Jawa: Web Archival in the Era of JavaScript

Ayush Goel[1], Jingyuan Zhu[1], Ravi Netravali[2], Harsha V. Madhyastha[1]
[1]*University of Michigan*, [2]*Princeton University*

Abstract—**By repeatedly crawling and saving web pages over time, web archives (such as the Internet Archive) enable users to visit historical versions of any page. In this paper, we point out that existing web archives are not well designed to cope with the widespread presence of JavaScript on the web. Some archives store petabytes of JavaScript code, and yet many pages render incorrectly when users load them. Other archives which store the end-state of page loads (e.g., screen captures) break post-load interactions implemented in JavaScript.**

To address these problems, we present Jawa, a new design for web archives which significantly reduces the storage necessary to save modern web pages while also improving the fidelity with which archived pages are served. Key to enabling Jawa's use at scale are our observations on a) the forms of non-determinism which impair the execution of JavaScript on archived pages, and b) the ways in which JavaScript's execution fundamentally differs between live web pages and their archived copies. On a corpus of 1 million archived pages, Jawa reduces overall storage needs by 41%, when compared to the techniques currently used by the Internet Archive.

1 Introduction

URLs are brittle pointers to information on the web. Over time, a page may cease to exist at the URL where it was originally available [44, 62] or the content available at that URL might change due to the page being modified [58, 36].

Therefore, web archives play a key role in the web ecosystem, enabling users to lookup the content that existed at any particular URL at various times in the past. Web archives are used for a wide variety of use cases, such as web-data analytics, genealogical analysis, and even as legal evidence [40]. To support these uses, a number of organizations—cultural heritage institutions, national libraries, and public museums—operate web archives to ensure long-term preservation of content on the web. A recent survey estimates that there are 119 web archives in the United States alone [35].

The largest and most popular of these archives, Internet Archive (IA), has archived over 600 billion web pages to date, storing data in excess of 100 petabytes [13]. It repeatedly crawls web pages over time and saves many snapshots of every page. For every page snapshot, IA first downloads all resources (e.g., HTMLs, CSS stylesheets, JavaScripts, images) on the page). It stores these resources after rewriting all URL references to point to the copy hosted by the archive. When a user wants to later view any stored snapshot of a page, the user's browser loads the snapshot from IA in the same manner as it would load any page on the live web.

In this paper, we argue that this modus operandi no longer suffices due to the preponderance of JavaScript on modern web pages [18, 38, 53]. Specifically, the widespread use of JavaScript hinders web archives from satisfying two of their primary objectives: 1) to capture and save as much of the web as feasible, and 2) to ensure that archived page snapshots faithfully mimic the original page.

- **Higher operational costs:** First, the total number of bytes on the median web page has more than tripled over the last decade [10]. A significant contributor to this increase has been the increased usage of JavaScript. For example, across Internet Archive's copies of the home pages of 300 randomly sampled sites, we see that JavaScript accounts for 44% of the bytes on the median page in 2020, as compared to 20% in 2000 (§2). Since web archives are typically run by non-profit institutions with limited budgets, needing to store more bytes per page reduces the number of pages they can crawl and archive.
- **Poor page fidelity:** The archived copies of many JavaScript-heavy pages render with missing images and improperly laid out content (§2.1). This occurs due to the non-deterministic execution of JavaScript; when a user loads an archived copy of a page, the resource URLs requested by the user's browser can differ from those saved by the archive when it crawled the page. Consequently, the web archive returns errors for some of the requested resources. Due to the complex dependencies between the resources on a page [65, 34, 54], one failed resource fetch often has a cascading effect on the rest of the page load.

The challenge in holistically addressing both problems is that trying to reduce storage overheads by not saving some of the JavaScript found on crawled pages risks further degrading fidelity. A web archive could statically or symbolically analyze the JavaScript code on every page to identify what subset is necessary to preserve correctness in *all* potential loads of the page. However, the computational overheads of such methods [42, 48] render them impractical at the scale of a web archive, e.g., the Internet Archive crawls roughly 5000 pages per second [64]. To jointly address JavaScript's adverse impacts on storage and fidelity using computationally lightweight methods, we observe and leverage three fundamental ways in which JavaScript's execution on archived pages differs from that on the live web.

First, a significant fraction of JavaScript is dedicated to either sending user data to a page's origin servers or processing dynamically constructed server responses, e.g., to enable

users to post comments or to push notifications. Any such functionality cannot work on archived pages, and therefore, the associated code need not be stored by web archives. Fortunately, the JavaScript code on any page is typically partitioned into several files, and we find that most of the code that will be non-functional in the context of a web archive is cleanly compartmentalized into a subset of these files that exhibit identifiable patterns in their URLs. Consequently, we show that web archives can efficiently, and safely, prune unnecessary JavaScripts by relying on URL-based filters to identify and discard JavaScript source files.

Second, many lines of JavaScript code are executed only in certain control flows, e.g., when a page is loaded on a smartphone, and not on a desktop. But, among the various sources of non-determinism that dictate whether or not a specific line might get executed, some sources are absent in loads of archived page snapshots; clients maintain no state across loads and server responses for the same request URL do not vary. Moreover, a web archive should actively eliminate those sources of non-determinism which can cause clients to request different resource URLs than those crawled. Thanks to the resulting reduction in non-determinism, we find that much of the JavaScript code on an archived page will never be exercised in any load of that page, making it moot for a web archive to store such code.

Lastly, a critical use of JavaScript is to enable users to interact with a page after the page's load has completed. On live pages, identifying *all* the code used to support such interactions is generally challenging because the code that is exercised varies based on how users interact with the page. For example, the input given to a search bar determines the server's response; based on the number of search results, JavaScript for paginating the results may or may not get executed. In contrast, we find that the subset of interactions that do work on archived pages (e.g., navigational menus and image carousels) distinctly differ from those that do not with respect to the properties of the page state they access. This greatly simplifies the task of identifying the code necessary to preserve post-load interactions.

Based on our three observations, we design and implement Jawa (JavaScript-aware web archive), a system for crawling and saving web pages. Jawa enables web archives to save many more pages than they could today for the same cost, e.g., it reduces the total amount of storage necessary to store a corpus of 1 million web pages by 41%. Importantly, Jawa enables this reduction both while increasing the rate at which pages can be crawled by 39% and significantly improving the fidelity of archived pages: for the vast majority of archived pages, Jawa ensures that the page is rendered in a manner identical to how it was when the page was crawled, and all page functionality that can possibly work on an archived page does work. Source code for Jawa, including scripts to reproduce the key results in the paper, are available at https://github.com/goelayu/Jawa.

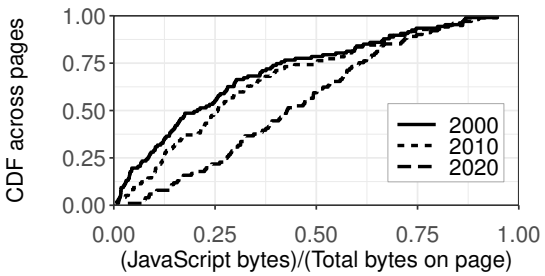


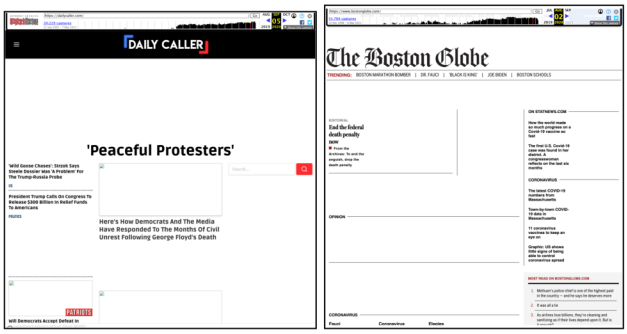
Figure 1: **Across the landing pages of 300 sites, distribution of fraction of bytes on the page accounted for by JavaScript.**

2 Background and Motivation

As mentioned earlier, the Internet Archive (IA) is the largest and most popular web archive in the world today. For every page that it crawls, IA stores all the individual resources on that page (such as HTMLs, CSS stylesheets, JavaScript files, and images) in the Web ARChival format (also known as the WARC format [22]). Client browsers can load archived pages from IA's Wayback Machine [24] in a manner identical to how they do on the live web. When the Wayback Machine receives a request for any resource, it looks up an internal index to locate the WARC record for this resource and then responds along with relevant HTTP headers. IA rewrites all resource files so that all statically embedded URLs point to IA's web servers. For URLs which are dynamically generated via JavaScript, IA rewrites them on the fly using client-side API shims.

This architecture sufficed when IA began operating two decades ago. However, the web today is very different. In particular, JavaScript (JS) has become significantly more common. For example, Figure 1 shows that JS accounts for 44% of the bytes on the median page today; up from 20% in 2000. In this section, we show that this increase in JS hinders the ability of web archives to meet their two primary objectives: 1) to crawl and capture as much of the web as possible, and 2) to preserve page fidelity, i.e., when an archived page is loaded by a user, it should ideally match the page as it was crawled, both in visual (how the page looks) and functional (user interactions supported on the page) aspects.

