Breaking Bad: Quantifying the Addiction of Web Elements to JavaScript

Romain Fouquet, Pierre Laperdrix, Romain Rouvoy

Breaking Bad: Quantifying the Addiction of Web Elements to JavaScript

ROMAIN FOUQUET, Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, France
PIERRE LAPERDRIX, CNRS, Univ Lille, Inria Lille, France
ROMAIN ROUVOY, Univ. Lille / Inria / IUF, France

While JavaScript established itself as a cornerstone of the modern web, it also constitutes a major tracking and security vector, thus raising critical privacy and security concerns. In this context, some browser extensions propose to systematically block scripts reported by crowdsourced trackers lists. However, this solution heavily depends on the quality of these built-in lists, which may be deprecated or incomplete, thus exposing the visitor to unknown trackers. In this paper, we explore a different strategy, by investigating the benefits of disabling JavaScript in the browser. More specifically, by adopting such a strict policy, we aim to quantify the JavaScript addiction of web elements composing a web page, through the observation of web breakages. As there is no standard mechanism for detecting such breakages, we introduce a framework to inspect several page features when blocking JavaScript, that we deploy to analyze 6,384 pages, including landing and internal web pages. We discover that 43 % of web pages are not strictly dependent on JavaScript and that more than 67 % of pages are likely to be usable as long as the visitor only requires the content from the main section of the page, for which the user most likely reached the page, while reducing the number of tracking requests by 85 % on average. Finally, we discuss the viability of currently browsing the web without JavaScript and detail multiple incentives for websites to be kept usable without JavaScript.

CCS Concepts: • **Information systems** → **World Wide Web**; • **Security and privacy** → *Browser security*.

Additional Key Words and Phrases: web privacy, javascript, page breakage

ACM Reference Format:
Romain Fouquet, Pierre Laperdrix, and Romain Rouvoy. 2023. Breaking Bad: Quantifying the Addiction of Web Elements to JavaScript. *ACM Trans. Internet Technol.* 1, 1, Article 1 (January 2023), 29 pages. https://doi.org/10.1145/3579846

1 INTRODUCTION

JavaScript has contributed to the development of web tracking and advertisement technologies for more than two decades. Tracking techniques have become more and more complex: simple, cookie-based tracking continues to exist, but new tracking strategies have emerged, leveraging the always-increasing complexity of the web, from browsers to server-side infrastructure. In response, many countermeasures have been investigated and deployed, embedded into browsers or delivered as extensions. They usually either try to prevent the browser feature from being used for tracking, for instance using site isolation [28, 31, 33], user permission requests [6] or data randomization [51, 52], or try to selectively block resources implementing user tracking features, while keeping the site functionality [7].

Blocking such tracking resources—i.e., scripts in most cases—usually relies on blocklists, which consist of rules matching requests to block. Blocklists are a simple and efficient privacy mechanism, but are well known for suffering from several limitations. Being maintained by humans, they tend

Authors' addresses: Romain Fouquet, Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000, Lille, France, romain.fouquet@inria.fr; Pierre Laperdrix, CNRS, Univ Lille, Inria Lille, Lille, France, pierre.laperdrix@inria.fr; Romain Rouvoy, Univ. Lille / Inria / IUF, Lille, France, romain.rouvoy@univ-lille.fr.

to be limited to geographical regions and languages with enough list maintainers [54]. They are also likely to be lagging behind, as trackers try to escape the blocklists [55]. There have been efforts to automate blocklist generation [23], but blocklists are also often technically limited by the type of resources they can block: aiming to selectively block the requests, they can only rely on the URLs pointing to these resources, which can be circumvented by inlining the scripts directly in the HTML document or by using an always-changing, hard-to-match URL [16].

JavaScript being the most powerful tracking vector, it would make it much more difficult for trackers to operate if JavaScript was simply completely blocked in the browser. However, JavaScript has also many legitimate uses, and blocking it is bound to break at least some features of many websites. Still, it is not clear whether these broken features would actually be needed by the user on most of their visits: they may be optional features which are not required at all to read or interact with the page the way the user intended to when reaching the page.

On top of this, there is no standard interface to detect whether a page element is broken, and the definition and criteria of breakage would anyway depend on its potential causes, e.g., blocking JavaScript.

This work looks at the problem of measuring web page breakage induced by JavaScript blocking through the following contributions:

(1) documenting HTML code and elements that require JavaScript to work properly,
(2) introducing a heuristic-based framework aimed at detecting potential functionality breakage introduced by blocking JavaScript, based on limited information,
(3) performing a large-scale crawl of popular web pages, including internal pages, quantifying how badly these pages are broken when JavaScript is blocked,
(4) semi-manually classifying page screenshots based on visual comparison,
(5) measuring the difference in request count, especially of tracking requests, when blocking JavaScript.

After covering some background about web tracking and advertisement relying on JavaScript and existing tooling related to this work, we introduce our bottom-up measurement approach, detailing the page features we measure and how they can be broken without JavaScript. Then, we present our web crawling methodology and the results of measuring the dependency on JavaScript of 6,384 pages. Finally, we detail some methodology limitations, discuss some misconceptions and incentives for websites to decrease their reliance on JavaScript and conclude this work.

2 BACKGROUND

In this section, we first discuss previous work that motivates the need for reducing the amount of JavaScript allowed client-side, then existing tooling that analyzes the usefulness of scripts sent to the user.

2.1 JavaScript for Advertisement and Tracking

Previous work has shown extensively how JavaScript is commonly used for advertisement and tracking, employing different techniques, both stateful and stateless. In particular, JavaScript is commonly used to handle cookies, `localStorage`, and `IndexedDB`, which do have many legitimate uses but are also notably adopted for tracking [5, 31], being standard, easy-to-use client-side storage technologies.

JavaScript is also one of the core components of browser fingerprinting [18, 30, 35–37, 41, 49, 69], a technique used to collect many different browser properties and generate a distinctive fingerprint. It can recognize the browser across sessions while storing no data on the client side, provided the fingerprint does not vary too much [38]. JavaScript also makes it easier to exploit the browser caches

for fingerprinting [32, 56]. Various security attacks targeting the browser, including timing-based attacks, are able to leak data from other processes by leveraging JavaScript for their exploitation [39], even though it is not strictly required [53].

Many defenses have been deployed in popular browsers over the past few years, but there still is an ongoing arms race between browser vendors and trackers. Even when a countermeasure is implemented, some future change can make the browser exploitable again, as highlighted by JavaScript timing exploitation in [50].

In addition, advertisement delivery heavily relies on JavaScript [19], for instance to try to circumvent adblockers [72]. One can therefore observe that JavaScript is a known vector threatening user privacy, but that existing countermeasures keep failing at proposing a sustainable solution to preserve JavaScript from undesirable exploitations.

2.2 Automatic JavaScript Removal

To the best of our knowledge, this is the first work to detect the dependency on JavaScript of web elements based on markup alone—without a differential analysis with the original page. Some previous works have investigated the selective removal of JavaScript, especially for performance purpose: Chaqfeh *et al.* have investigated the removal and replacement of non-essential JavaScript, with the aim of improving performance on low-end devices [15]. The system acts as a rewriting proxy, modifying pages to remove or replace non-essential JavaScript. The proxy does load the whole page in the first place, including all scripts, and can thus benefit from a better understanding of the page to make informed decisions. However, this also means that running this proxy locally would bring no privacy improvement. Other works that classify JavaScript usefulness based on the whole page [14] suffer from the same issues for this use case.

3 PAGE FEATURE BREAKAGE

In this section, we introduce our bottom-up analysis approach and detail the heuristics developed to measure the reliance of web page elements on JavaScript.

3.1 Breakage Detection

To investigate the reliance of web elements on JavaScript, we introduce a client-side measurement framework aimed at detecting potential functionality breakage when blocking JavaScript, relying only on the DOM state after the initial page load. This measurement framework is heavily unit-tested and is then used to measure the reliance of web page elements on JavaScript.

3.1.1 Limited Information Available. We aim to be able to detect web elements present on the page that rely on JavaScript to function. This is ultimately useful for users willing to browse with minimal JavaScript, to locate broken or incomplete page features, which may not always be easily spotted visually. Thus, the detection mechanisms are restricted to inspecting the Document Object Model (DOM) state obtained after the page is fully loaded, with scripts completely blocked—i.e., the markup received from the server. Since we must only rely on the markup of the page visited by the user, we cannot compare a possibly broken version of the page (with JavaScript blocked) with a supposedly working one (with JavaScript loaded), be it using DOM or visual analysis, since the scripts would then need to be downloaded and executed, defeating the privacy and security benefits. In this setup, there is no way of knowing, from the markup alone, whether a script is meant to attach an event listener to an element (e.g., a button) to handle its possible action.

As no programming interface is available to detect breakage of web elements caused by JavaScript blocking and based on this set of restricted information, we develop a framework of heuristics, embedded in a browser extension, that detects page features of interest and classifies them as being

either working or broken. We refrain from making any network request and from modifying the inspected page, to make it as less intrusive as possible.

Future extensions may also leverage this framework to detect and fix broken page features required by the user.

3.1.2 Bottom-up Approach: Detecting Broken Elements to Detect Broken Features. HTML documents are built as a combination of basic HTML elements. Some of them—like `<div>`—are not meant to be interactive elements, some others—like `<a>`—have a native, fully functioning behavior without JavaScript, while others—like `<canvas>`—always require JavaScript to be useful. These basic HTML elements are then combined to provide page features desired by the website developers, such as dropdown menus and accordions. In doing so, non-interactive elements are often used or misused to provide the desired interactive, complex page features (such as using a `<div>` for a button). This makes it much harder to detect broken page features.

Adding to this impediment, there is also no general interface to detect whether basic HTML elements are functionally broken, mostly because the definition and possible symptoms of breakage depend not only on the causes (e.g., blocking JavaScript), but also on the user preferences and tolerance to partial breakage. This means that custom heuristics, tailored to detect breakage induced by blocking JavaScript, are required to detect broken features. In this work, we adopt a bottom-up approach where, in place of trying to define possible symptoms of *page* breakage, we instead focus on detecting breakage of web *elements*, or combination of elements, when blocking JavaScript. Thus, breakage does not need to be defined on the whole page, only on individual web elements.

