JSRehab: Weaning Common Web Interface Components from JavaScript Addiction

Romain Fouquet, Pierre Laperdrix, Romain Rouvoy

JSRehab: Weaning Common Web Interface Components from JavaScript Addiction

Romain Fouquet
Inria / Univ. Lille
Lille, France

Pierre Laperdrix
CNRS / Univ. Lille / Inria
Lille, France

Romain Rouvoy
Univ. Lille / Inria
Lille, France

ABSTRACT

Leveraging JavaScript (JS) for User Interface (UI) interactivity has been the norm on the web for many years. Yet, using JS increases bandwidth and battery consumption as scripts need to be downloaded and processed by the browser. Plus, client-side JS may expose visitors to security vulnerabilities such as Cross-Site Scripting (XSS).

This paper introduces a new server-side plugin, called JSRehab, that automatically rewrites common web interface components by alternatives that do not require any JavaScript (JS). The main objective of JSRehab is to drastically reduce—and ultimately remove—the inclusion of JS in a web page to improve its responsiveness and consume less resources. We report on our implementation of JSRehab for Bootstrap, the most popular UI framework by far, and evaluate it on a corpus of 100 webpages. We show through manual validation that it is indeed possible to lower the dependencies of pages on JS while keeping intact its interactivity and accessibility. We observe that JSRehab brings energy savings of at least 5 % for the majority of web pages on the tested devices, while introducing a median on-the-wire overhead of only 5 % to the HTML payload.

CCS CONCEPTS

• **Information systems** → **World Wide Web**; • **Human-centered computing** → *Accessibility design and evaluation methods*; • **Security and privacy** → *Web application security*.

KEYWORDS

web security, web framework, javascript, content security policy, web accessibility, mobile web, energy savings

ACM Reference Format:
Romain Fouquet, Pierre Laperdrix, and Romain Rouvoy. 2022. JSRehab: Weaning Common Web Interface Components from JavaScript Addiction. In *Companion Proceedings of the Web Conference 2022 (WWW '22 Companion), April 25–29, 2022, Virtual Event, Lyon, France.* ACM, New York, NY, USA, 6 pages.

1 INTRODUCTION

Since its introduction in 1995, JavaScript (JS) has been widely adopted by websites and is now prevalent on the web. From web pages, embedding a few event listeners, to complex Single Page Applications (SPAs), whose interface entirely rely on JS, almost every website uses JS in one way or another [14].

Existing tools and previous works can reduce the amount of JS shipped to clients, *e.g.*, by removing useless code. In particular, major bundlers—such as Webpack [8] and rollup.js [7]—implement tree-shaking techniques to remove dead JS code by analyzing the import graph. Such dead code is widespread as it is common to include large third-party JS libraries and to only use some of the features they provide. Chaqfeh *et al.* have investigated the classification of scripts as *essential* or *non-essential* as part of a rewriting proxy, with the aim of removing non-essential JS to improve performance on clients, especially on mobile devices [9]. However, automatically replacing parts of JS implementations by noscript alternatives remains an unexplored strategy.

In this paper, we show that most User Interface (UI) components provided by popular frameworks, such as Bootstrap [20], can be automatically replaced by noscript alternatives. In particular, we introduce an automated HTML rewriting technique, named JSRehab, to replace these JS components by their noscript alternatives, hence reducing the dependency on client-side JS, even removing it in some cases. This contribution facilitates the deployment of stricter Content Security Policies (CSPs), enabling to entirely forbid client-side scripting if the page makes no other use of JS, hence dramatically improving client-side security. Reducing the amount of client-side JS can also bring performance improvements and energy savings, by optimizing the amount of data transferred and of scripts processed by the browser, which can be significant on low-end mobile devices [9]. The contributions covered by this paper include:

(1) introducing a stateful component abstraction to implement noscript alternatives,
(2) reporting on the implementation of a noscript alternative generator, currently targeting the most popular UI framework, Bootstrap,
(3) evaluating the payload overhead of these noscript alternatives,
(4) measuring the energy savings on mobile devices, and
(5) manually validating these noscript alternatives on a corpus of 100 webpages.

2 REWRITING HTML PAGES WITH NOSCRIPT ALTERNATIVES

This section introduces the principles underlying noscript alternatives and their automated generation using JSRehab.