To support our claims, in this section (and in the rest of the paper), we consider pages from 300 sites, comprising 100 randomly chosen sites from each of three ranges from Alexa's site rankings: [1, 1000], [1000, 100K], and [100K, 1M]. Using these 300 sites, we construct two corpuses. $Corpus_{3K}$ contains one of IA's copies from September 2021 for 1 landing and 9 internal pages per site. $Corpus_{1M}$ contains 3500 page snapshots for each site out of all of IA's page snapshots from September 2020. Note that both corpuses contain a mix of old and new pages. Though both corpuses contain page snapshots which were archived in the last couple of years, many of these pages were created before then. This is because IA recrawls pages over time to track changes to page content.



(a) dailycaller.com [21] (b) bostonglobe.com [20]

Figure 2: **Examples of page snapshots loaded from IA.**

2.1 Poor fidelity due to JS non-determinism

When a user loads a web page, scripts on the page often dynamically construct the URLs for many of the resources on the page. In doing so, JS execution can leverage various sources of non-determinism: client-side state (e.g., cookies, local-storage), client-characteristics (e.g., user-agent), random number generators, etc. When a user loads an archived page, these sources of non-determinism can potentially lead to a different set of resource URLs being requested compared to what was crawled by the archive. This, in turn, leads to two significant problems.

Failed fetches. First, IA returns a resource not found error (HTTP status code 404) for all resource URLs not stored at the archive, resulting in many archived pages rendering incorrectly. Figure 2 shows two examples of screenshots of page snapshots loaded from IA. In both cases, JS code on the page dynamically constructs the URLs of images to fetch by taking into account the screen size of the client loading the page. Since our client appears to differ from IA's crawler,[1] these pages end up being rendered incorrectly.

Runtime errors. Second, the execution of many scripts halts prematurely with runtime errors, which in turn leads to more resources going unfetched. We inspect the runtime logs generated by Chrome when loading the pages in $Corpus_{3K}$; specifically, the JavaScript console log and the network log. Figure 3 compares the number of errors seen in these logs during page loads from the web and when loading snapshots of these pages archived by the IA on the same day. Loading pages from IA results in more errors of both types. The total number of bytes that went unfetched because of these failed network requests cumulated to 5% and 45% of bytes on the median and 95th percentile page respectively.

2.2 High storage overhead

The more obvious downside of more JavaScript on web pages is that it increases a web archive's storage needs. To quantify this impact, we compute the total amount of storage required to store all the pages in $Corpus_{1M}$. Across all pages, we account for storing a single copy for every unique

[1] We look at the HTTP response headers of the archived resources to gather information about the client used by IA.

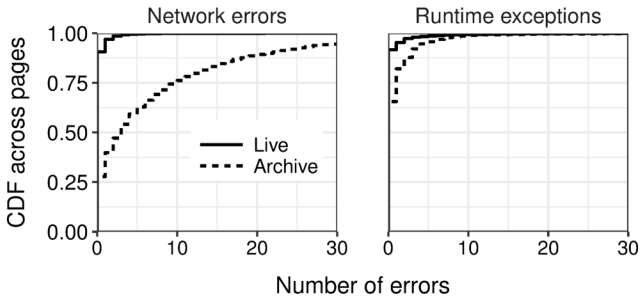


Figure 3: **Comparison of errors thrown during page loads from the web and from IA.**

(resource URL, SHA-256 content hash) combination; IA applies similar deduplication to reduce storage overheads [23]. Despite the fact that scripts are often shared across pages (e.g., JavaScript libraries like jQuery are used by many sites and pages on the same site include a common set of scripts), JS accounts for 49% of all the bytes stored; resources of all other types (HTML, CSS, images, etc.) account for the remaining 51%.

Note that these numbers account for the size of textual resources such as HTML, CSS, and JS after compression. Also note that our corpus of a million pages appears to be large enough to approximate the utility of deduplication at scale because the fraction of total bytes accounted for by JavaScript plateaus after 750K pages.

Overall, the fact that scripts roughly double the amount of storage that a web archive needs to deploy is concerning because web archives are largely reliant on donations to cover their operational expenses [7]. For example, IA spends around $18 million dollars each year in operational expenses and attributes over 60% of its earnings to donations [14]. Needing to store more bytes per page means that an archive can store fewer pages for the same cost.

2.3 Downsides of alternate archival formats

To sidestep the shortcomings of IA that we have discussed thus far, a web archive could instead store and serve the end result of any page load, thereby preempting the need for clients to execute JavaScript.

Preserving post-load interactions. One such alternate archival format is to store a screenshot of the rendered page in the PNG or PDF format, as employed by private archiving institutions like Stillio [19] and PageVault [16]. However, many pages today enable users to interact with the content on the page, and storing screenshots of pages fails to preserve these post-load interactions [56]. Web developers enable such interactions by registering event handlers while a page is being loaded; these event handlers are triggered and executed when the user later interacts with the page. For example, modern pages often include carousels or sliders to display images and tabs to group information in separate categories; see, for example, the infographics on https://www.nytimes.com/interactive/2021/world/

india-covid-cases.html. Event handlers are also used to enable users to navigate to other pages on the same site, e.g., the menu under the "Explore" button on https://www.coursera.org. Prior studies have shown that it is important to preserve such informational and navigational interactions even on archived pages [40].

We analyze the pages in *Corpus$_{3K}$* to determine how many contain interactions that should work on archived copies. Specifically, we load every page after instrumenting all scripts so that we can track all event handler registrations. We identify all event handlers which are associated with page elements whose attributes contain keywords such as menu, navbar, slider, carousel, dropdown, etc.; we consider 13 such keywords commonly associated with informational and navigational interactions. We find that 91% of the pages contained at least one such event handler.

Overhead of capturing JavaScript heap. Alternatively, client-local interactions enabled by event handlers could be preserved by storing a) every page's final rendered HTML, b) all resources referenced from this HTML (such as CSS and images), and c) the JavaScript heap, which stores custom, page-defined JavaScript state as well as native JavaScript objects [55]. However, modern browsers do not expose the entire JavaScript heap [43]; only the global scope of the heap is accessible using the "window" object. The closure scope, which is a non-global scope that is defined by any function and is accessible only by the nested functions that execute in that function's enclosed scope [51], is not accessible. This is a key roadblock because event handlers often access closure state; 47% of the pages in *Corpus$_{3K}$* contain at least one such handler (we describe how we perform the state tracking necessary to obtain this result in §4).

To access closure state, a web archive's crawler could statically analyze and rewrite the scripts on every page prior to executing them. However, we find that the combined overhead of performing the static analysis necessary to identify different scopes and running instrumented scripts inflates the time to crawl the median page in *Corpus$_{3K}$* by 2x; this overhead increases to 6x at the 99th percentile. Such computational overhead will significantly increase costs for a web archive crawling thousands of pages every second [64].

3 Overview

To overcome the adverse impacts of JavaScript on web archival, our high-level insights stem from two key differences between the loads of live and archived pages. In this section, we describe these differences and outline the challenges entailed in leveraging these differences.

3.1 Distinguishing properties of archived pages

No back-end origin server. Modern web pages include a range of functionalities which require communication with the page's origin servers, e.g., enabling users to post comments and having servers push updates to users while they are on a page. However, when a user loads an archived page snapshot, only that functionality on the page will work which can be served using the resources crawled when this snapshot was captured.

Limited sources of non-determinism. To deliver a dynamic user experience, many pages on the web adapt how they are rendered based on ① server-side state, ② client-side state (e.g., cookies, local storage), ③ client characteristics (e.g., user-agent, screen dimensions), and ④ "Date", "Random", and "Performance" APIs (we refer to these as *DRP* APIs for the sake of brevity). For example, after a script on a page fetches a JSON from the origin server, its subsequent control flow might depend on the contents of that JSON, which itself might be influenced by the contents of a client-side cookie. In loads of archived pages, the first two sources of non-determinism are absent: in response to the request for a particular resource in a specific page snapshot, a web archive will always serve the copy it fetched when crawling that snapshot; whereas, client browsers do not maintain any state across loads of archived pages.

3.2 Challenges

In order to leverage the above-mentioned differences to both improve page fidelity and reduce storage overhead in web archives, we need to answer several questions.

What are the causes of poor page fidelity? While some sources of non-determinism are absent in the loads of archived pages, the remaining sources – client characteristics, *DRP* APIs, and asynchronous execution of timer handlers and script fetches – still result in non-deterministic JS execution. Determining which of these factors is responsible for clients requesting different resource URLs than those crawled is key to eliminating failed resource fetches and the resultant runtime errors.

How to efficiently prune non-functional and unreachable code? In any page that it crawls, a web archive need not save any JS code that either relies on interactions with the page's origin servers or would never be executed in any load of the page (due to the absence of certain sources of non-determinism). One could potentially use methods like symbolic or concolic execution to perform reachability analysis and identify both unreachable code and non-functional code; the latter comprises code that is reachable from RPCs to origin servers. However, as reported in prior work [42, 49, 48], these methods for analyzing JS code are computationally expensive, requiring tens of minutes per page. Increasing the compute overheads of crawling to such a large extent would nullify any storage savings.

