∴

Improving Web User Privacy Through Content Blocking

Préserver la vie privée en ligne grâce au blocage de contenu

∴

Romain Fouquet

Université de Lille

École Doctorale MADIS
Centre Inria de l'Université de Lille
CRIStAL — Spirals Research Team

Thèse présentée et soutenue le 31 mai 2023
pour obtenir le grade de docteur en informatique

Directeur et co-encadrant

Romain Rouvoy	Professeur	Université de Lille
Pierre Laperdrix	Chargé de recherche	CNRS

Rapporteurs

Olivier Barais	Professeur	Université de Rennes
Nataliia Bielova	Chargée de recherche avec HDR	Inria

Examinateurs

Aurore Fass	Visiting Assistant Professor	Stanford University
Jean-Christophe Routier	Professeur, président du jury	Université de Lille

Résumé

Le Web a connu une croissance constante depuis sa création en 1990. En parallèle de cette production permanente de contenu, le pistage des utilisateurs est apparu rapidement et s'est développé de manière tout aussi soutenue. Les utilisateurs sont alors suivis aussi bien entre les sites Web que dans leur comportement sur chaque page. Pour combattre ce pistage, les développeurs de navigateurs et d'extensions ont proposé différentes stratégies, de l'isolation des sites les uns par rapport aux autres jusqu'à demander aux utilisateurs la permission d'utiliser certaines fonctionnalités sensibles. Dans cette thèse, on s'intéresse à une de ces stratégies de protection côté client : le blocage de contenu. Le blocage de contenu consiste à empêcher le chargement ou l'exécution de certaines parties des pages Web, protégeant ainsi les utilisateurs de celles-ci. Les outils de blocage de contenu existants reposent surtout sur des listes de filtrage qui spécifient les parties des pages à bloquer. Cependant, elles souffrent de nombreux inconvénients : elles peuvent notamment être incomplètes ou incapables de cibler certains types de ressources. On présente quatre contributions pour améliorer la vie privée des internautes en modifiant le contenu des pages :

1. On mesure la dépendance au JavaScript d'éléments courants des pages Web et dans quelle mesure bloquer JavaScript permet d'améliorer la vie privée. On trouve que 43 % des pages Web de notre échantillon ne dépendent pas strictement de JavaScript et que 67 % des pages sont susceptibles d'être utilisables si l'on se préoccupe seulement du contenu principal de la page.
2. En s'appuyant sur les connaissances acquises concernant la casse des pages quand JavaScript est bloqué, on conçoit un ensemble de réparations pour corriger les cas courants de casse. On introduit le concept de User Browsing Intent (UBI) (intention de navigation) et, en se concentrant sur la UBI « read-only », on mesure à quel point ces réparations sont utiles dans le cas de cette UBI.
3. On propose un système côté serveur pour remplacer les composants d'interface dépendants généralement de JavaScript par des versions sans JavaScript, et on évalue les bénéfices de cette substitution, notamment d'un point de vue sécurité et de réduction de la consommation des terminaux.
4. On conçoit un algorithme de signature pour produire des signatures robustes de fonctions JavaScript et détecter le *bundling* (empaquetage) de fonctions de pistage provenant de scripts de pistage connus avec du code bénin. On trouve que 22.7 % des domaines de notre échantillon comprennent de telles fonctions de pistage, qui contournent ainsi, de fait, les outils de blocage

existants. On propose finalement une technique pour bloquer ces fonctions de pistage tout en préservant les fonctionnalités du code environnant.

Abstract

The web has seen steady growth since its inception in 1990. Along with this constant production of content, user tracking has appeared early and seen continuous development. Users are thus followed across websites and their behavior observed on individual web pages. To combat user tracking, browser vendors and extension developers have proposed different strategies, ranging from site isolation to asking the user before using sensitive features. In this thesis, we focus on one of this client-side privacy protection strategy: content blocking. Content blocking consists in preventing unwanted parts of web pages from being downloaded or executed, thus protecting the user from them. Existing content blocking tools mostly rely on filter lists which specify what parts of web pages to block. They however suffer from several issues, including incomplete coverage and being unable to target certain kinds of resources. We present four contributions for improving user privacy by modifying page content:

1. We measure the dependency on common web page elements on JavaScript and how much blocking JavaScript can improve user privacy. We find that 43 % of web pages from our sample do not strictly depend on JavaScript and that 67 % of pages are likely to be usable when caring only about the main page section.
2. Building on the acquired knowledge of page breakage when blocking JavaScript, we design a set of repairs to repair common page breakage types. We introduce the concept of User Browsing Intent (UBI) and, focusing on the 'read-only' UBI, we measure how much these repairs are useful in the case of this UBI.
3. We propose a server-side system to substitute interface page elements usually relying on JavaScript with noscript alternatives, and discuss the benefits of this replacement in particular in terms of device energy savings and security.
4. We devise a signature scheme to generate robust signatures of JavaScript functions, and detect the bundling of tracking functions from known tracking libraries with functional code. We find that 22.7 % of domains in our sample bundle such tracking functions with functional code, effectively circumventing existing blocking tools. We propose a technique for blocking these tracking functions while preserving functional code.

Acknowledgments

I would like to thank everyone who contributed to the making of this thesis.

First, I would like to thank my two supervisors, Romain Rouvoy and Pierre Laperdrix. I want to express my gratitude for giving me the opportunity to do a Ph.D. on a topic I was and still am very interested in. It was a pleasure to share ideas with you and I appreciate all the advice you have given me. Thank you for listening to my lengthy weekly diatribes.

Even though the health situation did not always allow to come to the office, it has been a pleasure to come to the lab when it was possible. For that, I would like to thank all the members from the Spirals team for the stimulating discussions we had. Thank you Pierre Laperdrix and Lionel Seinturier for organizing online and in-person team seminars, so we get to know other team members and what everyone is working on. Thanks to my office mates from office B310 for all the fruitful discussions we had and for tips we shared about experiment design and workflow tools.

Outside the lab, I would also like to thank the contributors to the Rust language and ecosystem for making such a powerful language and nice crates. In particular, thanks to the maintainers and contributors to the crates thirtyfour and SWC, which made it possible to write robust crawlers and JavaScript processing programs and meet tight deadlines. Thank you to all to all the artists I listened to during these three years for fueling my coding and writing sessions.

Finally, I am incredibly thankful to my loved ones for their infallible support throughout my thesis and more generally during my studies.

Contents

Résumé **3**

Abstract **5**

Acknowledgments **7**

List of Figures **13**

List of Tables **15**

I Introduction **17**
I.1 Motivation . . . 17
I.2 Contributions . . . 18
I.2.1 Investigating Page when Disabling JavaScript . . . 18
I.2.2 Bridging the Gap Between the User and the Browser with User Browsing Intent (UBI) . . . 19
I.2.3 Reducing Interface Components Dependency on JavaScript Server-Side . . . 19
I.2.4 Detecting and Blocking Individual JavaScript Functions for Privacy . . . 20
I.3 List of Scientific Publications . . . 20
I.4 List of Tools and Prototypes . . . 20
I.5 Outline . . . 21

II Background and Context **23**
II.1 Tracking on the Web . . . 23
II.1.1 Actors and Motives . . . 23
II.1.2 Tracking Vectors . . . 25
II.2 Web Tracking Deployment . . . 29
II.3 Web Tracking Protections . . . 29
II.3.1 State Isolation . . . 30
II.3.2 Shorter-Lived State . . . 30
II.3.3 Permission System . . . 31
II.3.4 Disabling Browser Features . . . 31
II.3.5 Anti-Fingerprinting Protections . . . 31

II.4 Content Blocking . . . 32
II.4.1 Filter Lists . . . 32
II.4.2 Resource Replacement . . . 36
II.4.3 Dynamic User Rules . . . 36
II.4.4 Auto-Adaptive Blocking . . . 37
II.4.5 Additional Benefits of Content Blocking . . . 39
II.5 Recent Changes in Cornerstone Features of Browsers . . . 39
II.5.1 Request HTTP Headers . . . 40
II.5.2 Cookies and Other Client-Side Storage Types . . . 41
II.6 Conclusion . . . 42