To identify possible breakage of individual web elements and of complex page features, we carried out a comprehensive analysis of standard HTML elements, which can be found in Table 4 in the appendix, and of popular web component libraries (in Table 5), aided by the Web Accessibility Initiative — Accessible Rich Internet Applications (WAI-ARIA) widget list [65] and manual browsing. We identified their respective potential reliance on JavaScript, based on standard definitions and documentation, in-the-wild observations and common knowledge of the field regarding bad practices, and derived the custom breakage detection heuristics from this analysis. We do not have to handle user interactions separately, as they are already part of the expected behavior of individual elements, and thus are already covered by our heuristics.

The remainder of this section details the most relevant page features and the basics of the breakage detection heuristics.

Images. Standard `<img>` and `<picture>` elements do not require JavaScript to work properly. The browser renders them by directly downloading the source images from the `src` and `srcset` attributes. We can therefore objectively define the breakage of an image element as:

User-perceived breakage symptoms

Image is not rendered or a low-resolution placeholder is displayed instead.

Nonetheless, some websites implement image lazy loading in JavaScript. Instead of using the standard source attributes, they store the image source URLs in other attributes, often using custom `data-*` attributes [9, 66], accessible with the `dataset` IDL attribute. Then, when the image comes close enough to the viewport, some JavaScript logic copies the URL to the appropriate `src`/`srcset` attribute to load the image as detailed in Listing 1, effectively deferring fetching the image until it is actually needed, making the initial page load faster. In this scenario, if JavaScript is blocked, no image will be rendered.

However, this behavior does not need to be implemented in JavaScript anymore since the introduction of the `"lazy"` value of the `<img>` `loading` attribute in 2019–2020 [43], which is

```
<img class="lazyload" data-src="image_js.jpg" alt="Image" width="300" height="90">
<script>
  const lazyImageObserver = new IntersectionObserver((entries) => {
    entries.forEach(entry => {
      if (entry.isIntersecting) {
        const lazyImage = entry.target;
        lazyImage.src = lazyImage.dataset.src;
        lazyImage.classList.remove("lazyload");
        lazyImageObserver.unobserve(lazyImage);
      }
    });
  });

  Array.from(document.querySelectorAll("img.lazyload"))
    .forEach(lazyImage => lazyImageObserver.observe(lazyImage));
</script>
```

Listing (1) Lazy loading images with JavaScript

```
<img src="image_nojs.jpg" alt="Image" loading="lazy" width="300" height="90">
```

Listing (2) Lazy loading images without JavaScript

Fig. 1. Lazy loading images

gradually getting adopted [9] and implements native image lazy-loading, as shown in Listing 2. This native behavior of the browser is disabled when JavaScript is disabled, to prevent tracking of the scroll position [43].

Moreover, many websites implementing image lazy loading with JavaScript use a placeholder image until the real image is loaded. This placeholder image is often a 1×1 px base64-encoded GIF image supplied inline using the `data` scheme, thus requiring no extra request, while other websites use a lower-resolution image and a CSS unblurring animation when the full-resolution image is loaded.

Finally, it should be noted that some websites implementing image lazy loading with JavaScript also provide noscript fallbacks, using the `<noscript>` elements, which are only interpreted when JavaScript is disabled in the browser, allowing the image to load if JavaScript is blocked.

We consider as *large images* all images whose height and width are both greater than or equal to 100 px.

Forms. Forms are one of the few interactive mechanisms able to operate without JavaScript when implemented properly, by relying on server cooperation.

In this work, we call *forms* document sections delimited by `<form>` elements, containing form controls. Forms are meant to collect data input by the user and to send them to a remote server when the form is submitted. Various form controls that dictate the structure and logic of the form are available, most of them implemented as a `type` of the `<input>` element, the others as separate elements, namely `<button>`, `<textarea>` and `<select>`.

User-perceived breakage symptoms

Form cannot be submitted to the server, is not submitted to the intended API endpoint, or some form values are not submitted.

To submit the form, the browser builds an HTTP request of the type defined by the `method` form attribute (`GET` by default), using the URL specified as the `action` form attribute (the current

page's URL by default) and the values of form controls having a `name`, then sends this request to the server. For the Listing 3, typing "test" into the search field and checking the checkbox, then clicking the Search button, will result in a `GET` request with the URL ending with the following query parameters: `/search?q=test&check=on`. This means that, if a form control has no `name`, its value will not be sent; a form with at least one control having no `name` necessarily requires JavaScript to handle the form, using another reference than their `name` to access the form controls and to either modify the page accordingly or send a request to the server.

Unfortunately, one cannot guarantee that the server will take the request into account, be it a `GET` or `POST` request. The absence of the `action` attribute cannot even be used to presume of the non-handling of the form by the server, as some websites—mostly search engines—do intend to submit these requests to the form page's URL, which is the default when the `action` attribute is absent.

Furthermore, the form submission itself can be broken without JavaScript. Indeed, two separate mechanisms allow to submit a form: dedicated submission form controls and the implicit submission mechanism. When some form controls (`<button type="submit">`, `<input type="submit">`, `<image type="image">`) are part of a form, they will submit the form when activated. However, a form can still be activated when none of these form controls are included in the form, using implicit submission. Implicit submission allows to submit a form by hitting a key (usually "enter") when a text control is focused and the form has at most one single-line text control [67].

Otherwise, if the form cannot be natively submitted, JavaScript is required to either modify the page according to the form data, to trigger the form submission, or to manually send a request accordingly.

```
<form action="/search">
  <input type="search" name="q">
  <label>Check:
    <input type="checkbox" name="check">
  </label>
  <button>Search</button>
</form>
```

Listing 3. A valid form

Lone Controls. In this work, we call *lone controls* all form controls that have no form owner—i.e., that are not children of any `<form>`, nor are associated to a form using their `form` attribute.

User-perceived breakage symptoms

Activating the control does not trigger the intended behavior.

Most of these lone controls require JavaScript to be useful: to attach an event listener to them or to read their value. The only lone controls not necessarily requiring JavaScript are the stateful ones, `<input type="checkbox">` and `<input type="radio">`, because their state can be accessed from CSS with the `:checked` pseudo-class, see the *Disclosure Buttons* feature and Listing 5 for details.

Empty Anchor Buttons. An `<a>` anchor button represents a hyperlink to a destination page or a section within a page. In this work, we call *empty anchor buttons* all `<a>` elements that either:

- have no `href`, `name` or `id` HTML attribute[1],
- have an `href` attribute set to the empty string,
- have an `href` attribute set to `"#"`,

[1] An `<a>` element with no `href` but with a `name` is an obsolete, HTML4 way of marking a destination for another anchor [64].

- have a `javascript:` pseudo-protocol `href` that is a no-operation, see Listing 4.

They are very often used in a discouraged way to make buttons that look like other links on the page, with the drawback that they do not convey appropriate semantics.

User-perceived breakage symptoms

Activating the anchor does not redirect to a different URL, scroll the page to the indicated part of the document or trigger the intended custom behavior.

This means they require JavaScript in the same way a `<button>` element does (see *Lone Controls*), except when these empty anchor buttons are actually used as standard-compliant go-to-top buttons. HTML5 has indeed standardized the use of the empty URL fragment and the `top` fragment as a way to ask the browser to jump to the top of the page [68].

Some empty anchor buttons are also not used as buttons, but only for appearance consistency in a list of anchors, where this anchor has no target URL.

```
<a href="#">Button #0</a>
<a href="javascript:void(0);">Button #1</a>
```

Listing 4. Empty anchor buttons

Mislinked Fragment Anchors. Similarly, we define *mislinked fragment anchors* as all `<a>` elements whose fragment is not the empty fragment but targets an element that is not found in the page—i.e., no element has the fragment as `id`.

User-perceived breakage symptoms

Activating the anchor does not scroll the page to the indicated part of the document or trigger the intended custom behavior.

Some of these elements are used as buttons, in the same way as empty anchor buttons, others are of no actual use, and would not trigger any action, even with JavaScript.

Disclosure Buttons. The constructs discussed above consist of a single HTML element, whose intended use is standardized. Here, we discuss more complex page features, combining several HTML elements.

We call *disclosure buttons* elements having a button appearance and intended to reveal and/or hide other elements when actioned, possibly concealing information to the user if broken. Disclosure buttons include accordion buttons and dropdown menu buttons, see Figure 2. They are very common features, part of most popular component libraries but are also often custom components, specifically designed for the website.

User-perceived breakage symptoms

Activating the disclosure button does not reveal/hide the associated element.

Both accordions and dropdown menus can be built without JavaScript, but popular component libraries do require JavaScript for these features to work [57, 59]: an event listener is attached to the disclosure button that shows/hides the associated disclosable element when the button is activated.

A great diversity of custom implementations can be observed in the wild, using various elements for the disclosure button (usually a `<button>`, `<a>`, `<label>` or `<div>`) as well as different positions of the disclosable element relatively to the disclosure button in the DOM tree (most of the time found as the next sibling). Disclosable elements are most often a `<ul>` (especially for dropdown

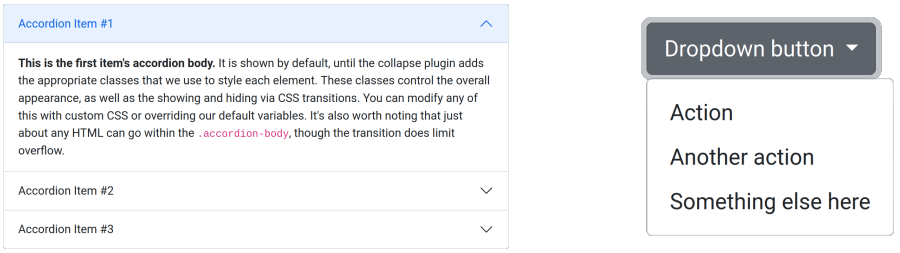

Fig. 2. Examples of disclosure buttons (from the Bootstrap 5 documentation [57, 59])

menus) or a `<div>`; they are sometimes dynamically inserted in JavaScript and thus can be missing from the initial DOM tree.