How to ensure code pruning does not hamper fidelity? While eliminating non-functional code reduces storage cost, doing so comes at the risk of inadvertently hurting fidelity. In particular, the code that is retained must function as it would if no code were discarded. Checking that any method identified for code elimination does preserve this property is

Goal	Observations	Section
Improve fidelity	APIs for client characteristics are the key cause for failed resource fetches	§4.1
	Differences in URLs due to *DRP* APIs can be resolved using server-side URL matching algorithms	
Prune non-functional code	Most of JS code which will not function on archived pages is in third-party source files, which can be identified based on their URLs	§4.2
	First-party scripts typically use third-party code cautiously, so that reliability of former is not dependent on availability of latter	
Prune unreachable code	*DRP* APIs typically have no impact on control flow	§4.3
	For event handlers associated with post-load interactions which work on archived pages, page state accessed is disjoint across handlers and user input does not influence control flow	

Table 1: **Overview of the main insights that influence our design of Jawa.**

non-trivial because browsers do not offer any APIs to extract runtime information that can be used to identify state dependencies between different scripts on any page.

3.3 Requirements

Based on all the considerations discussed thus far, we focus on three objectives.

- **High fidelity.** First, we seek to ensure that any archived page faithfully mimics the original page in two respects: 1) how the page is rendered, and 2) all functionality on the page which does not require communicating with the page's back-end servers works.
- **Low cost.** Second, we aim to enable a web archive to improve its coverage by reducing the amount of storage needed for any collection of page snapshots. In doing so, we seek computationally lightweight methods so as to minimize the cost overheads associated with maintaining the same rate of crawling pages as today.
- **Simplicity.** Lastly, our solutions must be simple to implement. In our discussions with the Internet Archive, they have emphasized that simplicity is key for any proposed changes to be viable in practice.

4 DESIGN

We describe our design of Jawa in three parts. We begin by describing how Jawa improves page fidelity by eliminating the sources of non-determinism which result in failed resource fetches while loading archived pages. Thereafter, we present the methods used by Jawa to identify what subset of crawled JS files need not be saved: first to eliminate non-functional code, and second to prune unreachable code while preserving post-load interactions. To enable Jawa's use at scale, the overriding principle that guides all aspects of our design is to minimize computational overheads by leveraging properties of JS typically found on the web; Table 1 provides an overview of our observations. Later (§7), we describe how a web archive which uses Jawa could potentially handle pages which do not satisfy these properties.

Analysis framework. Throughout this section, we use our custom JavaScript analysis framework (4.5K LOC) to study the properties of JavaScript found on pages in $Corpus_{3K}$. As in prior program analysis tools for JavaScript [49, 55, 38], our analysis framework first performs offline, static analysis of the JS in a page, converting each JS file into an abstract syntax tree (AST) representation. It then parses this AST to identify the different JS scope levels – local, block, closure, and global – and leverages this information to associate each JS variable to its corresponding scope. The framework also uses the AST to detect JS function invocations.

Building on these insights, our framework instruments pages with code that is triggered in each function invocation, and records the arguments to the function, all the closure and global scope variables read and written inside the function body, and the return value. Special care is taken to (1) record all accesses to the DOM, (2) track accesses of any global variable's properties via an alias, e.g., *"var a = window"* followed by a read of *"a.innerHeight"*, (3) identify DOM elements with registered event handlers and the corresponding handler functions, and (4) monitor and control the return values of browser APIs such as *"navigator.userAgent"*.

4.1 Improve fidelity by eliminating failed fetches

To ensure that users do not encounter failed resource fetches when they load archived pages, a web archive could rewrite every stored page to ensure that, when the page is loaded, the flow of execution and the return values of all browser APIs match those seen when the page was crawled.[2] If a web archive were to eliminate sources of non-determinism in this manner, we observe that fixing the schedule of execution cannot result in any loss of functionality; after all, developers of pages have no control over the client-side schedule of execution of asynchronous scripts. However, a page's developer can indeed ensure that code on the page behaves differently based on the results from browser APIs. Therefore, we seek to understand the impact of these APIs on resource URLs and eliminate only those sources of non-determinism which result in failed fetches during loads of archived pages.

Impact of different sources of non-determinism. We measure the impact of each source of non-determinism as follows. We first load our locally stored copies of all pages in $Corpus_{3K}$ with a desktop client. We then reload these pages mimicking a different client ("iPhone 6"). Mimicking a different client allows us to exercise different values of most

[2] Alternatively, a web archive could crawl every page under all possible combinations of non-determinism. Doing so is not only impractical, but would dramatically inflate compute and storage overheads.

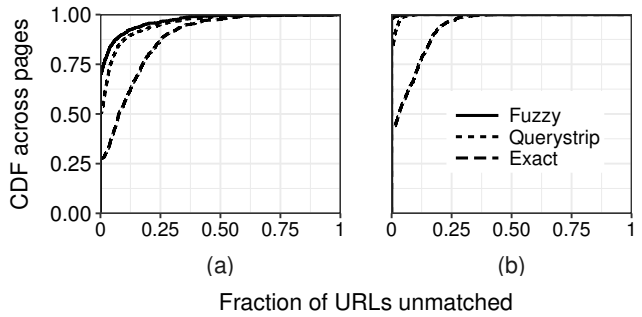


Figure 4: **For every page in *Corpus$_{3K}$*, fraction of resource requests which cannot be matched with any crawled resource. The impact of different URL matching algorithms is shown when the sources of non-determinism are (a) APIs for client characteristics as well as *DRP* APIs, and (b) only *DRP* APIs.**

client characteristics, such as user-agent, screen dimensions, and OS. We reload all pages once more, this time matching the client characteristics used in the original load.

On 72% of pages, at least one different resource URL was requested in the second load compared to the first load; these two loads differ in the values for both APIs for client characteristics and *DRP* APIs. Whereas, when comparing the third load to the first, which differ only with respect to *DRP* API values, the corresponding fraction was 52%. Note that, in both cases, even one failed resource fetch can have a cascading effect, resulting in many other resources going unfetched.

Variance in resource URLs due to non-determinism results in failed network fetches only if a web archive (like IA) expects requests from clients to specify URLs which are identical to the ones crawled. However, across loads of a page, if the same resources are being requested using different URLs, it might suffice for the web archive to employ a better algorithm to match URLs requested to those crawled.

To check if this is the case, we consider two URL matching algorithms used in prior work: ① *querystrip*, where the query string in any URL (i.e., the portion of the URL beyond the delimiter '?') is stripped before initiating a match [57], and ② *fuzzy matching*, which leverages Levenshtein distance [45] to find the best match for any given URL [26]. Querystrip relies on the fact that query strings are typically used for updating server-side state, and they do not influence the content of the response. Fuzzy matching accounts for cases where non-determinism across loads results in simple string transformations of the URLs for the same resources. In any page load, we match URLs in the order they are requested, and we match any requested URL against those crawled URLs that have not already been matched.

Figure 4(a) shows that, on many pages, a significant fraction of URLs were unmatched with both algorithms, when APIs for client characteristics were a source for diverging URLs. This is because, when client characteristics differ, often the *number* of resources fetched on the same page changes. For example, www.nytimes.com fetches the JavaScript file *player-embedded.js* on mobile clients to enable video players, whereas it fetches no such scripts on desktop clients.

Digging deeper into *DRP* APIs. In contrast, when *DRP* APIs are the only source of non-determinism, Figure 4(b) shows that either URL matching algorithm suffices to eliminate almost all failed resource fetches. However, this might be the case only because we compare two loads of every page, and the return values of *DRP* API invocations did not sufficiently differ to have an impact.

To capture the effects of all possible return values of *DRP* APIs, we turn to concolic execution [37, 61, 42], a variant of symbolic execution which executes programs concretely (rather than symbolically) while ensuring complete coverage of all control flows. We modify a prior concolic execution tool [42] to only track control flows influenced by *DRP* APIs. We then randomly sample 300 pages from *Corpus$_{3K}$* because it takes around 20 minutes per page with this tool. On all pages, *DRP* APIs had no impact on control flow. Thus, comparing any two loads of a page suffices to examine the divergence in URLs across loads due to these APIs.

Takeaways. These results influence our design of Jawa in two ways. First, we instrument all scripts on any page so that, when clients execute these scripts, all APIs for client characteristics return the same values as when the page was crawled. Compared to a thin-client model where a web archive serves requests for pages by executing page loads on behalf of users [26], our approach of letting users execute page loads on their devices reduces server-side overheads. Second, we do not need to account for any differences across loads in *DRP* APIs because the impact of these differences can be accounted for with server-side matching of requested URLs to crawled URLs.

Note that we choose to patch all invocations of client characteristic APIs, and not just the ones which influence the URLs fetched. This is because, even if a particular invocation of an API does not impact which URLs are fetched, it can impact the reachability of code which assumes that state dependent on the client's type has been setup earlier in the page load. Hence, if different API invocations return inconsistent values, this could exercise code which accesses uninitialized state, resulting in runtime errors.

4.2 Pruning non-functional code

We now turn our attention to reducing the storage overhead of JavaScript on web archives. Jawa's crawler uses two complementary approaches to take advantage of the two previously mentioned properties which distinguish archived page snapshots from pages on the web. The key consideration in both cases is to ensure that pruning any JavaScript code does not affect the execution of the remaining code.