III Investigating Page Breakage when Disabling JavaScript 45
III.1 Detecting Page Feature Breakage . . . 46
III.1.1 Breakage Detection . . . 46
III.1.2 Page Feature Relevance . . . 53
III.2 Data Collection . . . 54
III.2.1 Crawled Websites and Pages . . . 54
III.2.2 Collection of Website Categories . . . 55
III.3 Results . . . 56
III.3.1 Dataset Description . . . 56
III.3.2 Effect on Page Load Time . . . 56
III.3.3 Page Section Classification . . . 57
III.3.4 Page Feature Breakage . . . 57
III.3.5 Website Handling of Non-JavaScript Users . . . 61
III.4 Limitations . . . 64
III.4.1 Measurement Framework Limitations . . . 64
III.4.2 Crawl Limitations . . . 64
III.5 Discussion . . . 65
III.5.1 Manual Visual Analysis of Screenshots . . . 65
III.5.2 JavaScript Reliance of Most Visited Websites and Component Framework Trends . . . 65
III.5.3 Benefits and Viability of Aggressive JavaScript Blocking . . . 66
III.5.4 Website Usage of JavaScript Features With No Fallback . . . 68
III.6 Conclusion . . . 69

IV Bridging the Gap Between the User and the Browser with User Browsing Intent 71
IV.1 Introducing the User Browsing Intent . . . 72
IV.1.1 Current State of Content Blocking Solutions . . . 72
IV.1.2 More Aggressive Blocking with the User Browsing Intent . . . 72
IV.2 Blocking JavaScript and Repairing Specific Breakage Cases . . . 73
IV.2.1 Blocking JavaScript with a WebExtension . . . 73
IV.2.2 Scope of Targeted Repairs . . . 74

IV.2.3 Repairing Breakage Induced by JavaScript Blocking . . . 76
IV.3 Evaluating Repairs . . . 77
IV.3.1 Methodology . . . 77
IV.3.2 Experimental Results . . . 81
IV.4 Discussion and Limitations . . . 85
IV.4.1 Public Pages . . . 85
IV.4.2 Usage and Privacy–Usability Trade-off . . . 85
IV.4.3 Unfixed Breakage and Maintenance . . . 86
IV.5 Conclusion . . . 87

V Reducing Interface Components Dependency on JavaScript Server-Side 89
V.1 Rewriting HTML with Noscript Alternatives . . . 89
V.1.1 Introducing Noscript Alternatives . . . 90
V.1.2 Rewriting UI Components with JSRehab . . . 91
V.1.3 HTML & CSS Limitations . . . 93
V.1.4 About Accessibility Challenges . . . 94
V.2 Validating the Noscript Alternatives . . . 95
V.2.1 Validation Corpus Selection . . . 95
V.2.2 Validation Setup . . . 96
V.2.3 Compatibility Validation . . . 98
V.3 Results . . . 98
V.3.1 Rewriting Statistics . . . 98
V.3.2 Manual Validation . . . 99
V.3.3 Preliminary Measurements of Consumption . . . 99
V.4 Discussion . . . 101
V.4.1 Expected Benefits . . . 101
V.4.2 Ease of Adoption . . . 101
V.4.3 Beyond Bootstrap . . . 103
V.5 Conclusion . . . 103

VI Detecting and Blocking Individual JavaScript Functions for Privacy 105
VI.1 Motivation . . . 106
VI.1.1 Blocking and Usability . . . 106
VI.1.2 Related Work . . . 106
VI.1.3 Bundling and Minification . . . 108
VI.2 Building Function Signatures and Statically Detecting Tracking Functions . . . 112
VI.2.1 Generating Function Signatures . . . 112
VI.2.2 Collecting Scripts . . . 113
VI.2.3 Detecting Bundled Tracking Functions . . . 114
VI.3 Results . . . 115
VI.3.1 Script Statistics . . . 115
VI.3.2 Bundled Tracking Functions . . . 117

VI.4 Blocking Tracking Functions In-Browser . . . 119
VI.4.1 Leveraging Function Signatures for In-Browser Blocking . . . 119
VI.4.2 Function Substitution Strategies . . . 119
VI.5 Limitations and Discussion . . . 121
VI.5.1 Static Analysis and Obfuscation . . . 121
VI.5.2 Third-Party Scripts . . . 122
VI.5.3 Inline Scripts . . . 122
VI.5.4 Bundlers Usage Trend . . . 122
VI.6 Conclusion . . . 122

VII Conclusion **125**
VII.1 Contributions . . . 125
VII.1.1 Investigating Page when Disabling JavaScript . . . 125
VII.1.2 Bridging the Gap Between the User and the Browser with User Browsing Intent (UBI) . . . 126
VII.1.3 Reducing Interface Components Dependency on JavaScript Server-Side . . . 126
VII.1.4 Detecting and Blocking Individual JavaScript Functions for Privacy . . . 126
VII.2 Future Work . . . 127
VII.2.1 Keeping Filter Lists Lean . . . 127
VII.2.2 Increasing Fine-Grained JavaScript Blocking Ability . . . 127
VII.2.3 Assisting Shim Generation . . . 127
VII.2.4 Improving the Privacy–Usability Trade-off . . . 128
VII.3 Insights and Perspectives . . . 128
VII.3.1 Loose Coupling between Page Elements and Behavior . . . 128
VII.3.2 New Resource Types and Tighter Weaving . . . 129
VII.3.3 Non-Alignment of Interests of Producers and Consumers . . . 129

A Appendices to the Breaking Bad Study **131**

B Appendices to the UBI Study **139**

C Appendices to the JSRehab Study **143**

D Appendices to the JsRipper Study **145**
D.0.1 Frequency Analysis in Name Mangling . . . 145

Bibliography **149**

List of Figures

II.1 uBlock Origin pop-up in normal and advanced mode 37
II.2 uMatrix pop-up where only first-pary CSS, images, and iframes are allowed 38
II.3 The NoScript extension allows to selectively block scripts per domain (CC BY-SA 4.0 https://noscript.net/) 38
II.4 History of HTTP cookies and their security and privacy features. Dashed boxes represent the deployment in browsers of isolation of cookies for all users in normal windows. 41

III.1 Lazy loading images 48
III.2 Examples of disclosure buttons (from the Bootstrap 5 documentation [235, 233]) . . 51
III.3 Common page structure (the sidebar can be on either side of the page) 54
III.4 Dataflow diagram 55
III.5 Speedup when blocking JavaScript 56
III.6 Color represents the 90^{th} 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. 59
III.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 . 60
III.8 Normalized differential breakage (DBRn) of visible elements of the *main section*; negative normalized differential breakage is not shown for readability 61

IV.1 Screenshots of the homepage of www.sage.com, having fade-in elements, which make the content unreadable without JavaScript, fixed by our WebExtension 75
IV.2 Screenshots of pages having preloaders, which completely prevent accessing the page content without JavaScript 76
IV.3 Distribution of the pixel difference percentage between the JS and NoJS-UBI screenshots in our 30,728-page sample 78
IV.4 Tukey boxplots of the counts of repaired elements per page, when at least one element has been repaired: when a page contains a lazy-loaded image, it usually contains many of them, unlike preloaders, which usually appear only once on a page when present. 82

IV.5 Bivariate plot of the manual labeling; the grades are defined in subsubsection IV.3.1.3 83
IV.6 Correlation matrix of repairs modifying at least one element 84
IV.7 Tukey boxplots of the load time differences (measured with the 'load' event) when disabling JavaScript and when applying our repairs to the NoJS pages, in our 30,728-page sample 85

V.1 Intended deployment setup of JSRehab: it is meant to be used in an Static Site Generation (SSG) or in an Server-Side Rendering (SSR) context 92
V.2 JSRehab is built as a plugin for the existing PostHTML HTML preprocessor; it processes input HTML, along with optional additional stylesheets, to produce HTML with noscript alternatives 92
V.3 CDF of Bootstrap's JS usage according to website ranking; Bootstrap's JS usage distribution is uniform, except for the very best ranked websites 95
V.4 Popularity of Bootstrap's components; accordion and offcanvas components have only been recently introduced with Bootstrap 5, and are not widely used 96
V.5 Dataflow diagram of our empirical evaluation setup 97
V.6 Distribution of transformation durations on the corpus of tested pages 97
V.7 Distribution of the compressed body size overhead on the set of tested pages (some outliers are omitted to improve readability) 99
V.8 Energy savings of mobile devices when loading web pages without Bootstrap's JS and rewritten with JSRehab; outliers are excluded for readability 100
V.9 Popularity of Bootstrap's JS versions on the crawled pages; versions marked with a star (∗) are known to be vulnerable to at least one XSS vulnerability as listed by [168] 102