2.1 Introducing Noscript Alternatives

We define a noscript alternative as a web structure that implements an interactive behavior equivalent to the JS component it replaces. This structure may combine HTML and CSS constructs to implement the expected interactivity with no single line of JS executed on the page. It should be noted that, despite the naming similarity, noscript alternatives are not necessarily embedded as `<noscript>` tags—which are only interpreted by the browser when JS is disabled—but can be set up to be rendered by any user.

WWW '22 Companion, April 25–29, 2022, Virtual Event, Lyon, France
2022.

When rewriting a UI component as a noscript alternative, one needs to store and to update the component's *state* by only leveraging HTML and CSS constructs. For interactive components, this state can encode for example an opened/closed menu, clicked/focused button or checked/unchecked checkbox. Without this state, the page cannot react to user interactions, as no component will record the changes triggered by the associated events.

To deal with this challenge, we leverage the *checkbox hack* [10, 15], making it possible to record any component state by hiding a checkbox underneath. Interestingly, checkboxes can be natively toggled as checked or unchecked and, even if they are invisible, users can indirectly update them, offering a perfect candidate for implementing our noscript alternatives. Listing 1 illustrates such an example of a checkbox hack implementing a "dropdown button" label that is visible to the user, while the `#chkbox0` element is kept hidden. The menu is not visible as its current style is set to `display:none` but, as soon as the user clicks on the label, the state of `#chkbox0` is toggled to `checked` and the menu becomes visible with `display:block`.

Listing 1: Noscript alternative of a dropdown button

```
<div class="dropdown">
    <!-- dropdown state -->
    <input id="chkbox0" type="checkbox"
      style="position:fixed;opacity:0">
    <!-- dropdown label -->
    <label for="chkbox0">Dropdown button</label>
    <!-- dropdown menu -->
    <ul class="dropdown-menu">
        <li><a href="/item0">Item #0</a></li>
        <li><a href="/item1">Item #1</a></li>
    </ul>
</div>
<style>
    .dropdown-menu { display: none; }
    .dropdown #chkbox0:checked ~ .dropdown-menu {
        display: block;
    }
</style>
```

More generally, implementing noscript alternatives requires to identify: (a) HTML elements that are stateful and that the user can interact with, and (b) CSS selectors that can access the state of these elements. By combining both, one can implement UI components without JS. We studied the *CSS Selectors* specification [22] and derived the set of pseudo-classes that can be used to access the state of HTML elements and other mechanisms without JS. Table 1 reports on the mechanisms that we leveraged in JSRehab: *(1) checkboxes* to record boolean states, *(2) radio buttons* to store mutually exclusive boolean states, *(3) target links* to help page navigation, and *(4) hover/focus* to notify a component of page-level user interactions.

2.2 Rewriting UI Components with JSRehab

2.2.1 Designing noscript alternatives. The design of a noscript alternative for any UI component requires to analyze: *(1)* the UI framework's documentation and source code to identify the purpose and behavior of the component, *(2)* the best strategy in Table 1 to store the component's state. While this approach can be applied to any UI framework, inferring the exact transformation cannot be automated: each framework includes specificities, which may be encoded in a very specific way with a different architecture and corner-cases. This requires each transformation to be manually crafted in order to make sure that everything is appropriately tailored for the targeted UI framework.

Table 1: CSS selectors & elements/mechanisms whose state can be accessed

CSS selector	HTML element/mechanism
`:checked`	`<input type="checkbox">`
`:checked`	`<input type="radio">`
`:target`	Current document's URL fragment
`:focus/:focus-within`	Document focus
`:hover`	Cursor position

2.2.2 Generating noscript alternatives. Even though the *checkbox hack* has been well-known for more than a decade [15], it is not widely used on the web. This can be explained by several factors. Firstly, the implementation of noscript alternatives cannot be factored out and requires to be repeated for each component instance, making it a relatively verbose and error-prone solution when handwriting the required HTML and CSS. Moreover, this first point particularly stands out when compared to interface component frameworks—such as Bootstrap [20] or Foundation [26]—which only require the web developer to add a few classes and attributes to the UI components to enable JS behaviors. Secondly, some noscript alternatives may suffer from corner-cases and unexpected behaviors, making their implementation subtle, which is worsened by the first point, as their implementations cannot be factored out.