Characteristics of non-functional code. Our first approach for pruning JavaScript code is based on two observations about the code which will not work on archived copies of pages, i.e., code which relies on clients interacting with

origin servers. First, on a typical page, we find that most of such code is compartmentalized into a few files, rather than being evenly spread across all JavaScript source files on the page. As we will show later, these files do not contain any code that is worth preserving. Second, functionality which will not work on archived pages is largely implemented by third-party scripts. Even though some of the functionality which relies on communication with origin servers (e.g., intra-site search, login) is implemented by the first-party origin, we only focus on discarding third-party files, for reasons discussed shortly.

The implication of these observations is that, to identify most of the non-functional JavaScript code in archived pages, it is unnecessary to perform any complex code analysis. Instead, it suffices to assemble and use a "filter list" which captures the features distinctive to the URLs of scripts containing non-functional code; when crawling pages, a web archive would simply have to discard (and not even fetch) any script whose URL matches the filter list.

For example, via manual analysis of the URLs of all scripts seen in $Corpus_{1M}$, we assemble a filter list comprising 45 rules. We consider those script URLs which are included on many pages. For each such popular script, we first visit the domain on which the script is hosted to understand the services offered by that domain. In cases where a domain hosts scripts of many kinds, some of which are important to retain even on archived pages, we examine the script's content to determine its utility.

Every rule in our list matches URLs at one of three granularities: 1) domain, i.e., filter any file hosted on that domain (e.g., *"zephr.com"* enables support for user subscriptions), 2) file name, i.e., filter scripts if the file name matches, regardless of the domain hosting the script (e.g., *"jquery.cookie.js"* is used for cookie management), and 3) URL token, i.e., filter scripts if a specific keyword appears anywhere in their URL (e.g., *"pagesocial-sdk"* and *"recaptcha"*).

Recall that $Corpus_{1M}$ comprises page snapshots crawled from the Internet Archive, which already discards resources that users often block on the live web, e.g., ads. In contrast, our filter list aims to prune scripts which implement functionality that is important to preserve on the live web, but will not work on archived copies. Moreover, since a few popular third-party service providers are used by the vast majority of websites [46], we find that we only need to add 6 rules to our filter list to account for pages on 300 additional sites beyond the 300 sites included in $Corpus_{1M}$.

Filtering has no impact on fidelity. Discarding a subset of the JS files on a page might, however, break the execution of code in files that are retained. Therefore, we study the impact of filtering along two dimensions: 1) visual (i.e, does the page look the same?), and 2) functional (i.e, are post-load interactions that will work on archived pages unaffected?)

We load every page in $Corpus_{3K}$ with and without filtering enabled. We take a screenshot after every page load.

```
<script src="https://js.sentry-cdn.com/7bc8b.min.js" </script>
<script>
        if (window.Sentry) {
            window.Sentry.onLoad(function() {
                window.Sentry.init({
                    maxBreadcrumbs: 30,
                environment: 'prd', });
            });
        }
</script>
```

Figure 5: **Code snippet from www.nytimes.com where the main frame first fetches a third-party JavaScript file hosted on www.js.sentry-cdn.com and then cautiously invokes a function from it inside an if condition.**

Leveraging our JavaScript instrumentation described earlier, we also 1) identify all event handlers registered during each page load, 2) trigger all event handlers after the page load completes, and 3) track all values read or written from the JavaScript heap and DOM by these handlers.

First, when we compare the screenshots for every page with and without filtering, we observe that these screenshots differ in the value of at least one pixel for 109 of the 3000 pages in $Corpus_{3K}$. Upon manual examination of these 109 pages, we find that all differences are either due to animations or because *DRP* APIs result in a different timestamp on the page. Second, for all event handlers registered by the unfiltered files, we find 35 pages on which at least one value accessed by at least one of these event handlers differed across loads with and without filtering. Again, these differences were not consequential: they were due to differences in timing information, e.g., some event handlers log the times at which their execution starts and ends.

A key reason for these positive results, which show that Jawa's filtering has no impact on the fidelity of the code retained, is our explicit choice to only consider third-party source files for filtering. On the one hand, most third party scripts are self-encapsulated, i.e., the code in these files only interacts with itself or the files it subsequently fetches. On the other hand, as shown in Figure 5, first-party scripts typically invoke third-party code cautiously, so that the former is unaffected in the off chance that the latter fails to be fetched.

Note that one cannot simply eliminate *all* third-party scripts; that would render dysfunctional many post-load interactions which do work, and are important to preserve, on archived pages. As we show later in our evaluation (§6), while discarding files which match our carefully curated filter list enables significant storage savings, doing so preserves all navigational and informational interactions.

4.3 Prune unreachable code

In the Javascript files which do not match Jawa's filter list, many lines of code will never be executed in *any* page load. This is because 1) some sources of non-determinism are absent in loads of archived pages (§3.1), and 2) Jawa elim-

inates non-determinism caused by asynchronous execution and APIs for client characteristics (§4.1). Furthermore, we found that *DRP* APIs have no impact on control flow. Yet, identifying all reachable code remains challenging: beyond the code executed while crawling the page, we also need to retain the code for users' post-load interactions.

Challenges in preserving interactions. Post-load interactions are enabled via event handlers which are registered while a page is being loaded. Every event handler is associated with a specific DOM node on the page, and is bound to a specific action that would trigger the handler, such as a click, scroll, mouse hover, etc.

The code that is exercised when an event handler on a page is invoked can vary as a function of a) the order in which the user interacts with different elements on the page, b) the inputs that the user provides to these events [30], and c) the return values of browser APIs. It is easy to see how the latter two can impact code reachability, e.g., in response to a search query, the number of search results can influence certain client type-specific UI features, such as the option of splitting the results across multiple pages. The order in which events are triggered can impact the execution of some handlers if the state read (from the DOM or JavaScript heap) by one handler could have been written to in a prior invocation of this or another handler. In particular, since we only care about identifying reachable code, only read-write dependencies which impact branch conditions are of interest.

We analyze the impact of these sources of non-determinism on the event handlers found on pages in $Corpus_{3K}$. We capture the state accessed by event handlers as described earlier in §4.2. For each event handler on a page, we check whether there is a read-write state overlap with itself or with any other event handler on the page, and if there was an overlap, whether this state is used in a branch condition. We also identify all handlers which accept user inputs; these include mouse events (e.g., click, mouseover, mouseon), keyboard inputs (e.g., keyup, keydown), and text inputs (e.g., "INPUT" or "FORM" DOM nodes). When we invoke each such handler, if a branch statement is executed, we conservatively conclude that the handler's inputs could impact the control flow of the handler.

On each page, we compare two sets of event handlers: those which work on the live version of these pages, and the subset which will work on archived copies. The former set comprises all handlers registered when we load the page without filtering. We identify the latter set of handlers by loading every page with Jawa's filtering enabled, and ignoring handlers which interact with origin servers (i.e., they are registered on either "INPUT" or "FORM" DOM nodes with a corresponding "action" attribute). On the median page, 14 event handlers work on the live page and 7 on the archived copy. At the 90th percentile, the corresponding numbers are 170 and 44.

Impact of order. On 40 of the 3000 pages in $Corpus_{3K}$, at least one pair of handlers that work on the live page had a read-write dependency which could affect the control flow of one of these handlers. In contrast, we found no such case when focusing on the handlers which work on archived pages. This stark difference is because dependencies between handlers arise on live pages predominantly due to analytics, e.g., handlers registered with certain DOM nodes update locally maintained state to track if the user interacted with those nodes; when the user navigates away from the page, another handler reads this state and sends this information to back-end servers if the user did interact with those DOM nodes.

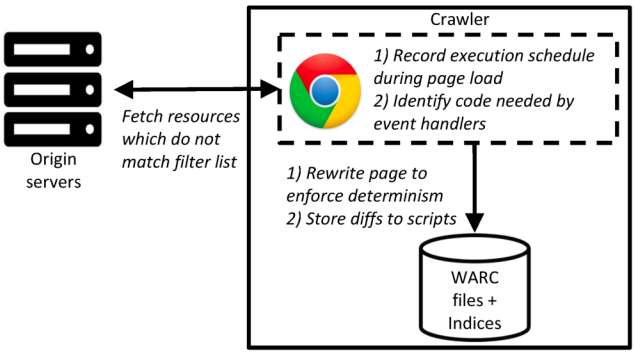


Figure 6: **High-level overview of Jawa.**

Impact of user input. Across all pages, none of the event handlers which work on archived copies read inputs which influenced branch predicates. Whereas, when we loaded all pages without filtering, 1134 of the 3000 pages had at least one handler which interacted with back-end servers. In such cases, the responses from servers could potentially impact what code gets executed on the client.

Impact of browser APIs. As discussed earlier (§4.1), Jawa eliminates non-determinism caused by APIs for client characteristics. That leaves *DRP* APIs. Among the handlers that work in an archived context, 449 of the 3000 pages had at least one handler which invoked an API from either "Date" or "Math.random". All the invocations of "Math.random" APIs were due to the jQuery library assigning unique identifiers to elements inside its `ElementSelector` function [15]. Whereas, the "Date" API was used only for logging the start and end time of handler executions. Thus, in all of these cases, *DRP* APIs did not impact the reachable code for any event handler.