Accordions and other disclosure elements can also be created without JavaScript, using a `<label>` as a button toggling an `<input type="checkbox">`, whose checked value can be used in a CSS selector to show the disclosable element using the `:checked` pseudo-class, as shown in Listing 5.

```
<div class="accordion">
  <input id="checkbox0" type=checkbox style="position:absolute;opacity:0">
  <label class="accordion-header" for="checkbox0">Header</label>
  <div id="body0" class="accordion-collapse">Body</div>
</div>
<style>
  .accordion-collapse {
    display: none;
  }
  #checkbox0:checked ~ #body0 {
    display: block;
  }
</style>
```

Listing 5. Accordion working without JavaScript

A simple disclosure element can also use the native element pair `<details>` and `<summary>`, see Listing 6: elements in the `<details>` are hidden by default, while the `<summary>` element is shown and adjoined by an arrow suggesting some content is hidden; when clicking the `<summary>` element, the hidden content visibility is toggled.

```
<details>
  <summary>Some question</summary>
  Some detailed answer that is hidden by default
  and toggled by clicking the summary element.
</details>
```

Listing 6. Native disclosure element

The diversity of implementations and the use of non-semantic elements make it a challenge to reliably detect all disclosure buttons while minimizing the number of false positives. Using the CSS class names can sometimes help to refine the classification, especially when these are from the most popular component libraries (like `.dropdown-toggle` and `.dropdown-menu`), but this is no silver bullet, some websites using class names in the website's language, while others obfuscate the class names.

Protected E-Mails.

User-perceived breakage symptoms

E-mail addresses (and sometimes, other strings with an at sign) are replaced by `[email protected]`.

Some websites try to prevent mass harvesting of e-mail addresses by requiring JavaScript. They embed the encoded address, often visually replaced by `[email protected]`, which is then decoded and displayed in place of this message. This is an example of a feature that deliberately relies on JavaScript.

Loader Overlays. Some pages display an overlay that covers the entirety of the page until the page is loaded. We call them *loader overlays* (they are also referenced as AJAX loaders or preloaders).

User-perceived breakage symptoms

The actual page content is hidden behind an overlay, which usually features a loading spinner.

Besides going against the flow of best practices, especially progressive loading, these overlay elements are only removed with JavaScript when the page is done loading, meaning that, when JavaScript is disabled, the overlay is never hidden and makes it impossible to read the page content, actually often properly loaded, even without JavaScript.

Loader overlays usually appear as a `<div>` and as a direct child of the `<body>` element (often being its first child) and have the `id` `"preloader"` or a `class` containing the word `"preloader"`.

Page Text.

User-perceived breakage symptoms

The page has no text content.

This heuristic checks whether the `<body>` has some text content (using the `textContent` and `innerHTML` properties), other than whitespace.

This is particularly useful when the page is a full-page app. Full-page apps are websites where the whole page is nested in a single element that is completely populated in JavaScript. Without JavaScript, the page is left blank and broken.

This heuristic applies to the whole page and is an all-or-nothing heuristic.

Stylesheets Loaded.

User-perceived breakage symptoms

The page has no style loaded.

Finally, this heuristic detects if the page has at least some basic stylesheet loaded by checking font styles of a few elements, especially headers, which are almost always changed by websites.

This heuristic applies to the whole page and is an all-or-nothing heuristic as well.

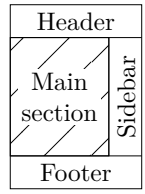

Fig. 3. Common page structure (the sidebar can be on either side of the page)

3.2 Page Feature Relevance

A web page is composed of different sections with each their own purposes, as can be seen in Figure 3. The header is often used to guide the navigation on a site whereas the main section offers the bulk of the site content. In order to provide a more focused analysis of page breakage, we look to identify these sections in the pages that we crawled.

These sections can be inferred from the DOM tree, using tag names, element `id`s, and class names. In particular, HTML5 provides semantic elements for these sections: `<header>`, `<footer>`, `<aside>`, and `<main>`. Not all websites use these semantic elements, but many do mark these sections with a combination of non-obfuscated `id`s and class names, which makes it possible to classify page elements according to their position and intended purpose.

In addition, some page features may never be accepted by the user, especially tracking and advertisement features, which do require JavaScript most of the time.

4 DATA COLLECTION

This section details the crawling methodology we implemented to measure the dependency of web page elements on JavaScript at scale.

4.1 Crawled Websites and Pages

We used a subset of the Hispar list [12], which provides a list of popular webpage URLs, including landing and internal ones. The Hispar list is built using the Alexa Top 1M domain list, querying the paid API of Google Search with the query `site:`ω for each domain ω and storing the first 50 search results along with their ranking, stopping as soon as the final list has 100,000 URLs. The user's location is set to the United States and results are limited to pages in English. We used the most recent list that was released at the end of January 2021 [13], and focused on the first three URLs (based on Google Search rankings) of each domain, which usually include the landing page and two internal pages. We limit the crawl to two internal pages because pages of lower ranks are very often of the same type—e.g., product pages of a shopping website. This makes up for 6,384 pages to crawl, from 3,774 domains (2,136 Alexa base domains). Visiting internal pages is very important in this study, since page features may be drastically different between the landing page and internal ones—e.g., the website could feature a carousel on the landing page and forms in the internal pages.

We implemented the breakage detection heuristics detailed in subsection 3.1 as a JavaScript library intended to be used as part of a browser WebExtensions extension. This library detects page elements matching features of interest and classifies them as either being working or broken.

Web crawling is then automated with Puppeteer, using Firefox Nightly 88.0a1, required for Puppeteer compatibility. The crawl uses two instances of Firefox running at the same time, with the breakage-detection extension loaded: one is using default preferences while the other has `javascript.enabled` set to `false`, which disables JavaScript globally for this instance, as depicted

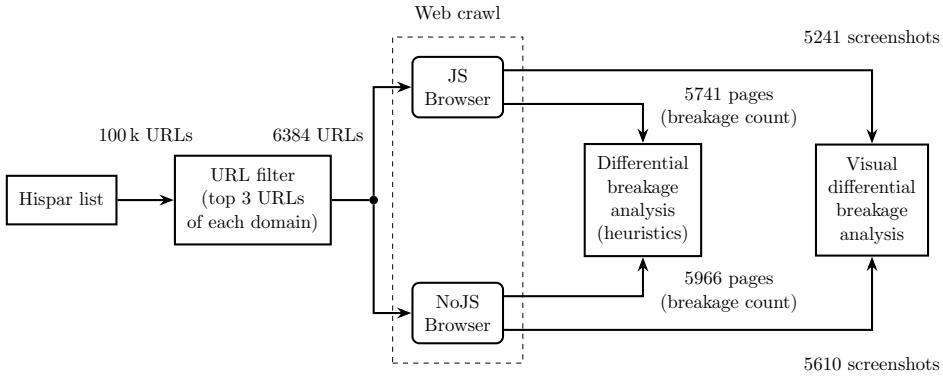

Fig. 4. Dataflow diagram

in Figure 4. For each page URL to crawl, the page is requested in a new tab in each browser instance at the same time and the following steps are followed:

(1) load the web page (with a 30 s timeout) then wait for 3 s,
(2) inspect the loaded web page for feature breakage,
(3) save a page screenshot and the current DOM state as HTML.

The browser tab is then closed and the crawl moves to the next page URL as soon as both instances are done with the current one. We have checked, with the help of the generated screenshots, that 3 s were enough for lazy-loaded content to be loaded. The crawl was run from our campus network, from 2021-03-26 to 2021-03-28. If the `DOMContentLoaded` event (indicating the initial HTML document has been loaded) is not fired after 10 s, the URL is skipped for this instance; if the `load` event is not fired after 30 s, the URL is skipped as well. The crawl waits for 3 s to leave enough time for asynchronous content to load, especially stylesheets and lazy-loaded images. Finally, the extension is given 60 s for the breakage-detection inspection.

The crawl with default preferences is hereafter referred to as `[plain]`, while the one with JavaScript disabled as `[nojs]`. For both these crawls and for each page, the following data are collected:

- *counts of broken and working elements* for each page feature, in the main section and in the whole page, and whether these elements are visible or not, forming a JSON feature report,
- *a page screenshot* for further, manual analysis,
- *the page DOM*, as an HTML file.

4.2 Collection of Website Categories

To discover and highlight breakage disparities between website categories, we adopted the classification from OpenDNS [46], which uses a crowdsourcing system to attribute categories to domains. Registered users can propose domains and vote on categories. To the best of our knowledge, it is the only website classification service that provides definitions of categories (as this is required for crowdsourcing, definitions are available at [47]) and that can attribute several categories to each website—i.e., categories are not disjoint. This comes with the drawback that only about 66 % of domains from the dataset are covered. Some highly correlated categories are merged into new ones to improve readability.

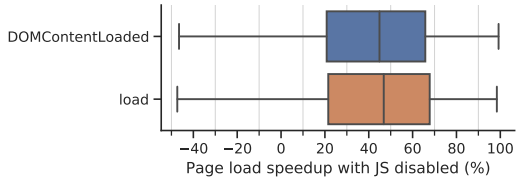

Fig. 5. Speedup when blocking JavaScript

5 RESULTS

In this section, we report the results of crawling the subset of the Hispar list built above: 6,384 pages from 3,774 domains (2,136 Alexa base domains).

5.1 Dataset Description

Table 1 presents the data collected during the crawl, and the result of each crawl step. Out of the 6,384 pages crawled, 5,827 pages were successfully loaded, feature breakage data was collected for 5,695 pages and the HTML was saved for 4,911 pages for the same crawl URL for both `[plain]` and `[nojs]` crawls.

Table 1. Success statistics for each crawl step

Crawl type Page	[plain]	[nojs]	both
Crawled	6384	6384	6384
Loaded (1)	5875 (92 %)	6119 (96 %)	5827 (91 %)
Inspected (2)	5741 (90 %)	5966 (93 %)	5695 (89 %)
HTML saved (3)	5241 (82 %)	5610 (88 %)	4911 (77 %)

Inability to load and process some of the pages can be attributed to various factors, including region-based blocking [62], possibly outdated URLs of the Hispar list at the time of crawl or load and processing timeouts due to long synchronous JavaScript rendering times and heavy page JavaScript processing.