However, we believe that the above limitations can be addressed by leveraging an HTML preprocessor, which makes it possible to factor out the noscript implementations as a transform function, thus providing polished and accessible noscript alternatives. To the best of our knowledge, this work is the first to propose using HTML rewriting rules to automatically generate noscript alternatives to common web interface components. Using the technique detailed in subsection 2.2, we succeeded to implement noscript alternatives for almost all Bootstrap components: the list of Bootstrap components and associated noscript mechanisms leveraged to replace them can be found in Table 2 and the JSRehab plugin repository[1] contains detailed documentation about each component.

We opted to use an HTML preprocessor called PostHTML [1] and create our own JSRehab plugin to carry out the transformation. By using existing transformation tooling, we also benefit from its integration into the web ecosystem, including bundlers, such as Webpack [5] and rollup.js [2], and web server frameworks, such as Express [17]. We can also generate noscript alternatives in both Static Site Generation (SSG) and Server-Side Rendering (SSR) contexts, by injecting them only once in the former or whenever the page is rendered in the latter case.

[1] https://gitlab.inria.fr/jsrehab

Table 2: Bootstrap components having a built-in JavaScript behavior [19];

Bootstrap component (latest version)	noscript alter-native	Noscript mechanism(s) used
Accordion (5)	Yes	`<input type="radio">`
Affix (3)	Yes	`position: sticky`
Alerts (5)	Yes	`<input type="checkbox">`
Carousel (5)	Yes	`<input type="radio">`
Collapse (5)	Yes	`<input type="checkbox">`
Dropdowns (5)	Yes	`<input type="checkbox">`
Modal (5)	Yes	`<input type="checkbox">`
Navs & tabs (5)	Yes	`<input type="radio">/:target`
Offcanvas (5)	Yes	`<input type="checkbox">`
Popovers (5)	Yes	`<input type="checkbox">`
Scrollspy (5)	No	*no access to viewport in CSS*
Toasts (5)	Yes	`<input type="checkbox">`
Tooltips (5)	Yes	`:hover/:focus`
Typeahead (2)	No	*cannot replicate autocompletion*

2.3 About Accessibility Challenges

Web accessibility is the practice of ensuring that there are no direct barriers to interact with a website for people with specific disabilities. In the case of the JSRehab plugin, we have to ensure that our noscript alternatives are not making the web harder to browse, by providing at least as good accessibility than the replaced frameworks, and to comply with legal requirements. Most countries having laws mandating accessibility for certain websites rely on the WCAG [24] which do not specify implementation details, only high-level requirements, such as the Success Criterion 2.1.1 Keyboard [25], indicating only that the page must be operable with a keyboard with no time-sensitive input.

As browsers already implement accessibility for standard HTML elements—*e.g.*, spacebar toggles checkboxes, and their change of state is properly announced by screen readers—noscript alternatives are accessible by default, making redundant WAI-ARIA state attributes [21]—such as `aria-checked`—which could not be toggled without JS.

3 VALIDATION METHODOLOGY

This section covers the methodology we applied to validate the noscript alternatives generated by JSRehab for Bootstrap components.

3.1 Validation Corpus Selection

As we focused on Bootstrap, we built a corpus of web pages that use this UI framework, so that we can (1) obtain detailed statistics about its usage, and (2) test and validate JSRehab on them. We crawled the Top 10 k domains from the Tranco list [13] with the following strategy: from the landing page, up to three URLs were extracted as a random sample of `<a>` tags `href` URLs sharing the same origin as the document's URL, but with a different path. Thus, our measurements were collected from up to four pages per domain: the landing page and up to three internal pages. For each web page, we detect the use of Bootstrap through a combination of (1) detecting the version number exposed in the global object, (2) parsing the source code to get the version number from a banner comment, and (3) using custom heuristics when other methods cannot work. We observed that Bootstrap's JS is used on 20.7 % of crawled pages and that its adoption is uniformly distributed across websites ranking, except for the very best ranked websites.

We also measured the popularity of Bootstrap's components by parsing the page's HTML and we conclude that the collapse, dropdown, and modal components were the most common, by far, being found on 53 % of pages using Bootstrap, 40 %, and 31 %, respectively, while all other components are found on less than 10 % of pages using Bootstrap.

Among the 21,341 pages we crawled, 3,291 pages were using Bootstrap's JS and included at least one component in the crawled page. Among these, 1,372 pages were using Bootstrap 4 or 5. Validation is split into two parts: measuring rewriting statistics over the whole sample of 1,372 pages, and empirically evaluating the interactivity and accessibility of noscript alternatives on desktop and mobile devices on a sample of 100 pages.