Takeaways. These results demonstrate why prior work which aims to identify code reachable by event handlers performs complex JavaScript program analysis [30, 59]. In contrast, we find that no source of non-determinism impacts the code executed by handlers which work on archived pages. Therefore, on any page, to identify the code necessary to retain for post-load interactions to work, it suffices for Jawa to invoke every handler once and save the code that is executed.

4.4 Summary

Put together, our observations on the differences between loads of archived and live pages enable Jawa to use a fairly simple methodology to crawl and save pages, as shown in Figure 6. For every page that it crawls, Jawa fetches all those resources which do not match its filter list. For the remaining files, it ① injects code to identify what code was executed during the page load and in what order, and ② triggers every registered event handler using default input values (e.g., the default x and y coordinates for a mouse click event is 0,0) and identifies the code executed. Finally, it stores those portions of the page that are exercised in either step above. It instruments the retained code so that, when users load the page, their browser follows the same execution schedule and uses the same client characteristics.

5 IMPLEMENTATION

Implementing a web archive involves several considerations which are outside the scope of this paper, e.g., distributing data across servers, detecting and coping with hardware failures, etc. Our implementation focuses on the aspects of a web archive addressed by Jawa (Figure 6), namely crawling and storing page snapshots. We also describe the impact of Jawa's design on serving page snapshots to users.

5.1 Crawling pages

When crawling a page, Jawa's crawler (1.2K LOC) uses a Node.js based man-in-the-middle proxy to interpose on all requests/responses. The proxy uses the Esprima [9] and BeautifulSoup [4] libraries to instrument JavaScript and HTML files as they are fetched. Jawa references the filter list for every outgoing request and, using regular expression matching, blocks the request for any resource whose URL matches any of the rules in the filter list. For all the remaining resources fetched, Jawa selectively instruments JS files prior to their execution. This instrumented code, upon execution, enables Jawa to 1) interpose on all browser APIs, 2) track the subset of JS code executed (in terms of JS functions), and 3) helps enumerate all event handlers registered on the page. The instrumentation overhead incurred by the crawler is significantly lower compared to when tracking all state accesses (§4).

5.2 Storing page snapshots

For every page that it crawls, Jawa saves only a subset of the JavaScript code on that page. Consequently, when the same JavaScript file (e.g., a library) is included on many pages, it is often the case that different subsets of this file need to be stored as part of different page snapshots, thereby preempting simple file-level deduplication, as used by the Internet Archive today [23].

Our solution is to store every unique file as a set of partitions; each partition represents a different disjoint subset of the file: from a specific start byte offset to an end byte offset. When Jawa crawls a new page snapshot, for every JavaScript file crawled that is not filtered, it identifies the subset of code in this file relevant for this snapshot. It then looks up the crawl index (Table 2) to determine if this subset is already covered by the byte ranges in this file that have previously been stored. The crawler creates new WARC records for portions of the file that have not been previously stored and appends new entries to the crawl index. The crawl index is processed asynchronously to produce the serving index (like is the case today with Internet Archive).

Crawl index		
	Key	*Value*
IA	URL	List of (content hash, WARC file ID) tuples
Jawa	(URL, content hash)	List of (start byte offset, end byte offset, WARC file ID) tuples

Serving index		
	Key	*Value*
IA	(URL, timestamp)	(WARC file ID, byte offset)
Jawa	(URL, timestamp)	List of (WARC file ID, byte offset) tuples

Table 2: **Comparison of indices maintained by IA and Jawa.**

5.3 Serving page snapshots

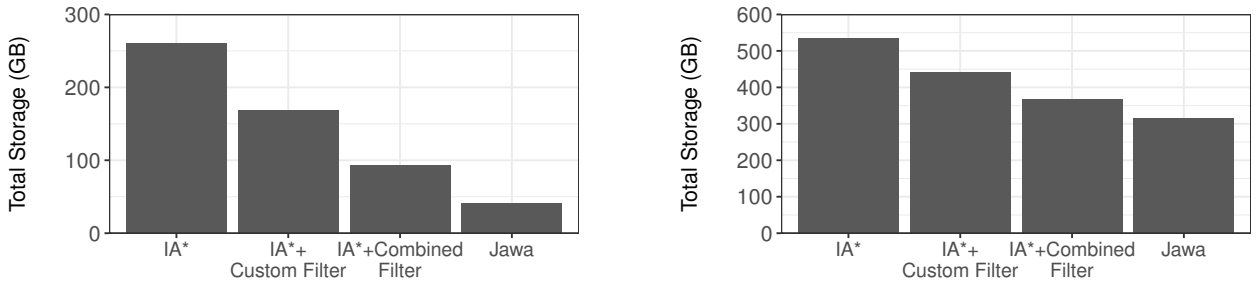
The implication of storing any JavaScript file's contents as above is that, when a client requests for a file while loading a page snapshot, one does not know which of the partitions stored for this file are relevant for this particular snapshot. Instead, a web archive which uses Jawa can return the union of all stored partitions for the requested JavaScript file; after all, the portion of the file needed for any snapshot is a subset of the stored partitions. Since the size of this union is at most equal to the size of the original file, clients will have to fetch no more bytes than they do today.

6 EVALUATION

We evaluate Jawa with three metrics: storage (to store crawled resources and to store indices), fidelity (similarity of archived page snapshots to the corresponding original pages) and performance (both for crawling and serving). In all cases, we compare against the corresponding techniques currently in use by the Internet Archive (§2), which we refer to as IA*.[3] In some cases, we also break down the utility/overhead of each of Jawa's components. The key findings from our evaluation are as follows:

- Jawa reduces the storage needed for our corpus of 1 million page snapshots by 41%. This reduction stems from Jawa discarding 84% of JavaScript bytes.
- Despite this significant reduction in storage, on a random sample of pages, all event handlers that one would expect to function on archived pages continue to work.
- When we mimic loads of archived pages from IA, at least a quarter of resource fetches fail on more than 10% of pages.

[3] IA* refers to us mimicking the techniques used by IA.



(a) JavaScript resources

(b) All resources

Figure 7: **Total storage necessary to store corpus of 1 million page snapshots.**

Whereas, on over 99% of pages, Jawa eliminates all failed network fetches and ensures that the set of resources requested from the archive match those crawled.

- Crawling throughput with Jawa improves by 39%, thanks to our use of lightweight techniques for code analysis and filtering of JavaScript files.

6.1 Storage

6.1.1 Storage for resources.

To begin, we consider the total amount of storage needed to store the resources in our $Corpus_{1m}$ corpus. We crawl all of these page snapshots from IA using our crawler (§5). On each page, Jawa's crawler only fetches third-party JavaScripts which do not match its filter list. Apart from our manually curated filter list for pruning code which will not function on archived pages, we also leverage the open-source filter list from EasyList [8], which is widely used by many browser extensions to identify ads and analytics. In every script that it does fetch when crawling a page snapshot, Jawa's crawler identifies the subset of code necessary for this snapshot and stores the portion of this subset that is not covered by the subsets of this file previously stored.

Figure 7(a) shows that Jawa stores 40 GB of JavaScript across the 1 million pages, a reduction of 84% compared to IA*. Of course, to store the entire corpus, all resources on every page snapshot need to be saved, not only JavaScripts. For resources other than scripts (images, CSS, HTML, fonts), Jawa offers no storage benefits; it stores them exactly as IA*. Yet, we see a 41% reduction in total storage: 535GB with IA* to 314GB with Jawa (Figure 7(b)). This is because, as seen earlier in §2.2, JavaScript files account for 49% of all the bytes across all pages, even after file-level deduplication. Since 63% of the more than 140 PB of data stored by IA is devoted to web page snapshots [12, 13], we estimate that Jawa can reduce IA's storage needs by 35 PB.

Sources of storage benefits. Storage savings enabled by Jawa stem from a combination of not storing filtered files and pruning unreachable code. When we break down the impact of the filter lists we use, Figure 7(a) shows that our custom filter list alone reduces the total amount of JavaScript saved by 36%, and EasyList's rules result in a further reduction of 28%. Jawa also significantly reduces storage needs by eliminating unused code: the difference between the two right most bars in Figure 7.

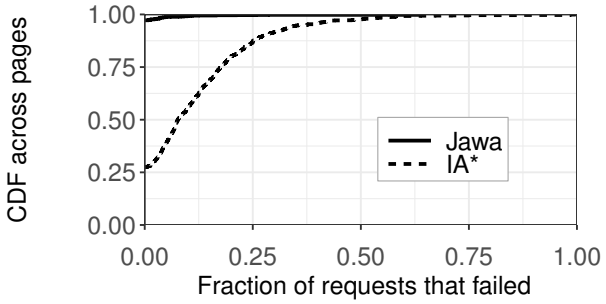
6.1.2 Storage for indices

In addition to storing crawled resources, both IA* and Jawa also need to store the crawling and serving indices (Table 2). The former enables the crawler to not store duplicate content, whereas the latter enables lookups of requested resources when serving page snapshots. For our corpus of 1 million page snapshots, we find that size of both indices is marginally smaller (15%) with Jawa than with IA*. First, for most script files, Jawa ends up having to store a single WARC record; for such files, after the first time a subset of the file's code is stored, all subsequent page snapshots which include the same file end up needing the same subset. Second, the increase in index entries for other files (for which multiple subsets end up being stored) is offset by the elimination from the index of filtered files.