5.2 Effect on Page Load Time

Blocking JavaScript brings up a significant page load speedup for most pages, as shown in Figure 5. In our dataset, the median load time with JavaScript enabled is 3,173 ms, while it is reduced to 1,582 ms when disabling JavaScript. This can be attributed to JavaScript files not being downloaded, parsed and executed, and to some content not being loaded by JavaScript, especially lazy-loaded images that do not provide a fallback.

Some pages are however slower to load, because of the event we used to detect the page load completion. The `load` event is fired by the browser as soon as the whole page is loaded, but this does not account for content downloaded in JavaScript, such as lazy-loaded images or font files. Moreover, as image lazy loading is not possible when JavaScript is disabled in the browser (the `loading="lazy"` standard behavior is disabled by the browser to prevent tracking), image load time is then included in this `load` event, explaining the higher load times measured when JavaScript is disabled.

Table 2. Ratios of pages where the following CSS selectors match at least one element

Selector	Page count	Ratio (%)
`main`	1557	27.6
`main, #main, .main`	2128	37.8
`main, header, footer`	3653	64.8
`main, header, footer, aside, nav, section, article`	4097	72.7
`main, #main, .main, header, #header, .header, footer, #footer, .footer`	4311	76.5

5.3 Page Section Classification

In this section, we validate the possibility of discovering the different sections of a page from the standard, semantic metadata built in the HTML document. Notably, we look for either explicit HTML elements, `ids`, or classes that relate to the structure of a page.

Table 2 details the distribution of pages from the dataset for which we were able to identify at least one of the specified selectors (*element*, *#id*, *.class*). The ratio of pages having a `<main>` element matches 2021 HTTP Archive's findings very well [11]. Around 37 % of pages have clearly identifiable main sections, while more than 76 % of pages mark the main section, header or footer of the page, making it possible to recover the remaining main section. In the following, when the main section cannot be determined, the whole page is considered as being the main section.

5.4 Page Feature Breakage

Then, we report about page dependency on JavaScript and we detail observed feature breakage when disabling JavaScript.

We start by introducing the quantities used to quantify feature breakage when blocking JavaScript. First, the differential breakage (DBR) of a page feature is the difference between the counts of broken elements with JavaScript disabled and with JavaScript enabled :

$$\mathrm{DBR}_{\mathrm{feat}_{\mathrm{page}}} = \mathrm{BrokenCount}^{\texttt{[nojs]}}_{\mathrm{feat}_{\mathrm{page}}} - \mathrm{BrokenCount}^{\texttt{[plain]}}_{\mathrm{feat}_{\mathrm{page}}} \tag{1}$$

A high, positive DBR denotes that the page has many more broken elements (of the relevant feature) when blocking JavaScript than with JavaScript enabled. The differential breakage can become negative when the `[nojs]` page has less elements detected as broken than the `[plain]` page. This may happen in different scenarios, including when:

- elements that are considered as broken are dynamically added in JavaScript (e.g., in single-page applications) and are thus not present in the `[nojs]` page,
- elements are hidden and only become visible with JavaScript (if the differential breakage is restricted to visible elements only),
- the page provides noscript fallbacks which are detected as working while the `[plain]` page has elements that cannot be detected as working or are actually broken.

Then, since elements are labeled as either broken or working, the total count of a feature is the sum of their counts:

$$\mathrm{TotalCount}^{\texttt{[nojs]}}_{\mathrm{feat}_{\mathrm{page}}} = \mathrm{BrokenCount}^{\texttt{[nojs]}}_{\mathrm{feat}_{\mathrm{page}}} + \mathrm{WorkingCount}^{\texttt{[nojs]}}_{\mathrm{feat}_{\mathrm{page}}} \tag{2}$$

Finally, the normalized differential breakage (DBRn) can be derived from these two quantities:

$$\mathrm{DBRn}_{\mathrm{feat}_{\mathrm{page}}} = \begin{cases} \dfrac{\mathrm{DBR}_{\mathrm{feat}_{\mathrm{page}}}}{\mathrm{TotalCount}^{\texttt{[nojs]}}_{\mathrm{feat}_{\mathrm{page}}}} & \text{if } \mathrm{TotalCount}^{\texttt{[nojs]}}_{\mathrm{feat}_{\mathrm{page}}} \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

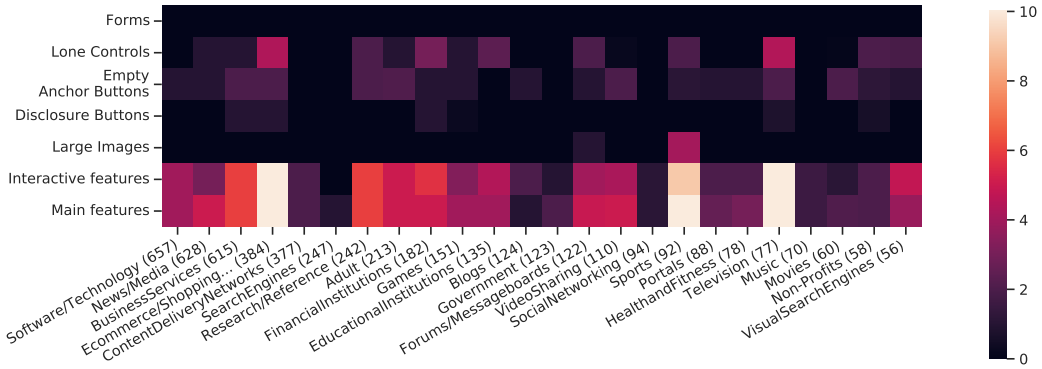

Fig. 6. Color represents the 90th percentile of differential breakage of visible elements in the *main section*. Lighter shades denote higher differential breakage. Only categories with more than 50 pages in the dataset are plotted, to improve readability. This highlights the disparity of breakage across website categories.

5.4.1 Aggregated Features. To ease the interpretation of elementary page features, we define two aggregate features as the following unions:

Interactive features = { Lone Controls, Forms, Empty Anchor Buttons, Mislinked Fragment Anchors, Disclosure Buttons }

and

Main features = { Page Text, Stylesheets Loaded, Interactive features, Large Images, Loader Overlays }.

As not all the features included in the *Main features* have the same weight (the fact that no large image is broken does not matter if the page has no text content), the maximum of each metric is taken when aggregating, so that the *Main features* feature gives the more reasonable classification of whether the page is broken or not.

5.4.2 Disparity Across Website Categories. Figure 6 reports on the 90th percentile of differential breakage of visible elements *from the main section* for each feature and across page website categories. This figure highlights the disparity of breakage observed in the main section across website categories. In particular, e-shopping pages have more broken interactive features in the main section than blogs have, mostly because they contain more interactive elements in this section (25.5 interactive elements for `[plain]` e-shopping pages and 8.5 interactive elements for `plain` blogs on average, in the main section): e-shopping pages can have an order form and other interactive elements (to build light boxes for example) on a product page, for instance. Figure 9 from the appendix takes into account the whole page for comparison.

5.4.3 Dependency of Main Page Features on JavaScript.

Reliance on JavaScript. Every crawled page has at least one `<script>` with `type text/javascript`. The page with the highest count of these tags has almost 300 of them.

Feature Breakage Report. Figure 7 reports on the proportions of pages for each page feature possible status, indicating if these features are broken or working, or if there are no elements matching this feature (which is thus not broken). It can be seen that 67 % of pages have all their main features from the main section working, which means they are likely to be useful to the user, even with JavaScript disabled. Even when taking the whole page into account, 43 % of pages have all their main features still working, meaning that the whole page is likely working as intended, when

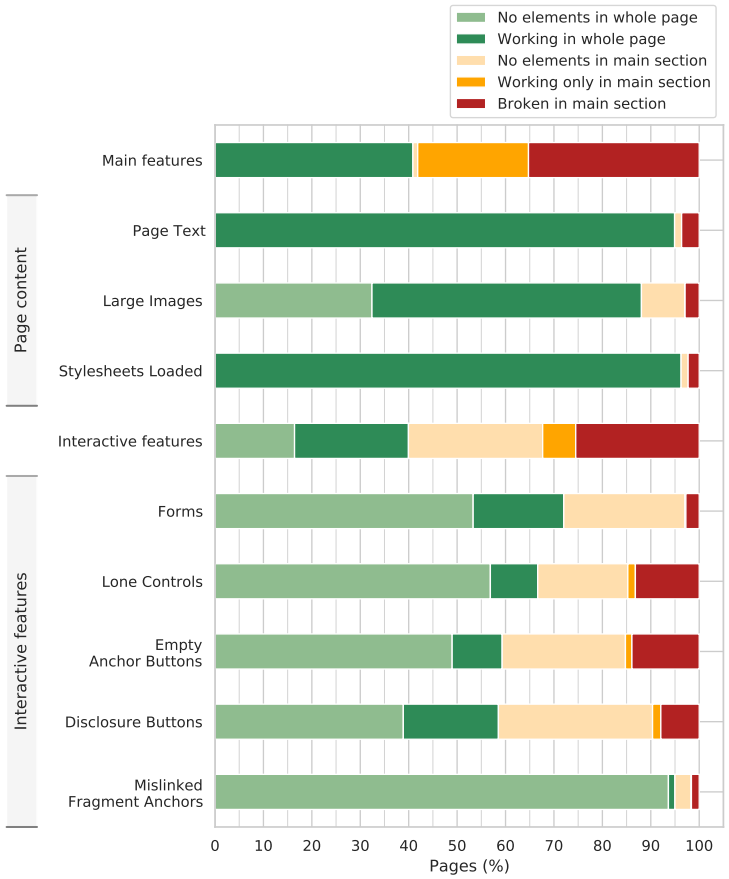

Fig. 7. Shares of pages for each page feature status; when present, a feature can either be working in the whole page, only in the main section or broken in the main section

blocking JavaScript. This difference between the main section and the whole page is easily explained by the fact that, on many pages, interactive features are used around the main section—e.g., for navigation— not in the actual content, as quantified in Figure 7. However, it should also be noted that for around 25 % of pages, at least one element from an interactive feature found in the main section is broken, which could impede the user from using the page as intended.