3.2 Validation Setup

Since it would not have been possible to deploy our solution on production web servers or as a static site generator for testing, we rather chose to implement an HTTP rewriting proxy, which applies the HTML transformation on the fly, whenever a page is requested, see Figure 1.

3.2.1 Rewriting Statistics. To collect statistics about all web pages from the validation set, the list of 1,372 pages is passed to cURL configured to use the HTTP proxy and the Firefox user-agent header, as some websites reject requests with no user-agent.

The HTTP proxy saves the transformation duration and the original and transformed sizes of the compressed HTTP response body, using the same compression method as used by the website (gzip or brotli).

3.2.2 Empirical Validation. The empirical validation is achieved by visiting the same URL three times: once in a control browser on desktop and with JS enabled, a second time in a browser with JS disabled by default and connected to the HTTP proxy, and a third time with a mobile browser, as shown in Table 3. Two viewport widths are tested as webpages often include a hamburger menu that only appears on mobile, leveraging CSS media queries for responsive design.

We chose to disable JS by default in the second browser, so that the original Bootstrap's JS and other custom JS added on the page for component interactivity do not interfere with the noscript alternatives; we only temporarily enable JS to be able to access some hidden components on the page. This, however, makes it impossible to compare the page load time or the time-to-interactive between the pages with and without noscript alternatives, as it could not be isolated from the mere JS blocking.

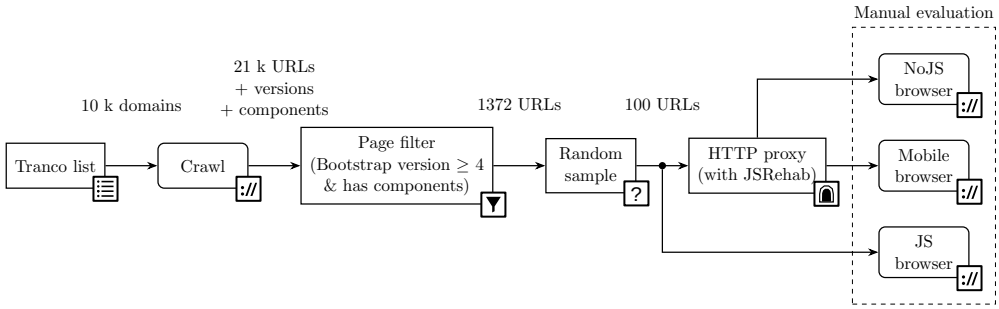

Figure 1: Dataflow diagram of our empirical evaluation setup

Table 3: Browser configurations used for our empirical evaluation

Browser	Device	Window width (px)	Input device	Screenreader	JS
Firefox 93.0	Desktop (Debian)	1280	Keyboard	No	Yes / No
Firefox 93.0	Desktop (Debian)	720 (responsive mode)	Keyboard	No	Yes / No
Firefox 92.0.1	Mobile (Android)	1440	Touchscreen	Yes (TalkBack 2021-04)	No

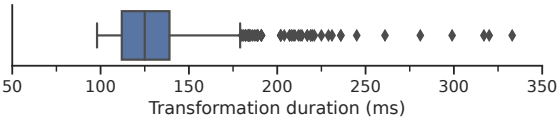

Figure 2: Distribution of transformation durations on the corpus of tested pages

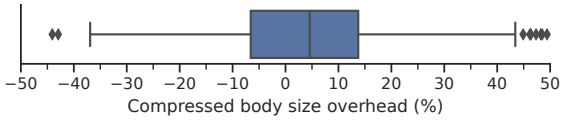

Figure 3: Distribution of the compressed body size overhead on the set of tested pages (some outliers are omitted to improve readability)

3.3 Compatibility Validation

After loading the web page in the browser, we manually validated the interactivity and accessibility of the noscript alternatives by checking the following features:

- on desktop devices:
  - noscript alternatives can be activated with a pointing device,
  - tab-controlled focus behave properly, and
  - noscript alternatives can be activated with spacebar/arrow keys.
- on mobile devices:
  - noscript alternatives can be activated with a touch device,
  - noscript alternatives can be focused using screenreader navigation,
  - noscript alternatives can be activated with screenreader navigation, and
  - screenreader speech announces noscript alternatives appropriately (providing understandable navigation).