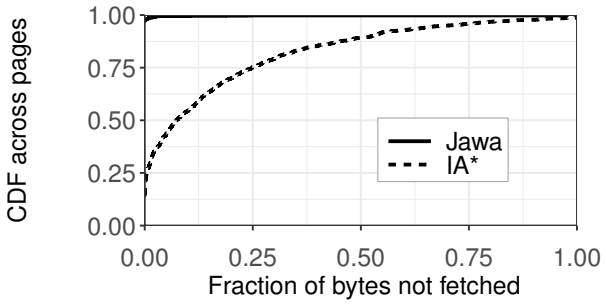
6.2 Fidelity

To evaluate Jawa's preservation of page fidelity, we crawl all 3000 pages in $Corpus_{3K}$ from the live web. We perform these crawls on a desktop, once with Jawa's crawler, and once without using any of its methods. We then load these pages from the two local copies, mimicking a different client ("iPhone 6"). When using page snapshots saved by Jawa, we match requested URLs to crawled URLs after stripping query strings.

Resource fetches. We first evaluate Jawa's impact on fidelity by examining the discrepancy between the set of resources stored for any snapshot and the set of resources fetched by a client when it loads that snapshot. Figure 8(a) shows that, while 7% of network requests return a 404 on the median page in loads of IA*, this fraction drops to 0% with Jawa. On the 95th percentile page, the corresponding fractions are 36% with IA* and 0% with Jawa. Consequently, Figure 8(b) shows that, while 10% of stored resources are not fetched on the median page when mimicking loads from IA, this fraction drops to 0% with Jawa. On the 95th percentile page, the corresponding fractions are 75% with IA* and 0% with Jawa.



(a)



(b)

Figure 8: **When snapshots of 3K pages are served, (a) number of resources requested by client which are not stored, and (b) fraction of resources stored for a snapshot which are not fetched by the client.**

Visual analysis. To check if the pages served by Jawa are identical to the ones it crawled, we take a screenshot of every page both when crawling it and when we reload it from our local copy. We then compare every pair of screenshots to check if the value of every pixel matches. Apart from the visual differences accounted for by animations and non-determinism in 54 pages, both screenshots matched exactly for every other page when using Jawa. Since loads of IA* do not patch APIs for client characteristics, differences in screen dimensions between clients make it moot to compare screenshots.

Interactions. Finally, to evaluate Jawa's impact on post-load interactions, we randomly sample 150 pages. For each page, we load the versions that would be served by IA* and by Jawa. To isolate the impact of Jawa's techniques, we also consider an intermediate design point (Only filter) where we only use Jawa's filtering but do not prune unreachable code.

We categorize all event handlers on every page into three types: 1) navigational, i.e., they help in navigating either to a different page (e.g., a navigational bar) or within the page (e.g., a scroll-to-bottom button), 2) informational, i.e., they help make more information available (e.g., carousels or tabs), and 3) transactional (e.g., login or post buttons). On archived pages, transactional event handlers will not function. So, on each of the 150 sampled pages, we manually trigger all event handlers that belong to the first two categories. All 124 navigational interactions and 100 informational interactions worked as expected in all three loads: IA*, Only filter, and Jawa. Key to preserving these post-load interactions are Jawa's carefully curated filter list for discarding non-functional code, and its methods for identifying and retaining all reachable code. In contrast, if we discard all third-party files or if we use Jawa's filter list but save only the functions registered as handlers, then only 42% of these interactions work in the former case and 10% in the latter.

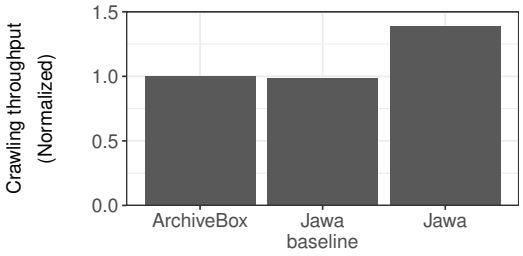


Figure 9: **Comparison of crawling throughput, normalized to that offered by ArchiveBox.**

6.3 Performance

Crawling throughput. IA's production crawler is not public to the best of our knowledge. Therefore, we turn to two open-source crawlers: Brozzler [5] and ArchiveBox [3]. Brozzler is operated by IA, and used alongside their production crawler. Whereas, ArchiveBox is a very active and commonly used crawler by individual archivists (over 12K stars on GitHub). We find that Brozzler is 20% slower than ArchiveBox because of the latter's more efficient implementation of their headless Chrome interface. We also note, that on a server with 32 cores and 128 GB RAM, we were able to crawl 5000 URLs in 15 minutes with ArchiveBox. With this crawling throughput, IA would need to dedicate 900 such servers for crawling pages, which is comparable to the number of servers they currently claim to use [11]. Therefore, we evaluate Jawa against ArchiveBox.

Figure 9 shows that Jawa's crawler offers throughput comparable to Archivebox when all of Jawa's techniques are disabled (Jawa baseline). Enabling all the methods in Jawa's design increases our crawler's throughput by 39%.

To breakdown the overheads, we measure the latency of each of the techniques used by Jawa's crawler in isolation, namely 1) filter: filtering JavaScript files, 2) code injection (CI): instrumenting the code in fetched scripts, 3) dynamic tracking (DT): dynamically tracking code execution and event handler registration, and finally 4) event triggering (ET): invoking event handlers and capturing the code executed. Figure 10 shows that not having to fetch filtered scripts completely offsets the overheads of all other techniques. Not only does Jawa's crawler not fetch any scripts which match its filter list, but all the resources that would have been fetched by the filtered files also go unfetched; this latter set of files often do not match the filter list.

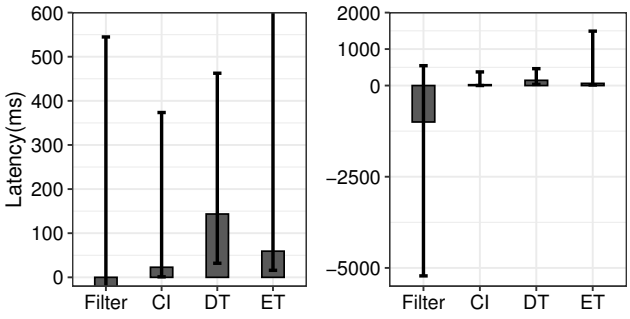


Figure 10: **Benchmarking the overhead of techniques used in Jawa's crawler. Bars plot median across pages, and whiskers plot 10^{th} and 90^{th} percentiles. Graph on the left zooms in on the yrange 0 to 500ms in the graph on the right.**

Index	I/Os per page with IA*		Reduction in I/Os per page with Jawa	
	50^{th} %ile	90^{th} %ile	50^{th} %ile	90^{th} %ile
Crawling	3	15	1	5
Serving	41	107	1	3

Table 3: **Writes on crawling index and reads on serving index; values shown for 50th and 90th percentile page on median site.**

Jawa also impacts crawling throughput by requiring more writes to the crawling index because, unlike IA*, it spreads the code in some script files across multiple WARC records. We cannot quantify the performance impact of doing so since our setup does not match a production archive like IA. However, we can quantify the number of additional writes that Jawa performs to the crawl index, compared to IA*. Table 3 shows that the number of writes to the crawl index *decrease* with Jawa; due to filtering, fewer files are crawled.

Serving performance. When serving page snapshots, Jawa's only overhead is in needing to potentially lookup multiple WARC records in order to respond to a request for a JavaScript file. We find that page load times on IA's Wayback Machine are proportional to the number of resources on the requested page snapshot, or equivalently, the number of WARC records that IA needs to lookup to serve the snapshot. Therefore, as a proxy for estimating Jawa's impact on user-perceived performance, we examine the increase due to Jawa in the number of WARC records read when serving page snapshots. Table 3 shows that the number of index lookups decrease with Jawa; again, thanks to filtering, a client has to fetch fewer files per snapshot.

7 Verifying Page Properties

Jawa's methods for pruning non-functional and unreachable code are based on three properties that we found to be true on archived web pages:

- *DRP* APIs have no impact on control flow
- Discarding third-party JavaScript files which match a manually curated filter list has no impact on fidelity
- For post-load interactions which work on archived pages, the event handlers which power them do not have read-write dependencies that influence branch conditions

All of these observations are rooted in our empirical analysis of a variety of web pages in *Corpus$_{3K}$*: 9 internal pages and 1 landing page in each of 300 sites, which span a wide range of rankings among Alexa's top million sites. However, we recognize that not all pages may abide by these properties. For example, consider a page which shows the time until a deadline and switches the font color when the time remaining is below a threshold; such a page would violate the first property listed above.

To handle such cases, we observe that web archives do not crawl every page just once; they repeatedly recrawl pages over time in order to capture changes to every page's content. For any given page, in some crawls of the page, a web archive can disable all of Jawa's methods and check if the properties expected to be true indeed hold on this page. For example, like the analysis we performed (§4), the web archive can instrument scripts to track state accesses, and then examine dependencies between event handlers and between files which do or do not match the filter list. It can also perform concolic execution to verify that *DRP* APIs have no impact on control flow.