VI.1 Minimal example of a bundle produced by Webpack 5 for a CommonJS module . . . 109
VI.2 JavaScript code for the Pythagorean theorem. The three pieces of code are all equivalent. 111
VI.3 File size of external scripts 116
VI.4 Script count per page 116
VI.5 Count of functions per script 116
VI.6 Source code size of functions 117

A.1 Color represents the 90^{th} 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. 138

B.1 Screenshot of the manual labeling tool (truncated vertically for readability), from left to right: the JS screenshot, NoJS screenshot, NoJS-UBI screenshot, and the color-coded difference image. 141

List of Tables

II.1 Deployment of network and complete state partitioning in browsers; PB stands for Private Browsing . . . 30

III.1 Success statistics for each crawl step . . . 56

III.2 Ratios of pages where the following CSS selectors match at least one element . . . 57

III.3 `<noscript>` purposes . . . 62

III.4 Mean request count and standard deviation for each request type with JavaScript enabled and disabled . . . 66

IV.1 Shares of pages among the 30,728 pages where repairs have been applied to at least one element . . . 82

IV.2 Shares of pages compliant with the 'read-only' UBI and the indicated tolerance . . . 83

IV.3 Counts of third-party tracking requests with JavaScript enabled and disabled, with our repairs. Some Script and XHR requests are still present without JavaScript because JSON objects loaded with `<script>` tags and `<link>` preloads are classified as such. . . . 86

V.1 CSS selectors & elements/mechanisms whose state can be accessed . . . 91

V.3 Browser configurations used for our empirical evaluation . . . 97

V.4 Summary of our empirical evaluation observations on a sample of 100 pages . . . 100

VI.1 Regular expressions used to recognize privacy-relevant APIs; `\b` denotes a word boundary . . . 115

VI.2 Tracking scripts whose tracking functions are most often bundled (truncated to top 5) 118

VI.3 Examples of scripts containing tracking functions, these scripts are not part of filter lists . . . 119

A.1 JavaScript reliance of standard HTML elements [185]. (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. 131

A.2 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. 135

D.1 Tracking scripts whose tracking functions are most often bundled 147

D.2 Examples of scripts bundling tracking functions, these scripts are not part of filter lists 148

CHAPTER I

Introduction

I.1 Motivation

Since its inception in 1990, the web has seen steady development and adoption, totaling more than 270,000,000 active websites in 2022 [192]. This impressive growth in size and content has however been accompanied by early and widespread use of user tracking by websites [151, 164]. Indeed, user tracking has been deployed by websites since as early as 1996 [164] for various purposes. User tracking is part of targeted advertising, which aims to show users advertisement relevant to their interest to maximize the chance the users click on them. Such user tracking thus follows users across multiples websites, using third-party trackers, to aggregate data about them and construct a profile, then used to select advertisements to display in advertising space of websites [116, 241, 69]. Websites also track detailed usage behavior of users, as if they were watching over the users' shoulders, using analytics and session replay tools. User tracking is so widespread on the web that Sánchez-Rola and Santos have reported that 90 % of websites perform user tracking in one way or another [212].

Such tracking may have financial consequences for users—e.g., higher insurance rates due to certain health conditions revealed by online tracking [48]—or even more serious fallout if used for government surveillance. User profiling is also leveraged for political advertisement targeting, to show different messages to different groups [174], sometimes very small—a practice in that case known as micro-targeting—endangering democracy. In addition, the sense of lack of privacy online may result in "chilling effects," where users refrain from inquiring about topics they perceive as sensitive, due to concerns of being tracked. Penney has investigated this effect and observed reduced traffic of certain sensitive Wikipedia articles following Snowden's disclosures [200]. Before that, Marthews and Tucker had also found that web searches for certain sensitive keywords had also decreased after these disclosures [173]. More generally, user tracking undermines users' privacy, which is recognized as a fundamental human right by the Universal Declaration of Human Rights [210].

Aggressiveness and prevalence of web tracking call for effective privacy defenses that users could employ to protect themselves when browsing the web. In this context, browser vendors and browser extension developers have deployed numerous privacy enhancing techniques to try

to preserve user privacy [69], while encumbering user's browsing as little as possible, highlighting the trade-off between user privacy and usability. As user tracking relies on a wide array of technologies, multiple privacy defense techniques have been deployed. User tracking can leverage client-side storage or try to build a unique browser fingerprint for re-identification [241]. Defenses thus include browser state isolation, reducing the lifetime of such state, or introducing a permission system for defeating stateful tracking. Stateless tracking protection try to break the uniqueness or stability properties of browser fingerprinting by ensuring attribute uniformity or randomizing these attributes. However, all these defenses are still not sufficient to protect against all forms of user tracking, thus calling for another strategy that we explore in depth in this thesis: content blocking, which consists in blocking unwanted parts of webpages. One of the most common strategy for content blocking is to rely on filter lists, detailing patterns of URLs to block. While being simple and useful, filter lists suffer from numerous issues, including incomplete coverage [222, 218] and being unable of blocking only parts of scripts [73]. In this thesis, we investigate other strategies to provide stronger privacy guarantees and remedy these issues. In the course of this exploration, we address the following research questions:

- How much are web page elements dependent on JavaScript?
- How useful is it to browse the web with JavaScript disabled from a privacy point of view?
- How viable is it to browse with JavaScript disabled?
- Can we repair common page breakage types when JavaScript is disabled and how useful can it be for making browsing without JavaScript viable?
- How can we automatically replace interactive elements usually relying on JavaScript with noscript versions?
- How often are tracking functions bundled with functional code, making it impossible to block these mixed scripts?
- Can we remove these tracking functions from bundles, while preserving functional code?

While defending against web tracking is a continual cat and mouse game, we hope that the following chapters will bring satisfying new strategies to combat tracking, along with presenting the foundations of online tracking and privacy protections.

I.2 Contributions

I.2.1 Investigating Page when Disabling JavaScript

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 contribution, 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 breakage. As there is no standard

mechanism for detecting such breakage, 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.

I.2.2 Bridging the Gap Between the User and the Browser with User Browsing Intent (UBI)

Content blockers are popular browser extensions to preserve the user privacy and security by blocking parts of web pages. However, they usually rely on filter lists, which may be incomplete or lagging behind. Acknowledging that users may visit a web page with different intents, we introduce the concept of User Browsing Intent (UBI), allowing us to apply more aggressive blocking strategies. We focus on the simplest form of UBI that is devoid of interaction with the page: the 'read-only' UBI—where the user only wants to read the page, and not interact with it. This allows us to provide a tool blocking JavaScript by default, while providing a set of hand-crafted repairs, aimed at automatically repairing common types of breakage induced by JavaScript blocking and fixable client-side. When evaluating this tool on a sample of 30,728 pages using semi-manual sampling and labeling, we find that (a) more than 62 % of pages are compliant with the 'read-only' UBI if the user only tolerates minor information loss, and that (b) more than 77 % of pages are compliant if the user also tolerates the loss of some non-central sections. Our in-browser repairs make more than 27 % more pages compliant with the 'read-only' UBI. Reducing the mean number of tracking requests by more than 97.7 %, our tool brings significant privacy and security improvements by recognizing that the user does not always require the whole page. If the page does not comply with the 'read-only' UBI, even with our repairs, or if the user's UBI is different, it is easy to enable back JavaScript on the page.

I.2.3 Reducing Interface Components Dependency on JavaScript Server-Side

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 contribution introduces a new server-side plugin, called JSRehab, that automatically rewrites common web interface components with alternatives that do not require any JavaScript. 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.

I.2.4 Detecting and Blocking Individual JavaScript Functions for Privacy

Content blockers aim to improve web privacy and security by blocking parts of web pages. However, they are limited in the type of content they are able to target and block: they can block inline scripts based on their text content or external scripts based on their URLs, but are currently unable to block only parts of external scripts. With the increased usage of script bundlers—which resolve dependency between authored JavaScript modules and generate a single script file—content blockers have no choice but to allow these bundles, which sometimes contain tracking scripts mixed with functional code. In this contribution, we investigate this practice by introducing a function signature technique making it possible to identify functions originating from known tracking libraries in other external scripts. We present robust function signatures that rely on function source code, able to detect tracking functions into existing or future scripts. With a web crawl, we collected a large set of script bundles and generated custom AST-based signatures for all JavaScript functions. By manually classifying functions from tracking scripts that access privacy-relevant APIs, we find that 4.37 % of unique scripts contain bundled tracking functions and that 22.7 % of domains from our sample contain such a script. As we are able to locate the functions responsible for tracking, it becomes possible to apply fine-grained blocking strategies, targeting specific functions within script bundles. Leveraging this ability, we present a proposed hybrid function-blocking strategy based on function signatures.