Feature Implementation Consistency. Figure 8 depicts the normalized differential breakage (DBRn) of visible elements of the main section, a high DBRn meaning that most of the elements matching a feature are broken. Figure 8 highlights that, when an elementary interactive feature is at least partially broken on a page, it is actually completely broken most of the time (short transitions between 0 % and 100 % DBRn)—i.e., all elements of this feature are broken. Beyond the fact that the main section usually contains few interactive elements, this can be attributed to the fact that the implementation of the different elements of a feature on a given page is likely to be the same, especially when the page is using a UI framework (e.g., Bootstrap [60]), that standardizes the implementation. In other words, for a given feature, it is very unlikely that a given page has elements that require JavaScript while others do not.

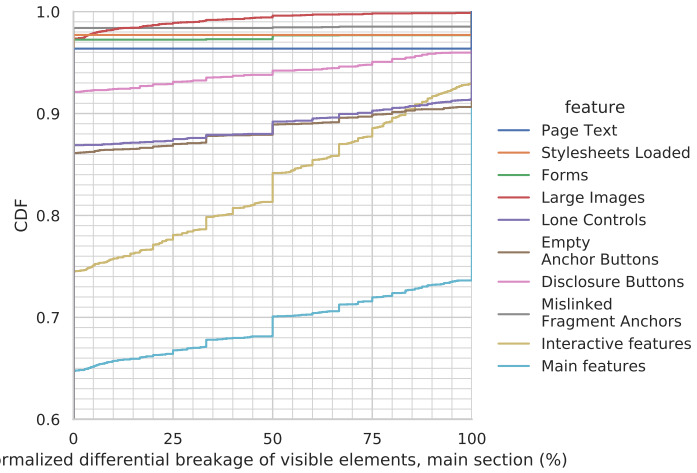

Fig. 8. Normalized differential breakage (DBRn) of visible elements of the *main section*; negative normalized differential breakage is not shown for readability

6 LIMITATIONS

In this section, we detail the limitations of the heuristic framework and of our crawling methodology.

6.1 Measurement Framework Limitations

6.1.1 Limited Information Available. Since our heuristic-based measurement is intended to only rely on information available when no JavaScript is loaded, it is limited to what is discoverable based on the initial state of the DOM only. It is thus unable to detect the reliance on JavaScript of elements whose such reliance cannot be derived form the markup, mostly some cases of misuse of non-semantic elements, e.g., `<div>`s used to build buttons.

6.1.2 User Expected Feature Granularity. The heuristics that comprise the measurement framework are derived from basic HTML elements and common components, following a bottom-up approach to detect the reliance of web elements on JavaScript. In some cases, this might not match the level of granularity of features expected by users, the framework being in those cases more fine-grained—e.g., only buttons of a modal would be reported broken, not the modal itself. This limitation results from the limited information available and from the web platform, since the relationship between elements is usually not automatically discoverable based on static analysis—e.g., it is usually impossible to discover that a button is used to close a modal, based on markup alone.

6.2 Crawl Limitations

The measurement crawl is limited to three URLs per domain, which does not cover the whole site, but we believe this still provides reasonable insight about reliance on JavaScript, especially because many webpages from a website actually follow the exact same template. In addition, the crawl is run from a single location, from a single device, on desktop, but we expect few differences on mobile since the usage of JavaScript is similar [10], except for components only shown on mobile, such as hamburger menus. We also expect very few differences between browsers and devices, since we browse without JavaScript, the interface of the web platform being very similar when JavaScript is not enabled.

7 DISCUSSION

In this section, we discuss some misconceptions about current reliance on JavaScript, the viability of JavaScript blocking, and incentives for website owners for making their websites usable without JavaScript.

7.1 Manual Visual Analysis of Screenshots

To back up these results, which can seem to go against the common assumption that web pages are utterly broken without JavaScript, we manually labeled page full-page screenshots collected during the crawls, according to their breakage seriousness.

To this end, we first automatically compared all `[plain]` and `[nojs]` page full-page screenshots pixel-to-pixel, automatically detecting and disregarding most of the differences due to vertical shift of content between the two versions. Screenshots were then put into 10 %-wide bins according to their pixel-to-pixel difference percentage. As pixel-to-pixel difference would greatly overestimate the actual, user-perceived page difference, and, in some cases, underestimate the user-perceived difference on pages where the background takes the most room, one of the author manually labeled 150 random samples in each of the four bins with the smallest difference, according to the amount of information lost to the user when disabling JavaScript, by visual comparison only. Missing advertisements, cookie banners or presentational-only images (not bringing specific information) are not considered information loss, while missing information-heavy images or substantial layout breakage are.

Based on the labeling we conducted and using the pixel-to-pixel difference bins previously constructed, we concluded that at least 50 % of pages feature no substantial amount of information loss—such as missing content text or figure that could mislead the user—when blocking JavaScript. This lower bound is negatively impacted by page differences resulting in a significant pixel-to-pixel difference percentage while actually making the page more usable when blocking JavaScript, a major example being semi-transparent cookie-consent overlays that cover the entire viewport, which do not appear when blocking JavaScript.

7.2 JavaScript Reliance of Most Visited Websites and Component Framework Trends

The common assumption that web pages heavily depend on JavaScript may stem from the fact that most visited websites do heavily rely on JavaScript. For instance, YouTube and other Google products, Twitter and Instagram are largely unusable when JavaScript is blocked: e.g., the YouTube landing page displays only loading placeholders in that case.

Moreover, deployment of relatively recent JavaScript client-side component frameworks such as React or Vue.JS, which may result in a blank page without JavaScript, is misrepresented by developer trends. These projects are indeed among the top 10 repositories on GitHub based on star rankings [34], but constitute only a very small share of websites actually deployed, React and Vue.JS accounting respectively for less than 3 % and 1 % of websites monitored by W3Techs [48].

7.3 Benefits and Viability of Aggressive JavaScript Blocking

On top of the numerous privacy and security benefits introduced in Section 1 and Section 2, disabling JavaScript brings additional changes of different natures.

7.3.1 Tracking Reduction Benefits. To understand the impact of disabling JavaScript on a user's online footprint, we performed a new crawl on the same set of URLs where we logged the number of requests triggered by each web page. We leverage the `onBeforeRequest` handler [44] of Firefox to classify each URL in real-time. The classification relies on Firefox Enhanced Tracking Protection

Table 3. Mean request count and standard deviation for each request type with JavaScript enabled and disabled

	[plain]		[nojs]		Mean change (%)
	M	SD	M	SD	
Request count					
All	72.6	59.1	28.3	34.0	-61.0
— Non-tracking	50.9	44.1	25.1	33.0	-50.7
— Tracking	21.7	29.8	3.3	8.1	-85.0
First party	27.7	31.3	16.4	24.5	-40.8
— Image	13.3	21.7	12.3	22.2	-7.6
— Stylesheet	2.6	5.0	2.4	4.6	-6.2
— Font	1.5	2.6	1.3	2.4	-10.2
— Script	7.1	11.4	0.0	0.0	-100.0
— XHR	2.5	5.7	0.1	0.7	-97.6
Third party	44.9	49.6	11.9	26.4	-73.4
— Image	15.8	26.4	8.5	24.7	-46.3
— Stylesheet	2.1	3.6	1.4	3.5	-30.5
— Font	2.3	3.7	1.4	2.6	-38.2
— Script	16.8	19.7	0.0	0.0	-100.0
— XHR	5.5	8.6	0.0	0.2	-99.8

and the built-in Disconnect lists [7] to indicate if a URL is either first or third party and whether it is involved in tracking or not.

Table 3 details the result of this new crawl. Considering all types of requests, blocking JavaScript presents a mean reduction of 61 % and this percentage is even higher at 85 % for tracking requests. This shows how beneficial disabling JavaScript can be when it comes to tracking. Looking at the difference between first and third party requests, we can notice a difference in the type of loaded resources. While images and stylesheets are mostly loaded in a first party context, scripts mostly come from third parties and blocking JavaScript here has a drastic impact, as these are never loaded in the browser. Some XHRs are also preloaded using `<link rel="preload" as="fetch" src=". . . ">`, which explains why a few XHRs are still detected with JavaScript disabled.

7.3.2 Browsing Comfort. In the case where the page is usable enough without JavaScript, having it disabled can actually improve the browsing experience by reducing the amount of aggressive page behaviors, such as pop-up advertisements, newsletter forms or unexpected animations, resulting in less obstructed browsing. In particular, cookie banners, that ask for user consent, are usually not shown in this scenario since they are often completely managed with JavaScript cookie consent frameworks, that set the cookies in JavaScript.

7.3.3 Reduced Data Size. Since external scripts are not loaded when JavaScript is disabled, this can result in a sizable reduction of transferred data; the HTTP Archive reporting a median size of 444 kB of JavaScript per page [8].

Reducing the amount of transferred data has several benefits, including faster page loads for most pages (see subsection 5.2), reduced connection-related energy consumption, especially on mobile devices, and reduced cost for users with a data cap on their mobile plan. However, in some cases, disabling JavaScript could actually result in higher bandwidth usage, as it would prevent loading some pieces of content only when actually needed, as with lazy-loaded images. For instance,

since all images would be loaded disregarding of the scroll position, it could happen that the user would actually leave the page without ever reaching its bottom where images would have been needlessly downloaded.

7.3.4 Reduced Client-Side Processing Load. Reducing the amount of scripts processed on the client side also reduces the processing stress on the end device. This results in a lower consumption which, especially on mobile devices, can increase battery life and device life span due to reduced heating and battery stress, reducing e-waste. Varvello and Livshits have tested 15 Android browsers and found that ad blocking can offer between 20 % to 40 % of battery savings with an additional 10 % when dark mode is enabled [63].

7.3.5 Impediments to Non-JavaScript Browsing. Despite all these benefits and while around two thirds of web pages are likely to be satisfactorily usable, it is currently hard to recommend browsing the web with JavaScript disabled for all websites, as it would still significantly reduce the number of websites the user could satisfactorily browse, besides the fact that the user may be required to use some websites (e.g., work-related or government websites) that do require JavaScript.

Browser extensions such as uBlock Origin [27], NoScript [40] and the unmaintained uMatrix [26], allow users to selectively enable or disable JavaScript for each domain (and even in a more fine-grained way for some of them), but they require manual action for each visited site, technical knowledge from the user and may not be easily usable on mobile because of reduced screen size.