The key to restricting the compute overheads of these heavyweight analyses is to run them on a sample of snapshots. To determine the sampling rate, a web archive can leverage properties that are stable across a page's snapshots. For example, upon analyzing all of IA's snapshots for 300 randomly chosen pages, we observe that the median page has the same number of runtime errors for an average of 53 snapshots. Therefore, once in every 53 crawls of any of these pages, a web archive can disable filtering and check if the number of runtime errors matches prior crawls where filtering had been used. If there is a mismatch, the web archive can disable the use of filtering for this page going forward. Since Jawa serves any JavaScript file to users as the union of all partitions of this file stored across crawls (§5), disabling filtering in one crawl of the page will also benefit all prior crawls of that page.

8 Discussion

How future proof is Jawa? In the immediate future, recent trends [18] indicate that the amount of JS on pages will continue to increase, making it important for web archives to adopt Jawa's techniques for pruning JS and for eliminating fidelity issues due to the non-determinism introduced by JS. In the long term, we expect that the principles that dictate Jawa's design will continue to hold: to serve pages with high fidelity, 1) archives must account for non-determinism, and 2) a large fraction of JS can be discarded with no risk.

Optimize already archived pages. Jawa's simple techniques make it highly amenable to be used with pages that have already been archived. First, a web archive can sig-

nificantly reduce its storage needs by discarding all JS files that match Jawa's filter list. Second, the web archive can rewrite the HTML of every archived page to include a custom script which will enforce the same client characteristics as the crawler when users load the page. The only aspect of Jawa that would be hard to use on already archived pages is the elimination of unreachable code, as that requires invoking all event handlers on every page.

9 Related work

Impact of JavaScript on web crawlers. Prior work has shown that it is important for web crawlers to execute JavaScript when crawling pages, both in the context of web archives [31, 32, 33] and web search engines [1], else many important resources on a page will often go uncrawled. Our work highlights that, due to the non-deterministic execution of JavaScripts, archived pages often have poor fidelity even when pages are crawled using a browser which executes all scripts on every page.

Beyond executing JavaScripts while crawling a page, systems like Conifer [6] also save all resources on the page that are fetched while the user is interacting with the page. However, such systems are designed for private web archival, i.e., a user saves a page and its constituent resources for the user's own personal use later. If users load a page archived by a different user using a different device/browser, they will face the same fidelity issues seen on the Internet Archive.

Coverage of web archives. Many measurement studies [27, 28] have demonstrated that web archives are far from comprehensive in archiving all pages on the web. Prior work [50, 41] has attempted to address the incompleteness caused due to large portions of the web not being openly available (e.g., behind paywalls) and requiring user logins (e.g., social media). In contrast, we seek to enable web archives to improve their coverage by reducing the costs associated with archiving any corpus of pages; thereby, for the same budget, a web archive can crawl and save more pages.

Supporting bulk processing of archives. Jawa focuses on enabling web archives to support the use case where users load individual page snapshots and interact with them. Alternatively, web archives are used by researchers to perform large scale analyses of historical information. Xinyue et al. [64] demonstrate the performance penalties of the WARC format for such batch processing workloads, and many systems [47, 2, 39] have been developed to enable programmatic analysis of large corpuses without needing to access each individual resource on every page.

JavaScript record and replay systems. A number of prior systems [29, 52, 60] enable users to record and replay JavaScript execution, both in the context of browsers [29] and independent JavaScript programs [60]. These record and replay tools are critical for debugging JavaScript based errors. Therefore, to ensure high fidelity replay, all of these systems identify and patch all sources of non-determinism to match the recorded version. In contrast, we analyze the individual impact of each source of non-determinism on the URLs fetched and patch them accordingly.

Code reachable through event handlers. JavaScript testing tools automate the process of testing by dynamically constructing test cases to achieve maximum code coverage. A key part of this process is identifying all code that can be potentially executed by event handlers. Doing so requires heavyweight symbolic execution analysis [42], or exhaustively going through all possible orders and inputs [30]. Jawa leverages the differences between archived and live web pages to simplify this analysis by only needing to use the trace from a single execution.

Program analysis on the web. JavaScript on the web has been notorious for various kinds of security, privacy and performance issues. A large body of prior work focuses on addressing such issues by relying on sophisticated program analysis techniques [63, 66]. Such techniques, however, incur a high computation cost. This is why, in solutions for optimizing web performance [42, 54, 49] which use computationally expensive JavaScript analysis techniques, web servers perform such analysis in the background to mitigate the impact of their overheads. For archival systems, even if crawled JavaScript resources are processed offline, the cost for computationally heavyweight processing is not sustainable. Hence, Jawa employs lightweight approaches, rooted in properties of JavaScript on the web.

Dead code elimination on the web. One way to optimize web performance is to eliminate dead code (i.e., code that is never reachable) from resources such as JavaScript and CSS. Tools [17, 25] which do so using static analysis are widely used. We observe that, in archived pages, a significantly greater fraction of code is potentially unreachable, since many sources of non-determinism (e.g., variation in client state and server responses) are absent. Jawa exploits this property to provide significant storage savings.

10 Conclusion

Since when the Internet Archive began operating in the late 1990s, a marked change on the web has been the increased use of JavaScript. In this paper, we shined light on two significant problems caused by this change: broken rendering of archived pages, and petabytes of storage wasted on JavaScript which will either be non-functional or never be used. Our design of Jawa addresses these problems while emphasizing low overhead on both crawling and serving pages. As a result of our work, web archives will be able to archive many more pages than they can today for the same cost and ensure that archived pages more closely approximate their original versions.

Acknowledgements: We thank the anonymous reviewers and our shepherd, Philip Levis, for their valuable feedback.

REFERENCES

[1] https://developers.google.com/search/docs/advanced/javascript/javascript-seo-basics.

[2] Archive unleashed. https://github.com/archivesunleashed/aut.

[3] Archivebox. https://github.com/ArchiveBox/ArchiveBox.

[4] BeautifulSoup. https://pypi.org/project/beautifulsoup4/.

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

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

[7] Donate to the Internet Archive! https://archive.org/donate/.

[8] EasyList. https://easylist.to/.

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

[10] HTTP Archive: State of the web. https://httparchive.org/reports/state-of-the-web#bytesTotal.

[11] IA infrastructure. https://archive.org/details/jonah-edwards-presentation.

[12] Inside wayback machine. https://thehustle.co/inside-wayback-machine-internet-archive/.

[13] Internet archive. https://www.archive.org/about/.

[14] Internet Archive tax return. https://projects.propublica.org/nonprofits/organizations/943242767.

[15] jQuery element selector. https://api.jquery.com/element-selector/.

[16] Page-vault. https://www.page-vault.com/solutions/.

[17] Prepack. https:/www.prepack.io.

[18] State of JavaScript. https://httparchive.org/reports/state-of-javascript.

[19] Stillio. https://www.stillio.com/.

[20] The Boston Globe: Internet archive's copy from August 2, 2020. https://web.archive.org/web/20200802084355/https://www.bostonglobe.com/.

[21] The Daily Caller: Internet archive's copy from September 5, 2020. https://web.archive.org/web/20200905133311/https://dailycaller.com/.

[22] The WARC format 1.0. https://iipc.github.io/warc-specifications/specifications/warc-format/warc-1.0/.

[23] WARC revisit tag. https://iipc.github.io/warc-specifications/specifications/warc-format/warc-1.0/#revisit.

[24] Wayback machine. https://www.archive.org/web.

[25] Webpack. https://webpack.js.org/guides/tree-shaking/.

[26] Webrecorder. https://webrecorder.net/.

[27] S. G. Ainsworth, A. Alsum, H. SalahEldeen, M. C. Weigle, and M. L. Nelson. How much of the web is archived? In *Joint Conference on Digital Libraries*, 2011.

[28] A. AlSum, M. C. Weigle, M. L. Nelson, and H. Van de Sompel. Profiling web archive coverage for top-level domain and content language. *International Journal on Digital Libraries*, 2014.

[29] S. Andrica and G. Candea. WaRR: A tool for high-fidelity web application record and replay. In *DSN*, 2011.

[30] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of javascript web applications. In *ICSE*, 2011.

[31] J. F. Brunelle, M. Kelly, H. SalahEldeen, M. C. Weigle, and M. L. Nelson. Not all mementos are created equal: Measuring the impact of missing resources. *International Journal on Digital Libraries*, 2015.

[32] J. F. Brunelle, M. Kelly, M. C. Weigle, and M. L. Nelson. The impact of javascript on archivability. *International Journal on Digital Libraries*, 2016.

[33] J. F. Brunelle, M. C. Weigle, and M. L. Nelson. Archival crawlers and javascript: discover more stuff but crawl more slowly. In *Joint Conference on Digital Libraries*. IEEE, 2017.

[34] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *NSDI*, 2015.

[35] Z. T. Fernando, I. Marenzi, and W. Nejdl. ArchiveWeb: Collaboratively extending and exploring web archive collections—how would you like to work with your collections? *International Journal on Digital Libraries*, 2018.

[36] D. Fetterly, M. Manasse, M. Najork, and J. L. Wiener. A large-scale study of the evolution of web pages. *Software: Practice and Experience*, 2004.

[37] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.

[38] A. Goel, V. Ruamviboonsuk, R. Netravali, and H. V. Madhyastha. Rethinking client-side caching for the mobile web. In *HotMobile*, 2021.