I.3 List of Scientific Publications

Parts of this thesis are adapted from the following publications:

[110] Romain Fouquet, Pierre Laperdrix, and Romain Rouvoy. 2022. JSRehab: Weaning Common Web Interface Components from JavaScript Addiction. In *Companion of The 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, 376–382. https://doi.org/10.1145/3487553.3524227

[111] Romain Fouquet, Pierre Laperdrix, and Romain Rouvoy. 2023. Breaking Bad: Quantifying the Addiction of Web Elements to JavaScript. ACM Trans. Internet Technol. 23, 1, Article 22 (February 2023), 28 pages. https://doi.org/10.1145/3579846

I.4 List of Tools and Prototypes

During the course of this thesis, we developed tools and prototypes to investigate our research questions, and collect and process data. The entirety of the source code is made available to obtain further details about the methodology and re-use these tools. The list of available artifacts is as follows:

- Breaking Bad: No-JS Breakage Detection Framework [14]
- Breaking Bad: Web Crawler [23]
- UBI: Repair WebExtension [36]
- UBI: Screenshot Diffing Tool [37]
- UBI: Screenshot Labeling Tool [38]
- UBI: Web Crawler [39]
- JSRehab: PostHTML Plugin Rewritting Interface Components with Noscript Alternatives [27]
- JSRehab: Bootstrap Version Detection WebExtension [26]
- JSRehab: Web Crawler [28]
- JsRipper: JavaScript Function Signature Generator [29]
- JsRipper: Web Crawler [31]
- JsRipper: Proof of Concept Blocking WebExtension [30]

I.5 Outline

This thesis is organized as follows.

We introduce the necessary background and context of this thesis in chapter II, along with the fundamentals concepts our contributions use. We present the actors of tracking on the web and their motives, before relaying the results of numerous studies having measured how widespread tracking is on the web. We then detail existing web tracking protections, before focusing on content blocking, exposing the strengths and weaknesses of current approaches. We finally revisit recent changes in cornerstone features of browsers, which vendors have been hesitant to make changes to.

In chapter III, we present a set of heuristics to measure the breakage of elementary HTML elements and constructs when JavaScript is disabled in the browser. We then leverage this set of heuristics to quantify the dependency on JavaScript of these elements, by completely blocking JavaScript and measuring the breakage of the elements. In addition, we also evaluate how much blocking JavaScript can help improve user privacy by minimizing the number of certain HTTP requests.

Building on this understanding on page breakage when blocking JavaScript, we introduce in chapter IV a new reasoning framework, that we call User Browsing Intent (UBI), to bridge the gap between the user and the browser, acknowledging that the user does not always require the entirety of the page to be working. This allows us to apply a more aggressive content-blocking strategy than what current systems are allowed to apply, and we repair specific types of breakage commonly occurring, thus making this aggressive strategy viable on more pages, hence improving user privacy.

Changing focus on server side, chapter V explores our server-side tool JSRehab, which provides a drop-in solution to website administrators to automatically substitute Bootstrap interface components with alternatives that do not require JavaScript for their interactivity in the browser, while preserving accessibility. When applicable, it allows to completely remove the Bootstrap JavaScript client-side library, improving security and decreasing device energy consumption.

Going back to the browser, chapter VI proposes a significantly more granular and content-focused JavaScript blocking WebExtension, making it possible to target individual JavaScript functions for blocking, using function signatures robust to script bundling. We leverage these function

signatures to measure how often tracking functions originating from known tracking scripts are bundled with functional code, effectively defeating existing blocking tools.

Lastly, chapter VII concludes this thesis by summarizing our contributions and proposing research perspectives for further improving web user privacy.

CHAPTER II

Background and Context

II.1 Tracking on the Web

In 2019, the W3C's Tracking Protection Working Group introduced the following definition of *tracking* with regards to the compliance with the Do Not Track preference [252]:

> Tracking is the collection of data regarding a particular user's activity across multiple distinct contexts and the retention, use, or sharing of data derived from that activity outside the context in which it occurred. A context is a set of resources that are controlled by the same party or jointly controlled by a set of parties.

In this thesis, we expand this definition to also include re-identification of the user and recordings of their activity within a single context, when it is not part of the primary goal of the page expected by the user—e.g., recording all user actions on a product page or re-identifying the user across browsing sessions even when the user is not logged-in.

Web tracking is widespread and has been the subject of numerous studies. In this section, we detail the actors of this industry and the reasons for web tracking. We then expand on the known tracking vectors, client-side technologies, and mechanisms enabling such tracking.

II.1.1 Actors and Motives

Different actors employ user tracking either directly or indirectly.

II.1.1.1 Business Model

Web tracking is a main component of web advertising, as part of user profiling. Indeed, behavior-based advertising companies, besides distributing advertisements banners, also profile users to deliver targeted advertisements tailored to consumer interests, therefore maximizing the probability they will click on the banner [273, 116]. Two common pricing models of online advertisers are CPM and CPC [116]. Cost Per Mille (CPM) pricing only depends on the number of advertisement impressions—i.e., the number of times the advertisement is shown to a user—this is similar to how

print advertising is usually priced: advertisement cost is fixed for a thousand impressions [279]. Having higher conversion rates—more consumers clicking on banners per impression—allows an advertiser to charge a higher CPM. Cost Per Click (CPC), however, takes into account the number of clicks effectively received by the banners: advertisement cost is thus defined per click. From these pricing models, and since the available advertisement space—i.e., the number of pages embedding the advertisement space of these companies—is limited, it ensues that advertisement companies have an incentive to track consumers to maximize the efficiency of their advertisement delivery—i.e., to show their advertisements to consumers who may be interested by the advertised products and may end up clicking on the banners.

Advertisers thus build customer profiles from their web browsing behavior, potentially including sensitive websites and search queries, leading to serious privacy harming practices.

II.1.1.2 Product Feedback and Visitor Profiling

When deploying websites, especially those with complex user interaction scenarios or where the user experience is crucial to the company revenue—e.g., e-commerce websites—website owners employ client-side libraries to observe user interactions and even allow to completely replay the browsing session on the website [144, 69, 241]—i.e., where the user clicked and typed. These aim to analyze the recorded browsing scenarios to improve the user experience, and increase the website revenue.

Website owners also deploy analytics systems, which range from simple page view counters to complex visitor profiling to understand who their visitors are. Counting unique visitors requires to populate the client-side storage with some value to detect that the visitor already browsed the page and should therefore not be counted again [208].

II.1.1.3 Government Surveillance and Hackers

Even when collected only for advertising purposes, profiling data about users may become relevant for government surveillance agencies. It has indeed been revealed, as part of Edward Snowden's disclosures, that the National Security Agency (NSA) has been leveraging Google's tracking cookies—initially indented for advertising—for targeting individuals to hack [114, 69]. More generally, the NSA is known to have partial access to data stored and processed by major American Internet companies with the PRISM surveillance program [115], potentially allowing it to profit from this data collection operated by private companies.

In addition, web tracking can also be implemented or exploited by independent hackers [103], to either follow their targets across the web or collect personal information about them.

II.1.1.4 Service Security

Finally, user tracking is also at the core of certain security systems, often to ensure the visitor is a human and not an automated crawler. To this end, some authentication systems rely on fingerprinting the user's browser [121], with the assumption that the number of devices a user logs in from is limited, and thus a log-in attempt from an unknown device or browser may help to detect fraudulent log-ins, such ones from as credential stuffing attacks. For instance, Google reCAPTCHAv3 and other

anti-bot services are believed to rely on an array of different metrics to tell humans and bots apart, including browsing fingerprinting, previous interactions of the user with reCAPTCHAv3, and on the presence of Google cookies [217, 61]. Many studies have also investigated how to use browser fingerprinting (see subsubsection II.1.2.3) for strengthened authentication [45, 155, 56, 157, 99].

II.1.2 Tracking Vectors

Client-side web tracking can take several forms and employ a wide array of technologies and browser mechanisms, sometimes by diverting them from their primary usage.