7.4 Website Usage of JavaScript Features With No Fallback

7.4.1 Negative Impact of UI Frameworks.

Minimum Developer Boilerplate. The fact that some interactive components are not usable without JavaScript is often due to them not being written from scratch specifically for the page where they will be used, but instead coming from a UI framework, be it an open-source or an in-house one. To make them easy to use, these UI frameworks rely on a set of CSS classes to apply on elements to style and define their behavior. For instance, Bootstrap only requires a couple of classes (`.dropdown-toggle` and `.dropdown-menu`) and a few attributes to be added to a button and a list so that they behave as a dropdown list [59]. Unlike the implementation from Listing 5, there is no need for the developer to manually insert an extra `<input>` to keep state, since the toggle behavior is entirely handled in JavaScript, thus requiring minimum manual boilerplate. Bootstrap explicitly documents that its components do not fall back gracefully without JavaScript, and leaves the implementation of noscript fallbacks to the developer [61], only hinting at displaying a noscript warning to tell the user that JavaScript is required. The same strategy is followed by other UI frameworks, such as ZURB Foundation [70] or Semantic UI [1].

Front-End Frameworks. This trend is exacerbated by the use of front-end frameworks, which require client-side JavaScript to deliver a significant part of the user interface. When using frameworks such as React [29], Vue.JS [71], AngularJS [20], or Svelte [4], which all, by default, rely on client-side JavaScript to build interface components, it may be tempting not to provide any fallbacks since the website is very likely to be significantly broken anyway, regardless of best practices, especially Progressive Enhancement [22], which recommends separating page semantics from interactivity, while making the former as robust and accessible as possible.

Server-Side Rendering/Static Site Generation. Some of these frameworks can be used as part of an SSR stack, such as Next.js [45] (for React) or NuxtJS [3] (for Vue.JS), able to render the page on the server, before sending it to the client, sparing it from the rendering burden and dependency on JavaScript for rendering the components. SSG can also sometimes be used to render pages ahead of

time, when they do not depend on user data, thus reducing the server load. However, this plays no role in providing client-side fallbacks for interactive elements, such as dropdowns, accordions, or forms.

7.4.2 Search Engine Optimization Motivation. A point of interest for websites to provide JavaScript fallbacks, at least for basic content, is to improve their ranking on search engines. Because it is an expensive operation, many search engines and social media crawlers do not execute JavaScript at all [17, 24], possibly completely missing the page content if it is not rendered without JavaScript, thus reducing the chance for the page to be properly indexed by the search engine. Some of the most popular search engines do run JavaScript, like Google Search (since at least 2014) [21, 25].

8 CONCLUSION

In this paper, we performed a crawl on 6,384 pages and quantified the use and reliance on JavaScript of websites to provide content and features that the user was likely to expect when reaching the page. We found that 43 % of pages were very likely to be completely working with JavaScript disabled and that more than 67 % were likely to be usable enough, when potentially broken elements were not part of the main section. We also observed that reliance on JavaScript was dependent on the website category, and that it could be really low for some categories, that do not rely much on interactive features. We finally detailed reasons for why it would be beneficial for websites to be non-JavaScript friendly and focused on possible reasons for which websites may not currently be supporting non-JavaScript users. Based on these findings, we suggest that future work explores solutions to ease adoption of non-JavaScript friendly implementations of interface features, when it is possible, in accordance with Progressive Enhancement principles.

AVAILABILITY

We make available the complete crawl infrastructure and all breakage detection heuristics at https://archive.softwareheritage.org/browse/origin/https://gitlab.inria.fr/Spirals/breaking-bad.git.

ACKNOWLEDGMENTS

This project is funded by the Hauts-de-France region in the context of the ASCOT project of the STaRS framework.

REFERENCES

[1] 2019. *Semantic UI Documentation.* Retrieved 2021-03-17 from https://semantic-ui.com/introduction/getting-started.html

[2] 2021. *About Tailwind Elements.* Retrieved 2021-08-10 from https://tailwind-elements.com/

[3] 2021. *NuxtJS Homepage.* Retrieved 2021-04-26 from https://nuxtjs.org/

[4] 2021. *Svelte Website.* Retrieved 2021-04-26 from https://svelte.dev/

[5] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juárez, Arvind Narayanan, and Claudia Díaz. 2014. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM, 674–689. https://doi.org/10.1145/2660267.2660347

[6] AliceWyman, Wesley Branton, Joni, tomrittervg, and user1632815. 2021. *Mozilla Support — Firefox's protection against fingerprinting.* Retrieved 2021-05-05 from https://support.mozilla.org/en-US/kb/firefox-protection-against-fingerprinting

[7] AliceWyman, Michele Rodaro, Joni, Marcelo Ghelman, Lamont Gardenhire, Jeff, Angela Lazar, PGGWriter, Samuele-grice@mymail.com, and Fabi. 2021. *Mozilla Support — Enhanced Tracking Protection in Firefox for desktop.* Retrieved 2021-05-05 from https://support.mozilla.org/en-US/kb/enhanced-tracking-protection-firefox-desktop

[8] Web Almanac and contributors. 2020. *HTTP Archive Web Almanac — JavaScript Usage.* Retrieved 2021-04-25 from https://almanac.httparchive.org/en/2020/javascript#how-much-javascript-do-we-use

[9] Web Almanac and contributors. 2020. *HTTP Archive Web Almanac — `data-*` attributes.* Retrieved 2021-03-18 from https://almanac.httparchive.org/en/2020/markup#data--attributes
[10] Web Almanac and contributors. 2022. *HTTP Archive Web Almanac — How much JavaScript do we load?* Retrieved 2022-05-23 from https://almanac.httparchive.org/en/2021/javascript#how-much-javascript-do-we-load
[11] Web Almanac and contributors. 2022. *HTTP Archive Web Almanac — Markup.* Retrieved 2022-05-30 from https://almanac.httparchive.org/en/2021/markup#main
[12] Waqar Aqeel, Balakrishnan Chandrasekaran, Anja Feldmann, and Bruce M. Maggs. 2020. On Landing and Internal Web Pages: The Strange Case of Jekyll and Hyde in Web Performance Measurement. In *IMC '20: ACM Internet Measurement Conference, Virtual Event, USA, October 27-29, 2020*. ACM, 680–695. https://doi.org/10.1145/3419394.3423626
[13] Waqar Aqeel, Balakrishnan Chandrasekaran, Bruce Maggs, and Anja Feldmann. 2021. *Hispar List — Archive.* Retrieved 2021-05-05 from https://hispar.cs.duke.edu/archive/hispar-list-21-01-28.zip
[14] Moumena Chaqfeh, Muhammad Haseeb, Waleed Hashmi, Patrick Inshuti, Manesha Ramesh, Matteo Varvello, Fareed Zaffar, Lakshmi Subramanian, and Yasir Zaki. 2021. To Block or Not to Block: Accelerating Mobile Web Pages On-The-Fly Through JavaScript Classification. *CoRR* abs/2106.13764 (2021). arXiv:2106.13764 https://arxiv.org/abs/2106.13764
[15] Moumena Chaqfeh, Yasir Zaki, Jacinta Hu, and Lakshmi Subramanian. 2020. JSCleaner: De-Cluttering Mobile Webpages Through JavaScript Cleanup. In *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen (Eds.). ACM / IW3C2, 763–773. https://doi.org/10.1145/3366423.3380157
[16] Quan Chen, Peter Snyder, Benjamin Livshits, and Alexandros Kapravelos. 2021. Detecting Filter List Evasion With Event-Loop-Turn Granularity JavaScript Signatures. In *IEEE Symposium on Security and Privacy*.
[17] Rachel Costello. 2019. *How JavaScript Rendering Works.* Retrieved 2021-05-06 from https://www.deepcrawl.com/knowledge/ebooks/javascript-seo-guide/how-javascript-rendering-works/
[18] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. 2018. The Web's Sixth Sense: A Study of Scripts Accessing Smartphone Sensors. In *Proceedings of the 25th ACM Conference on Computer and Communication Security (CCS)*. ACM. https://doi.org/10.1145/3243734.3243860
[19] Zainul Abi Din, Panagiotis Tigas, Samuel T. King, and Benjamin Livshits. 2020. PERCIVAL: Making In-Browser Perceptual Ad Blocking Practical with Deep Learning. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 387–400. https://www.usenix.org/conference/atc20/presentation/din
[20] Google. 2018. *AngularJS Website.* Retrieved 2021-04-26 from https://angularjs.org/
[21] Google. 2019. *Making JavaScript and Google Search work together.* Retrieved 2021-05-06 from https://web.dev/javascript-and-google-search-io-2019/
[22] Google. 2019. *web.dev — Without JavaScript.* Retrieved 2021-04-26 from https://web.dev/without-javascript/
[23] David Gugelmann, Markus Happe, Bernhard Ager, and Vincent Lenders. 2015. An Automated Approach for Complementing Ad Blockers' Blacklists. *Proc. Priv. Enhancing Technol.* 2015, 2 (2015), 282–298. https://doi.org/10.1515/popets-2015-0018
[24] Bartosz Góralewicz. 2017. *Going Beyond Google: Are Search Engines Ready for JavaScript Crawling & Indexing?* Retrieved 2021-05-06 from https://moz.com/blog/search-engines-ready-for-javascript-crawling
[25] Erik Hendriks, Michael Xu, and Kazushi Nagayama. 2014. *Understanding web pages better.* Retrieved 2021-04-26 from https://webmasters.googleblog.com/2014/05/understanding-web-pages-better.html
[26] Raymond Hill. 2020. *uMatrix Repository.* Retrieved 2021-05-10 from https://github.com/gorhill/uMatrix
[27] Raymond Hill. 2021. *uBlock Origin Repository.* Retrieved 2021-05-10 from https://github.com/gorhill/uBlock/
[28] Johann Hofmann and Tim Huang. 2021. *Mozilla Hacks — Introducing State Partitioning.* Retrieved 2021-05-05 from https://hacks.mozilla.org/2021/02/introducing-state-partitioning/
[29] Facebook Inc. 2021. *React Website.* Retrieved 2021-04-26 from https://reactjs.org/
[30] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. 2021. Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1143–1161. https://doi.org/10.1109/SP40001.2021.00017
[31] Jordan Jueckstock, Peter Snyder, Shaown Sarker, Alexandros Kapravelos, and Benjamin Livshits. 2022. Measuring the Privacy vs. Compatibility Trade-off in Preventing Third-Party Stateful Tracking. In *WWW '22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022*, Frédérique Laforest, Raphaël Troncy, Elena Simperl, Deepak Agarwal, Aristides Gionis, Ivan Herman, and Lionel Médini (Eds.). ACM, 710–720. https://doi.org/10.1145/3485447.3512231
[32] Soroush Karami, Panagiotis Ilia, and Jason Polakis. 2021. Awakening the Web's Sleeper Agents: Misusing Service Workers for Privacy Leakage. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, San Diego, California, USA, February 21-24, 2021*. The Internet Society.
[33] Eiji Kitamura. 2020. *Google Developers — Gaining security and privacy by partitioning the cache.* Retrieved 2021-05-05 from https://developers.google.com/web/updates/2020/10/http-cache-partitioning
[34] Takashi Kokubun. 2021. *GitHub Ranking — Repositories Ranking.* Retrieved 2021-08-11 from https://gitstar-ranking.com/repositories