[39] H. Holzmann, V. Goel, and A. Anand. Archivespark: Efficient web archive access, extraction and derivation. In *Joint Conference on Digital Libraries*, 2016.

[40] International Internet Preservation Consortium. Access Working Group. Use cases for access to internet archives. *IIPC Report*, 2006.

[41] M. Kelly, M. L. Nelson, and M. C. Weigle. A framework for aggregating private and public web archives. In *Joint Conference on Digital Libraries*, 2018.

[42] R. Ko, J. Mickens, B. Loring, and R. Netravali. Oblique: Accelerating page loads using symbolic execution. In *NSDI*, 2021.

[43] J.-w. Kwon and S.-M. Moon. Web application migration with closure reconstruction. In *WWW*, 2017.

[44] S. Lawrence, F. Coetzee, E. Glover, G. Flake, D. Pennock, B. Krovetz, F. Nielsen, A. Kruger, and L. Giles. Persistence of information on the web: Analyzing cita-

tions contained in research articles. In *CIKM*, 2000.
[45] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 1966.
[46] T. Libert. Exposing the hidden web: An analysis of third-party HTTP requests on 1 million websites. *International Journal of Communication*, 2015.
[47] J. Lin, M. Gholami, and J. Rao. Infrastructure for supporting exploration and discovery in web archives. In *WWW*, 2014.
[48] B. Loring, D. Mitchell, and J. Kinder. ExpoSE: Practical symbolic execution of standalone JavaScript. In *SPIN Symposium on Model Checking of Software*, 2017.
[49] S. Mardani, A. Goel, R. Ko, H. V. Madhyastha, and R. Netravali. Horcrux: Automatic javascript parallelism for resource-efficient web computation. In *OSDI*, 2021.
[50] C. C. Marshall and F. M. Shipman. On the institutional archiving of social media. In *Joint Conference on Digital Libraries*, 2012.
[51] J. Mickens. Rivet: Browser-agnostic remote debugging for web applications. In *USENIX ATC*, 2012.
[52] J. W. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic capture and replay for javascript applications. In *NSDI*, 2010.
[53] J. Nejati, M. Luo, N. Nikiforakis, and A. Balasubramanian. Need for mobile speed: A historical analysis of mobile web performance. In *TMA*, 2020.
[54] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *NSDI*, 2016.
[55] R. Netravali and J. Mickens. Prophecy: Accelerating mobile page loads using final-state write logs. In *NSDI*, 2018.
[56] R. Netravali, V. Nathan, J. Mickens, and H. Balakrishnan. Vesper: Measuring time-to-interactivity for web pages. In *NSDI*, 2018.
[57] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *USENIX ATC*, 2015.
[58] A. Ntoulas, J. Cho, and C. Olston. What's new on the web? the evolution of the web from a search engine perspective. In *WWW*, 2004.
[59] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *IEEE Symposium on Security and Privacy*, 2010.
[60] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *FSE*, 2013.
[61] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE*, 2005.
[62] D. Spinellis. The decay and failures of web references. *Communications of the ACM*, 46(1):71–77, 2003.
[63] O. Tripp and O. Weisman. Hybrid analysis for javascript security assessment. In *ESEC/FSE*, 2011.
[64] X. Wang and Z. Xie. The case for alternative web archival formats to expedite the data-to-insight cycle. In *Joint Conference on Digital Libraries*, 2020.
[65] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with WProf. In *NSDI*, 2013.
[66] S. Wei and B. G. Ryder. Practical blended taint analysis for javascript. In *International Symposium on Software Testing and Analysis*, 2013.

A Artifact Appendix

A.1 Abstract

Our open-source artifact contains the scripts and the data necessary to produce the key results from this paper. It also contains the code for the analysis framework which informed Jawa's design.

A.2 Scope

The artifact can be used to confirm the three main benefits of Jawa: a) reduced storage overhead, b) improved fidelity by eliminating almost all failed network requests, and c) improved crawling throughput.

A.3 Contents

The artifact contains all the code required to generate the key results with respect to three metrics: storage, fidelity and throughput. This includes a) Jawa's filter list and a NodeJS based crawler that leverages this filter list while loading web pages; b) a NodeJS based analyzer that injests JS files and instruments them to track all the JS functions executed at runtime, the set of event handlers registered, and the return values of browser APIs; and c) a set of scripts to automatically run the above code on a given corpus of pages. These scripts will produce the following results:

- **E1**: Reduced storage overhead using Jawa's two techniques: eliminating non-functional code using the filter list, and eliminating unused code by tracking the set of functions executed during the page load plus those required for enabling user interactions. This result will mimic the trend shown in Figure 7.

- **E2**: Improved page fidelity by eliminating almost all failed network requests. This result will reproduce the number of failed requests and the corresponding number of bytes not fetched, as shown in Figure 8.

- **E2**: Improved crawling throughput by reducing the number of IOs on the crawling index. This result will mimic the trend shown in the *"Crawling"* column of Table 3.

Apart from the scripts, the artifact contains a corpus of 3000 pages which is pre-recorded using the Mahimahi [57] tool. All scripts are run on this corpus of pages. Finally, the artifact also contains the JS analysis framework which was used to inform Jawa's design choices (§3).

A.4 Hosting

The source code of the artifact is hosted on https://github.com/goelayu/Jawa with the corresponding commit ID: *"07e358eeed7cc054747271b19070b5563f3ff189"*. The corpus of pages is hosted on Google Drive.

A.5 Requirements

Software dependencies

The artifact has been tested on Ubuntu 16.04.7 LTS. It requires installing the following dependencies, in addition to the NodeJS dependencies included in the github repo (§A.6):

```
$ sudo apt-get install mahimahi google-chrome-
    stable parallel r-base r-base-core
$ sudo sysctl -w net.ipv4.ip_forward=1
```

A.6 Installations

Setting up the artifact involves three steps: a) downloading the source code and installing the NodeJS dependencies, b) patching the NodeJS dependencies to use the modified versions included in the github repo, and c) fetching and extracting the corpus of pages to run the analysis on.

Install the code

```
$ git clone https://github.com/goelayu/Jawa
$ cd Jawa
$ npm install
$ export NODE_PATH=${PWD}
```

Patch the dependencies

```
$ vim node_modules/puppeteer-extra-plugin-
    adblocker/dist/index.cjs.js
# add to line 73:
  return adblockerPuppeteer.PuppeteerBlocker.parse
    (fs.readFileSync('../filter-lists/combined-
    alexa-3k.txt', 'utf-8'));
```

Fetch the data

```
$ cd data
# download tarball from https://drive.google.com/
    file/d/17j6AYgaaXMhmV0VKWUmU_kMcHibMryVV/view?
    usp=sharing
$ tar -xf corpus.tar
```

A.7 Experiments workflow

As listed in §A.3, the artifact scripts will produce results corresponding to three metrics: storage, fidelity and crawling throughput.

A.7.1 Fidelity

We provide scripts and data to exactly reproduce Figure 8 (both a and b). The corpus used for this experiment contained 3000 pages. On a single core machine, it takes roughly 20–30 seconds for each page to load and, therefore, takes about 20 hours to load all 3000 pages once. We recommend to either run this experiment on a smaller corpus of pages (more details below) or to use a multi-core (16–32 cores) machine to speed up the overall execution time.

```
$ cd ../ae
# Usage: ./fidelity.sh <corpus_size> <num of
    parallel processes>
$ ./fidelity.sh 3000 1 # depending on the number
    of available cores on your machine, provide
    the 2nd argument
```

The output graphs will be generated in the same directory: *"count_fidelity.pdf"* and *"size_fidelity.pdf"*, corresponding to Figures 8(a) and 8(b), respectively.

A.7.2 Storage

Reproducing Figure 7 requires processing 1 million pages, which would take around a week (even with 128 CPU cores). We instead provide scripts to process 3000 pages, and demonstrate storage savings derived from both of Jawa's techniques. We provide preprocessed web pages, i.e., injected with instrumentation code to detect which functions are executed at runtime, and code to track event handlers. You can fetch the the instrumented pages as follows:

```
$ cd ../data
# download tarball from https://drive.google.com/
    file/d/16Pt4a2l1CNxC8UBwjalgEki-UlGAnFUm/view?
    usp=sharing
$ tar -xf processed.tar
```

You can now run the end-to-end storage analysis script:

```
$ cd ../ae
# Usage: ./storage.sh <corpus_size> <num of
    parallel processes>
$ ./storage.sh 3000 1 # depending on the number of
     available cores on your machine, provide the
    2nd argument
```

The above script will print three storage numbers (in bytes) to the console. a) Total JS storage after deduplication (as incurred by Internet Archive); this mimics the *"IA*"* bar in Figure 7(a). b) Total JS storage after applying Jawa's filter; this mimics the *"IA*+Combined Filter"* bar in Figure 7(a). c) Total JS storage after removing unused JS functions; this mimics the *"Jawa"* bar in Figure 7(a).

A.7.3 Crawling throughput

We reproduce the throughput results from Table 3's *"Crawling"* column. The storage script above outputs the crawling index IOs as well. It prints the following two numbers: a) reductions in crawling IOs for the 50th percentile page, and b) reductions in crawling IOs for the 95th percentile page.