II.1.2.1 Network Identifiers

One of the most well-known means of tracking Internet users is to rely on network identifiers. In the case of the web, where the client and server machines are not part of the same local network, the only available network identifier is the IP address. Depending on how the IP address of the user device is allocated, it may be stable enough to allow to re-identify the user across browsing sessions. Mishra et al. have indeed measured that 87 % of users from their dataset retained one of their IP addresses for more than 30 days [178].

Because of the limited number of global IPv4 addresses, the usage of Network Address Translation (NAT) is widespread, thus effectively attributing a single IPv4 address to several users: for instance, in a residential network context the residents likely share a single Internet connection. This partly mitigates the uniqueness property of IP addresses that enables user tracking, but it may remain possible to discriminate between users, possibly leading to unique identification when used in conjunction with other tracking techniques.

Furthermore, the gradual deployment of IPv6 once again leaves users vulnerable to IP address-based tracking, as IPv6 addresses are globally unique.

II.1.2.2 Stateful Client-Side Tracking

Beyond IP addresses, cookies are a very well known client-side technology used for user tracking. Tracking cookies is a kind of stateful client-side tracking, as it stores a unique piece of data in the user's browser and expects to find it again when the user visits the website again.

HTTP cookies Many web APIs are dedicated to store data in the browser so that websites are able to persist state client-side. The most well-known is HTTP cookies, which have been introduced in 1994 [152] as the first way of having a concept of session. The cookie API is very primitive, as it consists only of a single string. Websites can set cookies using either an HTTP header—`Set-Cookie` [137]—or using the accessor JavaScript property `document.cookie`. For instance, a page could set a cookie named `foo` to the value `bar` with the following response HTTP header:

```
Set-Cookie: foo=bar; Max-Age=60
```

or the following JavaScript statement:

```
document.cookie = "foo=bar;max-age=60"
```

Those would set a cookie that expires after 60 s. If the `Max-Age` or `Expires` attributes are not set, the cookie is a *session* cookie, and is only valid until the browser is closed (note however that modern browsers restore the user session when restarting the browser and, therefore, session cookies may be valid longer than expected [104, 181]).

Cookies are sent by the browser with appropriate requests, in the `Cookie` request header [136]; cookies to be sent are concatenated and `key=value` pairs are separated by a semicolon and a space. Cookies can also be accessed with JavaScript using the `document.cookie` accessor property, but only if their `HttpOnly` attribute is not set; this helps protect again authentication token cookie collection with Cross-Site Scripting (XSS) attacks.

The optional `Domain` cookie attribute defines the host to which the cookie should be sent. If absent, the default host used is that of the current document URL. If the `Domain` is set to a domain which is not the document's domain or a superdomain of it, the cookie is said to be a *third-party cookie*.

Lastly, another notable attribute has been more recently introduced: the `SameSite` attribute. It allows to limit the scope of requests the cookie is sent with, to same-site requests only. The historical cookie-sending behavior is equivalent to `SameSite=None`, which is to send the cookies will all requests matching the set `Domain` and `Path` attributes. `SameSite` allows to prevent sending the cookies with cross-site requests [76], as an attempt to further mitigate Cross-Site Scripting (XSS) attacks, where an attacker triggers a request from a site A to another site B where the user is authenticated, and thus performs an action on this site B as the user.

Cookies have been used for tracking for almost as long as they have existed [164]. An example of historical tracking cookie setup is the following: an advertiser A has a banner embedded on two sites B and C; when the advertiser sends the banner on site B, it leverages the response to set a cookie in the user's browser—i.e., a third-party cookie—and this cookie will be sent to the same advertiser when the user later browses the second site C, thus tracking the user between the two sites.

With the gradual blocking of third-party cookies by browsers [214, 272, 183], this tracking setup is being replaced with other tracking mechanisms. For instance, some advertisers leverage DNS CNAME records to masquerade their third-party processing as a first-party subdomain [96], such circumventing the blocking of third-party cookies.

JavaScript storage APIs Long after the introduction of cookies in browsers, starting in 2009 [182], persistent JavaScript storage APIs have also been added to the web platform: `localStorage` and IndexedDB. `localStorage` is a substitute for cookies for specific use cases. Indeed, it allows the storage of several megabytes of data, while cookies are limited to 4,096 bytes [135]. In addition, as the data is only accessed through client-side JavaScript and is not sent with every request, it avoids sending unnecessary data on the wire, improving performance. IndexedDB was later introduced, in 2012 [243], as a complete client-side database, which can be queried with JavaScript. Predating these two APIs, another one, `sessionStorage`, was introduced in 2006 [244]. It is non-persistent and shares the same API as `localStorage`.

Browser caches Besides these browser APIs dedicated to client-side storage, a number of APIs whose aims are to improve performance and security introduce a form of client-side storage which

can be written to and accessed by websites and can therefore be leveraged for browser tracking. For instance, one of the most well-known performance-dedicated feature of browsers, the HTTP cache, could result in browsing history leakage or be exploited to re-identify users across sessions [197, 65], by first populating the cache with a unique set of pages, and later checking whether this unique set is present in the browser cache [107, 143, 274, 166], by measuring the load time of these resources. This was relying on the fact that the HTTP cache was single-keyed—i.e., the HTTP cache was not partitioned by origin and all sites shared the same cache [141]. Major browsers have since deployed double-keyed or triple-keyed caches, which mitigates this kind of attack [133, 148].

In addition to this, HTTP cache also specifies another performance mechanism which can be leveraged for re-identifying the user across sessions: ETags. An ETag is an HTTP response header which can be sent by an HTTP server as part of its HTTP caching strategy [85]:

```
ETag: "33a64df551425fcc55e4d42a148795d9f25f89d4".
```

The browser can then send this value when the resource is stale and it needs to check with the server whether the resource has changed, using the following request header:

```
If-None-Match: "33a64df551425fcc55e4d42a148795d9f25f89d4"
```

As the ETag is supposed to represent the contents of the resource—e.g., it can be a hash of the resource—the server can send the new version of the resource if it has changed, or reply with a `304 Not Modified` HTTP status otherwise. However, this mechanism can be exploited to uniquely re-identify users, as the ETag value is not guaranteed to actually represent the resource contents: the server can serve a dedicated resource and send unique ETags with each response and, since the browser will cache this value and send it back to the server as long as the resource is kept in the browser cache, the server will be able to link the two browsing sessions together.

Some client-side security mechanisms can also be exploited for user tracking. For instance, HTTP Strict Transport Security (HSTS) is a security feature allowing a website to indicate to the browser that it must always use HTTPS when connecting to the website instead of HTTP, thus preventing future downgrade attacks. A server can thus send the following HTTP header to instruct the browser to always use HTTPS for one year:

```
Strict-Transport-Security: max-age=31536000
```

As the browser needs to store this instruction, and since website are able to measure whether their website is loaded with HTTPS or not, HSTS can be exploited as a one-bit client-side storage [113] for re-identifying the browser across sessions.

Numerous other browser caches can be leveraged for re-identification and tracking, such as service workers [86], TLS 1.3 0-RTT resumption [230], or even favicons [225].

Clearly, blocking cookies is not sufficient to avoid stateful client-side tracking.

II.1.2.3 Stateless Client-Side Tracking

On top of the stateful tracking techniques expanded on in the previous section, another class of client-side tracking techniques has emerged, which relies on the diversity of software and hardware stacks used to browse the web: browser fingerprinting.

Browser fingerprinting entails collecting a set of browser attributes that may vary across users with the aim of building an identifying fingerprint of the browser. Examples of browser attributes include HTTP headers—e.g., the `User-Agent` string or the `Content-Language` header—some system values accessible in JavaScript—e.g., the screen resolution or the operating system—the exact image rendered by the Canvas element given some instructions [180, 158, 157], and on the installed browser extensions [226, 159, 224, 223]. The exact image rendered by the Canvas varies in particular depending on the installed fonts (including the system *emoji* font), on the font rendering stack, including font hinting and antialiasing [158, 157], and on hardware [154]. These attributes vary from browser to browser and, if unique enough, may lead to the unique identification of the browser and, thus, of the user.