For each web page, our testing protocol is as follows:

(1) we verify that the original components on the page are working with JS, but are unresponsive without it,
(2) we validate the aforementioned criteria on a modified page on both desktop and mobile devices with all the noscript alternatives included.

To ease the validation, the HTTP proxy highlights the noscript alternatives so that they are easier to locate and test. For components hidden by default, especially modals, which could not be shown as JS is disabled, some additional effort is made on desktop to make them appear, so that the noscript alternatives can be assessed.

Finally, only Bootstrap components—originally using Bootstrap's JS—are validated, while other components of the page are left untested.

4 EMPIRICAL RESULTS

This section reports on the results we obtained by evaluating JSRehab plugin on the validation corpus.

4.1 Rewriting Statistics

As noscript alternatives are injected in the HTML document, they increase its size, the difference depending on the type and the number of components included in the page. However, as depicted in Figure 3, the resulting overhead on the *compressed* body of the HTTP response containing the HTML document is extremely low, with a median overhead of 5 %; the overhead is lower than 15 % for more than 75 % of tested pages.

The distribution of the rewriting delay, measured by the HTTP proxy running on a high-end laptop for testing, can be found in Figure 2. The median rewriting delay is lower than 125 ms on the sample of tested pages. The rewriting delay mostly depends on the number of HTML nodes in the page, as generating the noscript alternatives requires multiple tree traversals.

4.2 Manual Validation

Among the 100 pages manually analyzed, 79 pages were testable, and all noscript alternatives were working in compliance with the criteria defined in the testing methodology, 19 pages had various issues preventing complete testing, and issues effectively due to noscript alternatives were only found on 2 pages, as detailed in Table 4.

Pages not being fully testable include error pages, which were not the intended pages, SPAs leaving a blank page when JS is disabled, pages with components dynamically added, which thus cannot be detected when processing the HTML document, and pages that were updated between the initial crawl and the validation.

The two pages presenting issues included unconventionally used Bootstrap components. One of them used collapse components with `data-parent` attributes on several different buttons of the web interface. This attribute is intended to be used to build accordions [18], which are supported by the JSRehab plugin; however, this page uses them to make the page menus mutually exclusive so that at most one is open at any time. The other page only partially implements the markup to make footer section headers collapse buttons on mobile, the JSRehab plugin produces collapse buttons for these headers that are also enabled on desktop.

For all other tested pages, the JSRehab plugin produced effective noscript alternatives to original components.

4.3 Preliminary Measurements of Consumption

To complement the rewriting statistics, we measured the energy consumption on mobile devices of a sample of web pages that use Bootstrap's JS. Using Android's built-in utility `dumpsys batterystats`, which can query the device consumption on a per-app basis, we compared the consumption of Chrome loading the unmodified page with a modified version where Bootstrap's JS is blocked and JSRehab is used to preserve page functionality. We focused on Chrome as it is by far the most popular browser on Android [16].

We use a rewriting proxy similar to the one presented above, serving for each requested page: *(1)* the original version with JS and *(2)* the version rewritten with JSRehab and blocking Bootstrap's JS, while still allowing other JS to load and execute. As isolating and blocking Bootstrap's JS is not possible on every page, we focused on the top 31 pages that include a file `bootstrap.min.js`, according to PublicWWW [6]. The proxy is configured to prevent all HTTP caching from the browser and injects a `Refresh=5` HTTP header, which forces the page to reload every 5 s. The energy consumption for each version of the requested web page is measured for 180 s, effectively averaging the measurements over 36 page loads, while we made sure that each page was able to load within the 5 s timeframe. The reported measurements have been performed on two low-to-medium-end phones, the Archos 50 Platinum 4G and the Moto Z, respectively running Chrome 50 and Chrome 96.

As depicted in Figure 4, one can observe that replacing Bootstrap's JS with JSRehab enabled significant energy savings on many web pages on the tested devices; websites operators should evaluate on a case-by-case basis the exact benefit on their own website.

