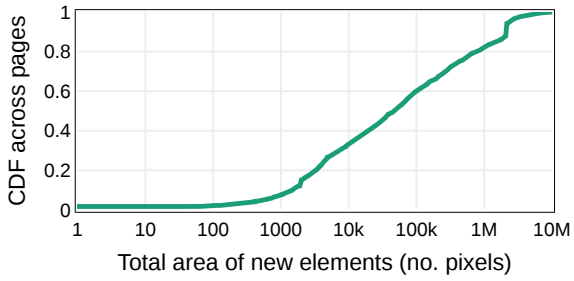
[61] L. Zhang, L. Zhang, and S. Khurshid. Injecting mechanical faults to localize developer faults for evolving software. *ACM SIGPLAN Notices*, 48(10):765–784, 2013.

[62] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *IEEE/ACM International Conference on Software Engineering*, 2012.

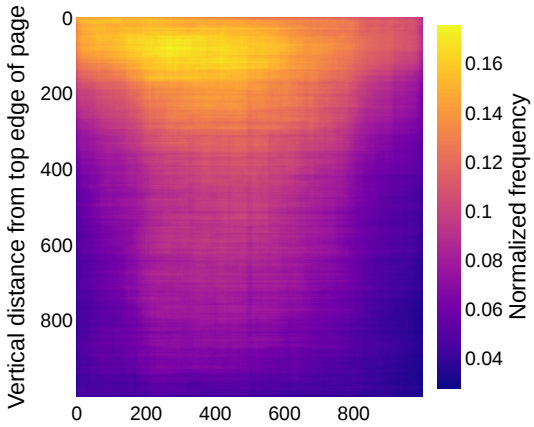
[63] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618–643, 2010.

[64] J. L. Zittrain, J. Bowers, and C. Stanton. The paper of record meets an ephemeral web: An examination of linkrot and content drift within The New York Times. *Available at SSRN 3833133*, 2021.

[65] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, 47(2):332–347, 2019.



(a)



(b)

Figure 13: **Impact of fidelity issues detected by FIDEX: (a) Distribution of the area (in pixels) accounted for by the elements missing from the archived page. (b) A heat map of the locations on the page where the missing elements reside on the original page, normalized to a 1000x1000 square.**

A Impact of fidelity violations

Figure 13(a) shows that the elements missing on the archived page often span a reasonably large area, with an area of 48K pixels on the median page. Pages with differences in the range 1K–10K pixels are typically missing small components such as buttons and icons. In the range 10K–100K, pages are usually missing an iframe (e.g., Trustpilot reviews) or a notification that spans the page width. Pages with differences of over 100K pixels are usually missing multiple components. For example, the main section of the page is missing, together with a change in the page's style. These are cases where errors that occur early in the page load preclude many subsequent executions that build the page.

Figure 13(b) shows that most of the affected components appear near the top left corner of the page. This highlights the significance of the fidelity issues detected by FIDEX as web developers place important content closer to the top and left margins of the page.

B Efficiency

Crawling overhead. We run our crawlers on a server with a 32-core 2.1 GHz Intel Xeon CPU, a 1 Gbps network connection, and 128 GB memory. We launch 16 crawlers in parallel to saturate all cores. We compare the baseline crawler with no instrumentation against FIDEX's crawler. We also perform this comparison after augmenting both crawlers to capture a screenshot after the initial load and after every interaction. Capturing a screenshot when running FIDEX can help users understand the fidelity violations identified by it. Figure 14 shows the normalized crawling throughput across these crawling strategies.

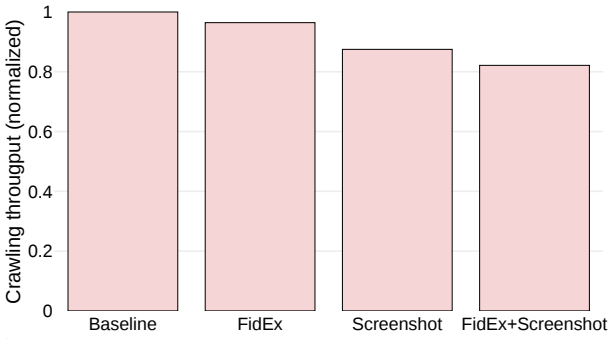
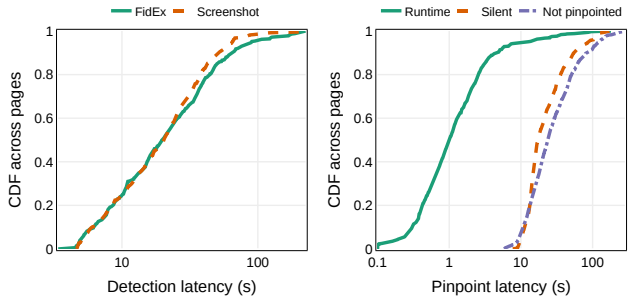


Figure 14: **Comparison of crawling throughput, normalized to that offered by baseline crawler without any instrumentation.**



(a) (b)

Figure 15: **FIDEX's runtime to (a) evaluate each archived copy's fidelity, and (b) pinpoint the root cause on every page deemed to violate fidelity. The latter separately shows when pinpointing is based on runtime errors, pinpointing is based on silent errors, or FIDEX is unable to pinpoint any erroneous rewrites.**

Runtime for fidelity violation detection. After FIDEX crawls a page and loads its archived copy, we measure how long it takes to process the information captured and compute the difference between the two layout trees. Figure 15(a) shows that, for roughly 60% of a random sample of 1,000 archived pages, FIDEX's detection latency is comparable to — or even lower than — that of the screenshot-based method. This is because the latency is largely dominated by the time needed to load an archived page and capture relevant state, for which FIDEX has lower overhead. FIDEX takes longer to process larger pages which produce bigger layout trees and more JS writes. We expect this overhead can be reduced, as our implementation of FIDEX's comparison of layout trees is not yet optimized, whereas we compare screenshots using mature, highly-optimized libraries.

Runtime for pinpointing. On a random sample of 1,000

pages flagged by FIDEX as violating fidelity, Figure 15(b) shows the distribution of latency for FIDEX to pinpoint pywb's problematic rewrites. FIDEX requires a median latency of only 1 second to pinpoint the erroneous rewrite if it results in a runtime error. Whereas, if the erroneous rewrites only cause the execution to diverge, pinpointing the cause takes longer: 17 seconds at the median. This is because FIDEX needs to reload the archived copy after selectively removing each silent error in order to validate its impact. When FIDEX fails to pinpoint any errors, it requires 24 seconds on the median page.