Uniqueness The uniqueness of each attribute is quantified by its entropy: the higher the entropy, the more identifying the attribute is [100]. The entropy of an attribute cannot be obtained in isolation, from its definition alone. Instead, it is required to measure its actual diversity in the wild, from which the entropy can then be derived. Multiple studies have thus set up browser fingerprint collection web platforms, and published the results including the entropy of each studied attribute in isolation [100, 158, 117, 165, 57]. As building a fingerprint from multiple independent attributes greatly increases its uniqueness and its usefulness for re-identifying browsers—i.e, the joint entropy is higher than the sum of individual attribute entropies—these studies have also reported on the uniqueness of the whole browser fingerprint—i.e., when taking into the account the whole array of collected attributes—and found that a large share, or even most, browser fingerprints are globally unique.

Stability Uniqueness of browser fingerprints is not sufficient for them to be used for user tracking, they also have to be stable enough so that the same browser instance results in the same fingerprint in subsequent visits. Events which could lead to fingerprint instability include browser and extension installs and updates, hardware modifications, and window resizing [100]. Fingerprint stability has also been the subject of studies [100, 195, 247, 155, 56, 57], which were mostly focused on determining whether fingerprinting was suitable for increasing the security of web authentication. They have found high disparity within their samples, where high-entropy attributes—e.g., Canvas fingerprint—of some browser instances were stable for several months while they were changing every few days for other browser instances [155, 57] .

Fingerprinting deployment on the web Lastly, studies have measured the prevalence of browser fingerprinting on websites, through large-scale crawls [43, 195, 42, 102, 139, 207, 62]. These works rely either on dynamic analysis—observing access to sensitive browser APIs used for fingerprinting—or on static analysis—detecting known scripts or script features—to detecting web pages employing fingerprinting. The usage of browser fingerprinting reported by these studies varies between a few percents and around 10 %, and seems more prevalent on the most popular websites.

II.2 Web Tracking Deployment

Along with uncovering exploitable tracking vectors, previous works have also measured how widespread web tracking is.

In 2009, Krishnamurthy and Wills provided the first longitudinal analysis of web tracking [151]. They observed an increase of penetration of the top 10 third-party trackers from 40 % to 70 % between 2005 and 2008 and that multiple trackers were used on the same site. They also found evidence of DNS aliasing—now called CNAME cloaking—which disguises third-party servers as first-party. Mayer and Mitchell, Roesner et al. and Tran et al. have investigated in 2012 the behavior and dynamics of third-party tracking [176, 208]. In 2014, Acar et al. have then investigated new stateful and stateless tracking techniques along with cookie syncing [42]—i.e., client-side cookie sharing between different trackers. In 2016, Englehardt and Narayanan have conducted a large-scale crawl to obtain widespread tracking trends [102], they for instance found that news websites had the most trackers. Lerner et al. leveraged the Wayback Machine to run a longitudinal study of tracking, finding trackers as early as 1996, when the Wayback Machine started archiving websites [164]. More recently, in 2018, Sánchez-Rola and Santos performed a large-scale crawl and used machine learning techniques to recognize tracking scripts [212]. They found that more than 90 % of websites were tracking users. Dimova et al. have conducted large-scale measurements of CNAME cloaking, consisting in using DNS configuration to disguise third-party trackers as first-party, thus circumventing some blocking strategies [96]. They discovered than popular websites were more likely to deploy this evasion technique, and that 10 % of websites from the top 10,000 Tranco were using it. Yang and Yue and Cassel et al. have investigated the differences between mobile and desktop web tracking and found that both mobile and desktop have their own specific features, as tracker sets of both platforms do not completely overlap. In 2022, Dambra et al. leveraged telemetry data from an antivirus company to study trackers from the vantage point of users: they concluded that 80 % of websites visited by users included trackers, Google's trackers being present on 63 % of them [91].

II.3 Web Tracking Protections

The privacy-harming nature of web tracking, along with its prominence and diversity, calls for a wide range of web tracking protections. Recent years have seen the advent of stronger privacy laws, in particular in Europe and in California, with the General Data Protection Regulation (GDPR) and the California Consumer Privacy Act (CCPA). These are aimed to protect personal data, by regulating their usage and processing, and their collection process. However, not all web tracking methods rely on personal data, and thus are not covered by these laws. Moreover, it is not guaranteed that these laws are enforced on the sites the users browse. Therefore, technical protections are needed against web tracking, that users can employ to protect themselves online. The challenge that these protections face is to defend against tracking while minimizing user friction.

Web tracking protections can be divided into a few categories depending on their action and usage. In this section, we detail the different kinds of existing web tracking protections, built in browsers or deployed as browser extensions. We leave the content blocking approach for the next

Table II.1: Deployment of network and complete state partitioning in browsers; PB stands for Private Browsing

Browser	Network Partitioning	State Partitioning
Firefox	2021 [101]	2021 (PB and ETP Strict mode [47]) [133], 2022 [183]
Chrome	2020 [148]	2023 (planned) [163]
WebKit (Safari)	2013 [2]	2017 [268]
Brave	2021 [239]	2021 [238]

section, as it is the main focus of this thesis. The presented protections are not mutually exclusive and may often be used in conjunction.

II.3.1 State Isolation

One of the most powerful protection against cross-site tracking is state isolation. We have seen that many web APIs store state client-side, and state isolation consists in segregating the state of different sites in separate storage containers, to prevent the unwanted access of state stored by one party by another party. This is a major change in the web platform compared to its historical architecture, but major browsers have recently implemented and deployed such state isolation. Chrome has indeed implemented HTTP cache isolation in 2020 [148] while Firefox deployed network isolation in 2021 [101]. Firefox then implemented more complete state partitioning, including that of cookies and JavaScript storage, first in private browsing windows in 2021 [133], then in normal windows as well in 2022 [183]. Such state isolation—also called storage partitioning—is being standardized by the W3C [123]. Current and planned deployment of this feature in major browsers can be found in Table II.1.

Firefox has also introduced another concept, tab Containers, enabling users to manually isolate browser storage at the tab level [189]. This feature was deployed in 2017 [250] as an optional WebExtension, and allows the user to define contexts—e.g., work, banking, etc.—and to select which context to use when opening a new tab. Browser storage is then shared between tabs of the same context, but isolated from other contexts.

II.3.2 Shorter-Lived State

An additional tracking protection, used in conjunction with state isolation, involves reducing the lifetime of client-side storage, by deleting parts of their contents either periodically or when the user chooses to do so. This can be an efficient solution against long-term tracking while avoiding page breakage, as websites are still allowed to write to and read from the storage, making this protection transparent from their point of view. The oldest manifestation of this strategy is the private browsing mode of browsers, introduced as early as 2005 in Safari [170], which keeps persistent storage in-memory so that it is effectively cleared when all private browsing windows are closed.

Browser extensions have also implemented this concept by automatically deleting unused client-side storage when a tab is closed. To this end, storage associated with a domain is cleared when the last tab having a page from this domain open is closed. The Cookie AutoDelete WebExtension

currently implements this behavior [97]; it was predated in 2013 by Self-Destructing Cookies, which was abandoned in the switch to WebExtensions.

WebKit and Brave have implemented built-in features which limit the lifetime of storage, under some conditions. WebKit, as part of its Intelligent Tracking Protection (ITP) system, implemented the partitioning of third-party tracking cookies, which were then deleted at most 30 days after their storage [268, 269, 270, 271]. Brave clears all third-party storage when the site is closed [238] and has evaluated the web compatibility of this strategy [145].

II.3.3 Permission System

Another kind of protection is simply to ask the user before allowing a website to use a sensitive API. As this adds an additional burden on the user and degrades the user experience, this strategy is usually limited to the most privacy-invasive features, which also happen to be used by a small number of websites [221], for APIs whose behavior are easily understood by users: Geolocation [78], microphone and camera [79], and more recently third-party Storage Access [81]. When a website wants to use one of these APIs, the browser prompts the user to allow or deny its usage, through a browser-shown pop-up.

II.3.4 Disabling Browser Features

An additional strategy employed by browsers to reduce client-side tracking is to reduce the surface of APIs that can be leveraged for tracking, or that the browser is currently unable to prevent them from being used for it. For instance, Firefox, as of 2023, does not allow storing data with IndexedDB in private browsing windows [15]. Service workers APIs are also disabled in private browsing windows [16]. However, this has the downside of making the private browsing mode detectable by websites.

II.3.5 Anti-Fingerprinting Protections

The tracking protections presented in the previous sections are mostly directed towards preventing stateful tracking—i.e., tracking exploiting client-side storage. In this section, we detail the main directions to reduce the risk of stateless tracking, also known as browser fingerprinting, as presented in subsubsection II.1.2.3.