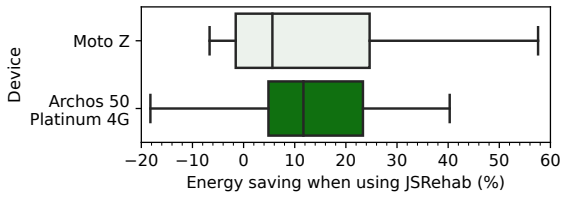

Figure 4: Energy savings of mobile devices when loading web pages without Bootstrap's JS and rewritten with JSRehab; outliers are excluded for readability

5 DISCUSSION

5.1 Expected Benefits

5.1.1 Improving security and privacy. Deploying the JSRehab plugin makes it possible to remove the Bootstrap dependency, which can then be removed from allowed JS sources in the CSP, hence contributing to reducing the attack surface. If the web page makes no other use of JS, the execution of JS can even be forbidden in the page by adopting a strict CSP directive, further mitigating the risk of Cross-Site Scripting (XSS). Website owners may thus be incentivized to further reduce the amount of JS included in their web pages, as one of the key usage of JS directly benefiting the user—component interactivity—has been substituted.

Another strong incentive to use JSRehab is that it protects a website from yet-to-be-discovered vulnerabilities. At the time of development, a developer can integrate the latest available version of a UI library that may be assumed as safe. Then, weeks later, a vulnerability can be discovered, hence requiring the dependency to be updated. With JSRehab, a website is protected as the JS code is simply not there. This problem is widespread as a lot of the crawled websites were using outdated versions of Bootstrap. This lack of security fixes can open users to undesired security problems.

5.1.2 Improving performance. Replacing JS components with their noscript alternatives can also bring performance improvements. Indeed, noscript alternatives adding only a median 5 % overhead, and since Bootstrap's JS is not needed if the page only includes the supported components, the amount of data transferred on the wire is reduced, leading to faster page loads. This point is even more significant as major browsers have implemented HTTP cache partitioning [11, 12], preventing Bootstrap's JS to be reused between websites when the same version was loaded from a CDN.

Moreover, the processing burden on the client is reduced as the browser does not need to parse and execute the additional JS (Bootstrap 5's JS weights 60 kB minified but uncompressed), which can be significant, especially on mobile devices [9] and can extend device lifespan. The time-to-interactive of web pages can also be reduced, thus improving page responsiveness. Typically, such performance improvements are not achieved by JSCleaner [9], as component interactivity scripts would be considered as *essential*.

5.2 Ease of Adoption

When using Bootstrap components as intended, the JSRehab plugin produces effective noscript alternatives with almost no configuration. It only needs to be provided with the stylesheets used by the

Table 4: Summary of our empirical evaluation observations on a sample of 100 pages

Observed behavior	Count
Web page is fully interactive and all noscript alternatives are behaving correctly	79 pages
No component found in the page with JS: the page likely changed between the initial crawl and the validation	6 pages
Error pages on web pages with JS enabled (different from the initial crawl)	4 pages
Buggy pages due to inappropriate usage of original Bootstrap	4 pages
SPAs or components dynamically added by JS	3 pages
Custom component styling that cannot be triggered by noscript alternatives and cannot be manually bypassed for testing	2 pages
Web page is interactive, but some noscript alternatives are misbehaving	2 pages

page, so that it can generate matching styling. As PostHTML is already integrated into various bundlers and web server frameworks, such as Webpack, rollup.js or Express, it is straightforward to adopt JSRehab in an existing project, with no change in tooling; the JSRehab repository contains an example of configuration file for Webpack.

Furthermore, as noscript alternatives are generated for each page separately, it is possible to progressively transition to using the JSRehab plugin and enforcing a stricter CSP.

With a median transformation delay of 125 ms, JSRehab performance is compatible with a Static Site Generation (SSG) setup, where pages are rendered once, then served as part of a static site. Depending on the website expectations, it may currently be too slow for a Server-Side Rendering (SSR) context, where pages are rendered on the web server upon requests. The JSRehab plugin would need to be further optimized for this context or rewritten in a programming language more suited to string processing.

5.3 Beyond Bootstrap

We specifically demonstrated the generation of noscript alternatives for replacing components from the Bootstrap framework, but this technique can be leveraged for other component frameworks as well, be them public or in-house, to factor out noscript alternative implementation and make them easier to use. As many UI frameworks share a large set of components [3, 26], porting the JSRehab plugin would mostly require updating the class names used as component identifiers, which are specific to each framework.

Other types of components, not implemented by Bootstrap, could also be automatically replaced with noscript alternatives, including sortable tables, image lightboxes, and data plots. Other components could be implemented without JS if browsers were to implement CSS pseudo-classes such as `:has()` [23] or `:in-viewport` [4].