[35] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. 2020. Browser Fingerprinting: A Survey. *ACM Trans. Web* 14, 2 (2020), 8:1–8:33. https://doi.org/10.1145/3386040

[36] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. 2016. Beauty and the Beast: Diverting modern web browsers to build unique browser fingerprints. In *37th IEEE Symposium on Security and Privacy (S&P 2016)*. San Jose, United States. https://hal.inria.fr/hal-01285470

[37] Pierre Laperdrix, Oleksii Starov, Quan Chen, Alexandros Kapravelos, and Nick Nikiforakis. 2021. Fingerprinting in Style: Detecting Browser Extensions via Injected Style Sheets. In *30th USENIX Security Symposium*. Online, France. https://hal.archives-ouvertes.fr/hal-03152176

[38] Song Li and Yinzhi Cao. 2020. Who Touched My Browser Fingerprint?: A Large-scale Measurement Study and Classification of Fingerprint Dynamics. In *IMC '20: ACM Internet Measurement Conference, Virtual Event, USA, October 27-29, 2020*. ACM, 370–385. https://doi.org/10.1145/3419394.3423614

[39] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. 2017. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In *Computer Security – ESORICS 2017*, Simon N. Foley, Dieter Gollmann, and Einar Snekkenes (Eds.). Springer International Publishing, Cham, 191–209.

[40] Giorgio Maone. 2021. *NoScript Repository*. Retrieved 2021-05-10 from https://github.com/hackademix/noscript

[41] Francesco Marcantoni, Michalis Diamantaris, Sotiris Ioannidis, and Jason Polakis. 2019. A Large-scale Study on the Risks of the HTML5 WebAPI for Mobile Sensor-based Attacks. In *30th International World Wide Web Conference, WWW '19*. ACM.

[42] Mozilla and individual contributors. 2021. *Mozilla Developer Network — HTML elements reference*. Retrieved 2021-03-17 from https://developer.mozilla.org/en-US/docs/Web/HTML/Element

[43] Mozilla and individual contributors. 2021. *Mozilla Developer Network — <img>: The Image Embed element — loading attribute*. Retrieved 2021-03-17 from https://developer.mozilla.org/en-US/docs/Web/HTML/Element/img#attr-loading

[44] Mozilla and individual contributors. 2021. *webRequest.onBeforeRequest — Additional objects*. Retrieved 2021-05-18 from https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest/onBeforeRequest#additional_objects

[45] Tim Neutkens, Naoyuki Kanezawa, Guillermo Rauch, Arunoda Susiripala, Tony Kovanen, Dan Zajdband, and contributors. 2021. *Next.js Homepage*. Retrieved 2021-04-26 from https://nextjs.org/

[46] OpenDNS. 2021. *OpenDNS — Domain tagging*. Retrieved 2021-04-30 from https://community.opendns.com/domaintagging/

[47] OpenDNS. 2021. *OpenDNS — Domain tagging — Categories*. Retrieved 2021-04-30 from https://community.opendns.com/domaintagging/categories

[48] Q-Success. 2021. *Usage statistics of JavaScript libraries for websites*. Retrieved 2021-08-11 from https://w3techs.com/technologies/overview/javascript_library

[49] Valentino Rizzo, Stefano Traverso, and Marco Mellia. 2021. Unveiling Web Fingerprinting in the Wild Via Code Mining and Machine Learning. *Proc. Priv. Enhancing Technol.* 2021, 1 (2021), 43–63. https://doi.org/10.2478/popets-2021-0004

[50] Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. 2021. SoK: In Search of Lost Time: A Review of JavaScript Timers in Browsers. In *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*. IEEE, 472–486. https://doi.org/10.1109/EuroSP51992.2021.00039

[51] Jonathan Sampson. 2020. *What's Brave Done For My Privacy Lately? Episode #3: Fingerprint Randomization*. Retrieved 2021-05-27 from https://brave.com/privacy-updates-3/

[52] sanketh. 2020. *mozilla-central — In RFP mode, turn canvas image extraction into a random 'poison pill' for fingerprinters*. Retrieved 2021-05-27 from https://hg.mozilla.org/mozilla-central/rev/ab2a75db3ebe

[53] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. 2021. Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2863–2880. https://www.usenix.org/conference/usenixsecurity21/presentation/shusterman

[54] Alexander Sjösten, Peter Snyder, Antonio Pastor, Panagiotis Papadopoulos, and Benjamin Livshits. 2020. Filter List Generation for Underserved Regions. In *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen (Eds.). ACM / IW3C2, 1682–1692. https://doi.org/10.1145/3366423.3380239

[55] Peter Snyder, Antoine Vastel, and Ben Livshits. 2020. Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced Ad Blocking. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 2 (2020), 26:1–26:24. https://doi.org/10.1145/3392144

[56] Konstantinos Solomos, John Kristoff, Chris Kanich, and Jason Polakis. 2021. Tales of Favicons and Caches: Persistent Tracking in Modern Browsers. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/tales-of-favicons-and-caches-persistent-tracking-in-modern-browsers/

[57] Bootstrap team and contributors. 2021. *Bootstrap Documentation — Accordion.* Retrieved 2021-10-25 from https://getbootstrap.com/docs/5.1/components/accordion/
[58] Bootstrap team and contributors. 2021. *Bootstrap Documentation — Components.* Retrieved 2021-10-25 from https://getbootstrap.com/docs/5.1/getting-started/introduction/#components
[59] Bootstrap team and contributors. 2021. *Bootstrap Documentation — Dropdowns.* Retrieved 2021-10-25 from https://getbootstrap.com/docs/5.1/components/dropdowns/
[60] Bootstrap team and contributors. 2021. *Bootstrap Homepage.* Retrieved 2021-04-25 from https://getbootstrap.com/
[61] Bootstrap team and contributors. 2021. *Bootstrap JavaScript.* Retrieved 2021-04-26 from https://getbootstrap.com/docs/5.1/getting-started/javascript/
[62] Michael Carl Tschantz, Sadia Afroz, Shaarif Sajid, Shoaib Asif Qazi, Mobin Javed, and Vern Paxson. 2018. A Bestiary of Blocking: The Motivations and Modes behind Website Unavailability. In *8th USENIX Workshop on Free and Open Communications on the Internet, FOCI 2018, Baltimore, MD, USA, August 14, 2018*, Lex Gill and Rob Jansen (Eds.). USENIX Association. https://www.usenix.org/conference/foci18/presentation/tschantz
[63] Matteo Varvello and Benjamin Livshits. 2020. On the Battery Consumption of Mobile Browsers. *CoRR* abs/2009.03740 (2020). arXiv:2009.03740 https://arxiv.org/abs/2009.03740
[64] W3C. 1999. *HTML4 Specification — Specifying anchors and links.* Retrieved 2021-03-18 from https://www.w3.org/TR/html401/struct/links.html#h-12.1.3
[65] W3C. 2021. *ARIA Specification — Design Patterns and Widgets.* Retrieved 2021-03-17 from https://w3c.github.io/aria-practices/#aria_ex
[66] WHATWG. 2021. *HTML Specification — Custom data attribute.* Retrieved 2021-03-17 from https://html.spec.whatwg.org/#custom-data-attribute
[67] WHATWG. 2021. *HTML Specification — Implicit submission.* Retrieved 2021-03-18 from https://html.spec.whatwg.org/multipage/form-control-infrastructure.html#implicit-submission
[68] WHATWG. 2021. *HTML Specification — Scroll to the fragment identifier.* Retrieved 2021-03-18 from https://html.spec.whatwg.org/multipage/browsing-the-web.html#scroll-to-the-fragment-identifier
[69] Zhiju Yang and Chuan Yue. 2020. A Comparative Measurement Study of Web Tracking on Mobile and Desktop Environments. *Proc. Priv. Enhancing Technol.* 2020, 2 (2020), 24–44. https://doi.org/10.2478/popets-2020-0016
[70] Foundation Yetinauts. 2021. *Foundation Documentation.* Retrieved 2021-03-17 from https://get.foundation/sites/docs/
[71] Evan You and contributors. 2021. *Vue.js Website.* Retrieved 2021-04-26 from https://vuejs.org/
[72] Shitong Zhu, Xunchao Hu, Zhiyun Qian, Zubair Shafiq, and Heng Yin. 2018. Measuring and Disrupting Anti-Adblockers Using Differential Execution Analysis. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018.* The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_04A-2_Zhu_paper.pdf

APPENDIX

Table 4. JavaScript reliance of standard HTML elements [42].
(0) does not require JavaScript based on the standard, and is very unlikely to require JavaScript in the wild,
(1) does not require JavaScript based on the standard, but sometimes legitimately uses JavaScript in the wild, for additional features,
(1†) does not require JavaScript based on the standard, but sometimes uses JavaScript in the wild, in a non-semantic manner,
(2) does not require JavaScript based on the standard, but requires JavaScript in some use cases,
(2*) does not require JavaScript based on the standard, but often requires JavaScript when used outside a form,
(3) always requires JavaScript based on the standard.