II.3.5.1 Reduced Configuration Disparity

As browser fingerprinting relies on the uniqueness property of browser fingerprints, one the possible defense strategy is to reduce the diversity of the browser fingerprints in the wild. This is a challenging enterprise, as it collides with the desire of users to personalize their browser and to set it up to match their needs. Indeed, reducing the potential of the browser to be fingerprinted in this manner requires to set the same configuration for every user—e.g., the language, timezone, user-agent string, available fonts, etc. This approach is employed by Tor Browser, which spoofs globally unique values of attributes which could otherwise vary between users [201]. Thanks to the efforts

of the Tor Uplift project [3], this uniform configuration can be applied to Firefox as well, by setting the `privacy.resistFingerprinting` (RFP) preference to `true` [46].

This configuration uniformity strategy needs to be followed with great care, as it can easily be undermined by unintended user modifications, such as installing browser extensions whose effect can be observed by websites, making these users stand out more.

II.3.5.2 Randomization

The other defense technique against browser fingerprinting is to try to defeat the stability property, by randomizing extracted values of certain attributes [194, 156]. This technique makes sense for attributes for which it is too difficult to reduce their possible values, or whose values are influenced by parameters that the browser is not in control of, such as hardware or operating system differences. This strategy is thus adopted in conjunction with the previous one, and is often applied to the Canvas element when deployed. For instance Firefox has implemented it in 2020, but this defense is only enabled when using RFP [213] or in Tor Browser. Brave has also deployed this technique in 2020. Multiple browser extensions also provide randomization [149, 191], with varying degrees of robustness [92], due to their implementation as WebExtensions. Randomness can also be introduced by browsers in the mechanisms exploited to collect attributes, such as JavaScript timers [209]; browsers have indeed lowered the resolution of these and artificially introduced jitter to make timing attacks more difficult.

II.4 Content Blocking

In the previous section, we have introduced different types of web tracking protections. They are effective at varying degrees and can be used in conjunction. However, a large share of web tracking is not addressed by the aforementioned protections—notably same-site tracking—which therefore calls for an additional layer of protection: client-side content blocking. Content blocking involves targeting and blocking specific web resources. In this thesis, we focus on leveraging content blocking for improving user privacy in the browser, therefore aiming to block content enabling or taking part in user tracking.

Three main criteria are available for targeting content to block in the browser:

- content's location (based on content's URL)
- content's contents
- content's behavior

Currently deployed tracking protections mostly rely only on content's location for targeting and blocking [177].

II.4.1 Filter Lists

One of the oldest type of content blocking is filter lists [1]. Filter lists are blocklists that describe content to block based, in particular, on their URL. They have existed for a long time for security

purposes, especially to block malware files [169], but have also been used to block advertisement banners and privacy-invasive content.

II.4.1.1 Syntax and Capabilities

In this section, we present the core features of the filter list syntax used by AdGuard [44], Adblock Plus [202], and uBlock Origin [131]. The most well-known filter lists following this syntax are EasyList (advertisement blocking) [105] and EasyPrivacy (privacy) [6]. Blocking tools also often ship their own supplementary lists.

Network rules Network rules allow to block network requests, *before* they are actually sent by the browser. Since the resource has not yet been loaded, it would be impossible to make a blocking decision based on its content, much less from its behavior. Network rules therefore rely on the URL, on the type of resource, and on the context of the request—e.g., is the request third-party?, what is the domain of the first-party document? The syntax of network rules feature many wildcard characters, to make it easier to write general rules matching only parts of URLs. For instance, the following rule will block all outgoing requests whose URL path is `/pixel.gif` and having a query string, blocking some tracking pixels:

```
/pixel.gif?
```

The following rule matches and blocks all third-party requests whose URL's domain ends with the string `google-analytics.com`; the dollar sign marks the start of optional rule options:

```
||google-analytics.com^$third-party
```

Similarly, the following rule blocks all external scripts whose URL path is `/analytics/js`:

```
/analytics/js$script
```

In addition, filter lists also feature exception rules, which unblock requests that would otherwise be prevented by a blocking rule. They start with `@@` and otherwise follow the same syntax. The following rule thus always allows scripts coming from URLs whose domain ends with `adobedtm.com` and whose path contains `/satellite-`.

```
@@||adobedtm.com^*/satellite-$script
```

Finally, network rules can redirect requests to a local, extension-provided, benign resource: this technique is called resource replacement. Syntax for this technique varies from rule engine to rule engine. The following rule, from the uBlock Origin's list, redirects the matching advertisement script requests to a no-operation script, provided by uBlock Origin, that does nothing:

```
||cloudfront.net/ads/*$script,redirect=noopjs,domain=cbs.com
```

Blocking of inline scripts HTML allows to embed scripts inline, using the `<script>`. As these scripts are already included in the HTML document, they will not trigger an HTTP request and, therefore, cannot be blocked using network rules. To remedy this problem, content blocking tools make available two different solutions: HTML filtering and script defusing.

HTML filtering processes the HTML response data and allows to remove HTML tags before the browser engine even parses the received HTML document. Tags to remove can be targeted using either CSS selectors or a custom selector allowing to match the inline script text contents [131]. For instance, the following HTML filtering rule deletes inline script elements whose text contains `FingerprintJS`, on `userscloud.com`:

```
userscloud.com##^script:has-text(FingerprintJS)
```

The other available technique to defeat unwanted inline scripts is to leverage scriptlets (called snippets by Adblock Plus) to prevent their successful execution. To this end, some content blocking tools allow to inject in the page predefined scripts, named scriptlets or snippets, designed for specific purposes. These tools do not allow the filter rules to contain arbitrary scripts to avoid arbitrary code execution from filter lists. uBlock Origin thus provides scriptlets to abort the current inline script or to abort a script when a given property is read from or written to [131]. Aborting the script execution is performed by throwing a `ReferenceError`, effectively putting a halt to the script execution. For example, the following rule aborts inline scripts (`acis` stands for "abort current inline script") whose text contains the string `sp.blocking` and accessing the `$` global variable (referencing the jQuery library):

```
gamespot.com##+js(acis, $, sp.blocking)
```

Similarly, the following rule aborts all scripts writing to the `Fingerprint2` global variable, thus preventing the definition of this fingerprinting library (`aopw` stands for "abort on property write") and thus its usage:

```
shink.me##+js(aopw, Fingerprint2)
```

Cosmetic rules Along with the previously discussed network rules, content blocking tools can also feature cosmetic features, dedicated to *hiding* certain page elements. They are mostly used to conceal advertisements banners that cannot be hidden using network rules or to hide the remaining, empty advertisement space. However, as their name suggests, they provide no privacy benefits. Cosmetic rules rely on CSS selectors to target page elements to hide and inject appropriate CSS to hide the relevant elements. For instance, the following cosmetic rule hides all elements having the class `adbannerright`:

```
##.adbannerright
```

II.4.1.2 Strengths

Filter lists thus have several strengths. They are indeed simple to understand and to deploy, as browsers provide browser extension APIs making it straightforward to build WebExtensions that

selectively block requests. Indeed, the `webRequest.onBeforeRequest()` [84] introduced an interface that makes it easy to block individual outgoing HTTP requests before they are sent by the browser. This method indeed takes a callback function, passing to it the URL of the request, along with the resource type and other data, which is able to ask the browser to cancel the request.

In addition, content blocking based on filter lists is easily predictable: the requests or page elements will be blocked if and only if they are present in one of the lists used by a user, these lists being public and easily viewable.

Finally, the filtering process based on filter lists is relatively efficient, as it mostly rely on the URL only, which is important as all outgoing requests need to classified.

II.4.1.3 Weaknesses

However, filter lists also suffer from several weaknesses.

Incompleteness and disparity in coverage Even though filter lists are decent enough as suggested by their popularity, there is no guarantee of their completeness and other, recent studies have also shown that a significant share of trackers is still not blocked [73] and that coverage heavily depends on geographical regions and languages: regions whose language have few speakers tend to have lower coverage [218]. This is easily explained by the fact that these filter lists are mostly crowd-sourced [53] and are thus generally as good as the "crowd" is large. This is despite previous works which investigated the automated generation of these lists, to complement this manual effort [124, 108, 161, 160].