6 CONCLUSION

In this paper, we introduced a server-side technique to automatically replace common web interface components implemented by UI frameworks with noscript alternatives. We implemented this technique as a set of HTML rewriting rules that generate noscript alternatives for Bootstrap and we discussed the key benefits and current limitations of our contribution. We also validated these noscript alternatives on a corpus of 100 webpages, and we observed that they deliver convincing alternatives by assessing their interactivity and accessibility from both desktop and mobile devices, while introducing only minimal overhead on the compressed HTML document and that they enable energy savings.

REFERENCES

[1] 2021. *PostHTML.* https://github.com/posthtml/posthtml
[2] 2021. *rollup.js.* https://rollupjs.org/guide/
[3] 2021. *Semantic UI.* https://semantic-ui.com/
[4] 2021. *The Web We Want — I want a CSS pseudo-selector for elements that are in the viewport.* https://webwewant.fyi/wants/63/
[5] 2021. *webpack.* https://webpack.js.org/
[6] 2022. *PublicWWW — "bootstrap.min.js".* Retrieved 2022-01-24 from https://publicwww.com/websites/%22bootstrap.min.js%22/
[7] 2022. *rollup.js — Tree-Shaking.* Retrieved 2022-03-04 from https://rollupjs.org/guide/en/#tree-shaking
[8] 2022. *Webpack — Tree Shaking.* Retrieved 2022-03-04 from https://webpack.js.org/guides/tree-shaking/
[9] 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
[10] Chris Coyier. 2020. *The "Checkbox Hack" (and things you can do with it).* Retrieved 2022-03-04 from https://css-tricks.com/the-checkbox-hack/
[11] Tim Huang Johann Hofmann. 2021. *Mozilla Hacks — Introducing State Partitioning.* Retrieved 2022-03-04 from https://hacks.mozilla.org/2021/02/introducing-state-partitioning/
[12] Eiji Kitamura. 2020. *Google Developers — Gaining security and privacy by partitioning the cache.* Retrieved 2022-03-04 from https://developers.google.com/web/updates/2020/10/http-cache-partitioning
[13] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2021. *Tranco — A Research-Oriented Top Sites Ranking Hardened Against Manipulation.* https://tranco-list.eu/
[14] Q-Success. 2021. *W3Techs — Usage statistics of client-side programming languages for websites.* Retrieved 2021-10-19 from https://w3techs.com/technologies/overview/client_side_language
[15] Ryan Seddon. 2010. *Custom radio and checkbox inputs using CSS.* Retrieved 2022-03-04 from https://ryanseddon.com/css/custom-inputs-using-css/
[16] StatCounter. 2022. *statcounter — Mobile Browser Market Share Worldwide.* Retrieved 2022-01-24 from https://gs.statcounter.com/browser-market-share/mobile/worldwide
[17] StrongLoop, IBM, and other expressjs.com contributors. 2021. *Express - Node.js web application framework.* https://expressjs.com/
[18] Bootstrap team. 2021. *Bootstrap 4 — Collapse, Accordion example.* Retrieved 2021-10-20 from https://getbootstrap.com/docs/4.6/components/collapse/#accordion-example
[19] Bootstrap team. 2021. *Bootstrap 5 — Components.* Retrieved 2021-10-19 from https://getbootstrap.com/docs/5.1/components/
[20] Bootstrap team. 2021. *Bootstrap · The most popular HTML, CSS, and JS library in the world.* https://getbootstrap.com/
[21] W3C. 2021. *Accessible Rich Internet Applications (WAI-ARIA) 1.3 — WAI-ARIA States and Properties.* https://w3c.github.io/aria/#introstates
[22] W3C. 2021. *Selectors Level 4.* https://drafts.csswg.org/selectors-4/#overview
[23] W3C. 2021. *Selectors Level 4.* https://drafts.csswg.org/selectors/#relational
[24] W3C. 2021. *Web Accessibility Laws & Policies.* https://www.w3.org/WAI/policies/
[25] W3C. 2021. *Web Content Accessibility Guidelines (WCAG) 2.1 — Success Criterion 2.1.1 Keyboard.* https://w3c.github.io/wcag21/guidelines/#keyboard
[26] Foundation Yetinauts. 2021. *Foundation — The most advanced responsive front-end framework in the world.* https://foundation.zurb.com/