Element	JavaScript reliance	JavaScript use cases
Main root		
`<html>`	(0)	
Document metadata		
`<base>`	(0)	
`<head>`	(0)	
`<link>`	(1)	Asynchronous loading
`<meta>`	(0)	
`<style>`	(0)	
`<title>`	(0)	
Sectioning root		
`<body>`	(0)	
Content sectioning		
`<address>`	(0)	
`<article>`	(0)	
`<aside>`	(0)	
`<footer>`	(0)	
`<header>`	(0)	
`<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>`, `<h6>`	(0)	
`<main>`	(0)	
`<nav>`	(0)	
`<section>`	(0)	
Text content		
`<blockquote>`	(0)	
`<dd>`	(0)	
`<div>`	(1†)	Non-semantic button
`<dl>`	(0)	
`<dt>`	(0)	
`<figcaption>`	(0)	

Element	JavaScript reliance	JavaScript use cases
<figure>	(0)	
<hr>	(0)	
	(0)	
	(0)	
<p>	(0)	
<pre>	(0)	
	(0)	
Table content		
<caption>	(0)	
<col>	(0)	
<colgroup>	(0)	
<table>	(0)	
<tbody>	(0)	
<td>	(0)	
<tfoot>	(0)	
<th>	(0)	
<thead>	(0)	
<tr>	(0)	
Demarcating edits		
<del>	(0)	
<ins>	(0)	
Inline text semantics		
<a>	(1†)	Non-semantic button
<abbr>	(0)	
<b>	(0)	
<bdi>	(0)	
<bdo>	(0)	
 	(0)	
<cite>	(0)	
<code>	(0)	
<data>	(0)	
<dfn>	(0)	
	(0)	
<i>	(0)	
<kbd>	(0)	
<mark>	(0)	
<q>	(0)	
<rp>	(0)	
<rt>	(0)	
<ruby>	(0)	
<s>	(0)	
<samp>	(0)	

Element	JavaScript reliance	JavaScript use cases
`<small>`	(0)	
`<span>`	(1†)	Non-semantic button
`<strong>`	(0)	
`<sub>`	(0)	
`<sup>`	(0)	
`<time>`	(0)	
`<u>`	(0)	
`<var>`	(0)	
`<wbr>`	(0)	
Image and multimedia		
`<area>`	(0)	
`<audio>`	(1)	Custom controls
`<img>`	(1)	Lazy-loading
`<map>`	(0)	
`<track>`	(0)	
`<video>`	(1)	Custom controls
Embedded content		
`<embed>`	(0)	
`<iframe>`	(0)	
`<object>`	(0)	
`<param>`	(0)	
`<picture>`	(0)	
`<portal>`	(0)	
`<source>`	(1)	Lazy-loading
SVG and MathML		
`<svg>`	(1)	Embedded JavaScript
`<math>`	(0)	
Scripting		
`<canvas>`	(3)	
`<noscript>`	(0)	
`<script>`	(2)	May embed a script or a data block
Forms		
`<button>`	(2*)	
`<datalist>`	(2*)	
`<fieldset>`	(0)	
`<form>`	(2)	Requires JavaScript when the form or its values cannot be submitted
`<input>`	(2*)	
`<label>`	(0)	
`<legend>`	(0)	
`<meter>`	(2*)	

Element	JavaScript reliance	JavaScript use cases
`<optgroup>`	(2*)	
`<option>`	(2*)	
`<progress>`	(2*)	
`<select>`	(2*)	
`<textarea>`	(2*)	

Table 5. JavaScript reliance of common UI framework components, based on their documentations and manual testing.
(0) does not require JavaScript,
(0*) does not require JavaScript in the documentation, but is likely to be used with JavaScript in the wild,
(1) does not require JavaScript to be displayed but requires JavaScript to be dismissed, or is used to display transient state, mainly useful with JavaScript,
(1*) does not require JavaScript based on the standard, but requires JavaScript in some use cases (mostly when used outside a form),
(2) does not require JavaScript to be displayed, but requires JavaScript for interactive behavior,
(3) requires JavaScript and displays nothing otherwise.

Component	Bootstrap 5 [58]	Foundation 6 [70]	Tailwind Elements [2]	Semantic UI [1]
Accordion	(2)	(2)	(2)	(2)
Advertisement	N/A	N/A	N/A	(0)
Alerts/Message/Callout	(1)	(1*)	(1)	(1)
Badge(s)	(0)	(0)	(0)/(1)	N/A
Breadcrumb(s)	(0)	(0)	(0)	(0)
Button(s)	(1*)	N/A	(1*)	(1*)
Button group	(0)	N/A	(0)	(0)
Card(s)	(0)	(0)	(0)	(0)
Carousel/Orbit	(2)	(2)	(0*)	N/A
Charts	N/A	N/A	(3)	N/A
Chips	N/A	N/A	(1)	N/A
Checkbox/Checks	(1*)	N/A	N/A	(1*)
Close button	(2)	(2)	N/A	N/A
Collapse	(2)	N/A	N/A	N/A
Comment	N/A	N/A	N/A	(1*)
Container	N/A	N/A	N/A	(0)
Datepicker	N/A	N/A	(3)	N/A
Dimmer	N/A	N/A	N/A	(1)
Divider	N/A	N/A	N/A	(0)
Drilldow menu	N/A	(2)	N/A	N/A
Dropdown(s)	(2)	(2)	(2)	(2)
Embed	N/A	N/A	N/A	(0)
Feed	N/A	N/A	N/A	(0)
File input	N/A	N/A	(0)	N/A
Flag	N/A	N/A	N/A	(0)

Component	Bootstrap 5	Foundation 6	Tailwind Elements	Semantic UI
Floating labels	(0)	N/A	N/A	N/A
Footer	N/A	N/A	(0)	N/A
Form validation	(0)/(2)	N/A	(0)	(3)
Input group, Layout/-Form(s)	N/A	(0)	(1*)	(1*)
Gallery	N/A	N/A	(0)	N/A
Grid	N/A	N/A	N/A	(0)
Headings/Header	N/A	N/A	(0)	(0)
Image(s)	N/A	N/A	(0)	(0)
Icon	N/A	N/A	N/A	(0)
Item	N/A	N/A	N/A	(0)
Form controls/Input(s)	N/A	N/A	(1*)	(1*)
Label	N/A	(0)	N/A	(0)
List group/List	(0)	N/A	(0)	(0)
Menu	N/A	(0)	N/A	(0)
Media	N/A	(0)	N/A	N/A
Modal/Reveal	(2)	(2)	(2)	(2)
Multiselect	N/A	N/A	(1*)	N/A
Navs & tabs/Tab(s)/Pills	(2)	(2)	(2)	(2)
Navbar/Topbar	(0)	(0)	(0)	N/A
Offcanvas/Sidebar	(2)	(2)	N/A	(2)
Pagination	(0)	(0)	(0)	N/A
Placeholder(s)	(0)	N/A	N/A	(1)
Popover(s)/Popup	(2)	N/A	(2)	(2)
Progress/Progress Bar	(1)	(1)	(1)	(3)
Radios	(1*)	N/A	(1*)	N/A
Rail	N/A	N/A	N/A	(0)
Rating	N/A	N/A	(0)	(3)
Range/Slider	(1*)	(2)	(2)	N/A
Responsive Accordion Tabs	N/A	(2)	N/A	N/A
Responsive Embed	N/A	(0)	N/A	N/A
Responsive Navigation	N/A	(2)	N/A	N/A
Reveal	N/A	N/A	N/A	(0)
Scrollspy/Magellan	(2)	(2)	N/A	N/A
Search(s)	N/A	N/A	(1*)	N/A
Select	(1*)	N/A	(1*)	N/A
Segment	N/A	N/A	N/A	(0)/(1)
Shape	N/A	N/A	N/A	(3)
Sidenav	N/A	N/A	(0)	N/A
Spinners/Loader	(1)	N/A	(1)	(1)
Statistic	N/A	N/A	N/A	(0)
Sticky	N/A	N/A	N/A	(2)
Switch	N/A	(1*)	(1*)/(2)	N/A
Stepper/Step	N/A	N/A	(0)	(0)

Component	Bootstrap 5	Foundation 6	Tailwind Elements	Semantic UI
Table(s)	N/A	(0)	(0)	(0)
Thumbnail	N/A	(0)	N/A	N/A
Textarea	N/A	N/A	(1*)	N/A
Timepicker	N/A	N/A	(1*)	N/A
Toast(s)	(1)	N/A	(1)	N/A
Tooltip(s)	(2)	(2)	(2)	N/A

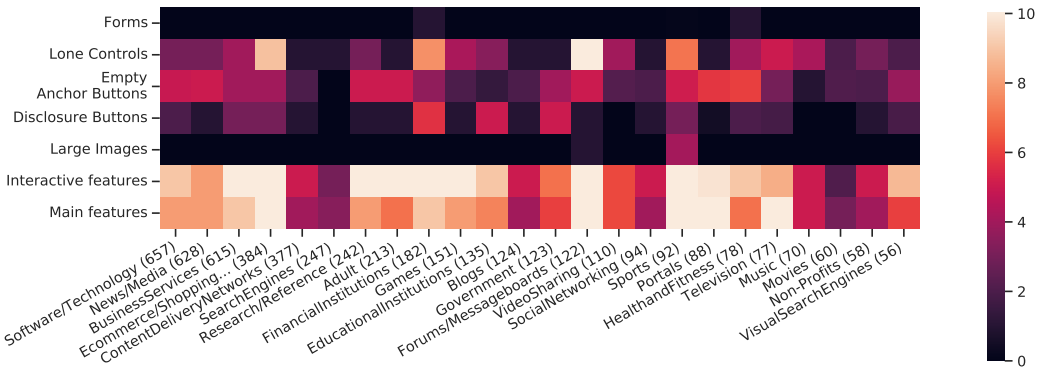

Fig. 9. Color represents the 90th percentile of differential breakage of visible elements in the *whole page*. Lighter shades denote higher differential breakage. Only categories with more than 50 pages in the dataset are plotted, to improve readability. This highlights the disparity of breakage across website categories.