List update latency Furthermore, as filter lists are updated by humans and there is no automated watching process, filter lists are likely to be lagging behind the websites. Indeed, filter lists rely on low-level selectors—URLs and CSS selectors—and are thus susceptible to break when websites change, be the changes local or a complete website redesign. Studies have investigated these dynamics, including the lifetime of rules [140] and have found that the newest rules are significantly less used than the oldest [222], as the latter are usually more generic.

Always-increasing size Because it is not possible to know whether a filter rule is still useful—i.e., whether the pages where the rule would be triggered on still trigger it—filter lists tend to become "append-only" in practice, that is, rules are added but seldom removed. This may eventually hinder their use on mobile, where the hardware resource constraints are higher, as the lists would become prohibitively big. This has also been brought under the spotlight recently, as Google has announced the progressive sunset of the Manifest V2 API of WebExtension [94], replaced with Manifest V3, which introduced tight limits on the number of rules filter-list-based content blocking WebExtensions would be able to enforce [95, 222]. As of 2023, the minimum number of rules that a browser must support according to the Manifest V3 is indeed of 30,000, smaller than the number of rules in EasyList alone. Snyder et al. have investigated reducing the set of rules by keeping only the most frequently used rules [222].

Misalignment of aim and method Last but not least, a major underlying weakness of filter lists is that they rely on the location of the resources to be blocked, while the motivation to block them is because of their behavior. This misalignment of the aim—preventing certain behavior—and of the method—blocking resources based on their location—is at the root at the brittleness of filter lists. In particular, this makes filter lists easy to evade, as moving the resources—i.e., changing their URLs—is enough to evade the network rules [73, 167]. Some websites also inline specific scripts which would normally be external scripts, effectively avoiding blocking [73] by network rules.

Moreover, the unit of blocking of those filter lists is the whole resource, and it is thus impossible to block only parts of a script.

II.4.2 Resource Replacement

Filter lists are mostly aimed at simply blocking resources. However, when these resources are scripts that other scripts depend on, merely blocking them can introduce some forms of page breakage. Indeed, as illustrated by Listing II.1, if a blocked script was providing a function called before some functional code, a `ReferenceError` will be thrown when trying to execute that blocked function as it is thus `undefined`, and the following functional code will therefore not be executed. As a solution to this problem, content blocking tools have introduced means to inject dedicated scripts, called *shims*, which replace these blocked scripts. Shims thus contain no-operation functions, that get called in place of the original, blocked functions. uBlock Origin has introduced this solution as part of its scriptlet injection mechanism [130], presented above: filter lists can request predefined scripts to be inserted in the indicated websites. Firefox Enhanced Tracking Protection also injects shims when blocking certain tracking scripts [184]. These shims are manually written, and their contents are kept minimal while matching the exposed API of the blocked script they replace.

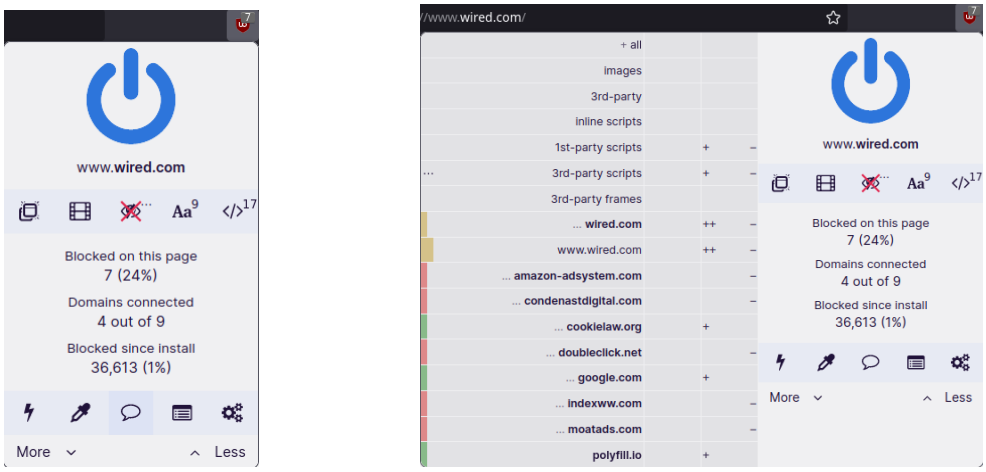
```
function() {
  function_from_a_blocked_script();
  functional_code();
}
```

Listing II.1: Example of a JavaScript function that will throw a `ReferenceError` if `function_from_a_blocked_script` does not exist, thus preventing `functional_code` to be executed.

Brave has developed an automated solution [220] to generate replacement scripts for scripts bundling both tracking and functional code, producing scripts where browser privacy-sensitive APIs are transiently neutralized. While being an automated and more general solution, it raises performance and legal concerns about the distribution of these scripts, as the whole modified scripts need to be shipped by the browser.

II.4.3 Dynamic User Rules

The previously discussed filter lists can be called *static* rules, as they are written once and distributed to all users. Advanced users, however, may want to enforce their own rules on individual websites. To this end, several content blocking tools provide graphical interfaces so that users can define



(a) uBlock Origin pop-up in normal mode (b) uBlock Origin pop-up in advanced mode

Figure II.1: uBlock Origin pop-up in normal and advanced mode

what requests to block or allow. These tools are limited to defining rules based on the content type and on the destination domain of the requests, not on their complete URL, as it would be difficult to visually represent this kind of selection. uBlock Origin makes available such rules in advanced user mode (which can enabled in the settings), as shown in Figure II.1. uMatrix provides a more complete version [128], with the downside of being more difficult to approach (Figure II.2). Finally, the NoScript extension (Figure II.3), shipped by default with the Tor Browser, is dedicated to blocking scripts, hence its name.

II.4.4 Auto-Adaptive Blocking

The content blocking solutions discussed above either rely on static filter rules or on user intervention to decide which content should be blocked. To overcome the limitations of each approach, some tools have tried to provide auto-adaptive solutions, which locally observe the user browsing and learn from it what content is tracking the user across websites. WebKit had introduced such a system with Intelligent Tracking Protection (ITP) [268, 270] in 2017, which classified certain domains as tracking and applied a stricter information sharing policy to them, but had to change this behavior and apply a more uniform treatment of third-party domains after Google's security team disclosed [142] that, since this introduced a global state which could be written to and read by websites, it could also be exploited for tracking or leaking parts of the browsing history. Similarly, the Privacy Badger browser extension, developed by the Electronic Frontier Foundation (EFF), used to provide a similar anti-tracking solution, learning tracking domains while the user browses, but had to disable this behavior following similar disclosure [60].

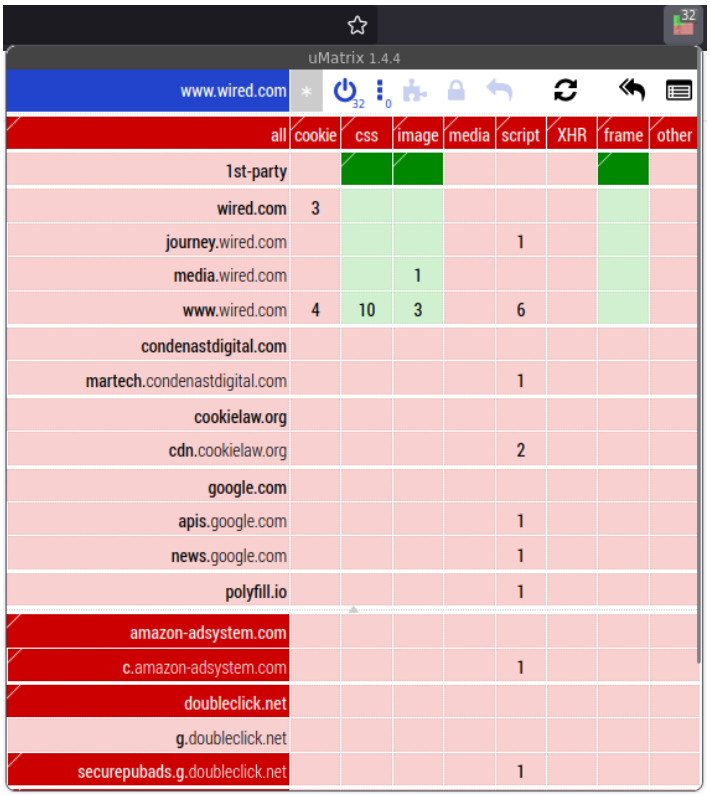


Figure II.2: uMatrix pop-up where only first-pary CSS, images, and iframes are allowed

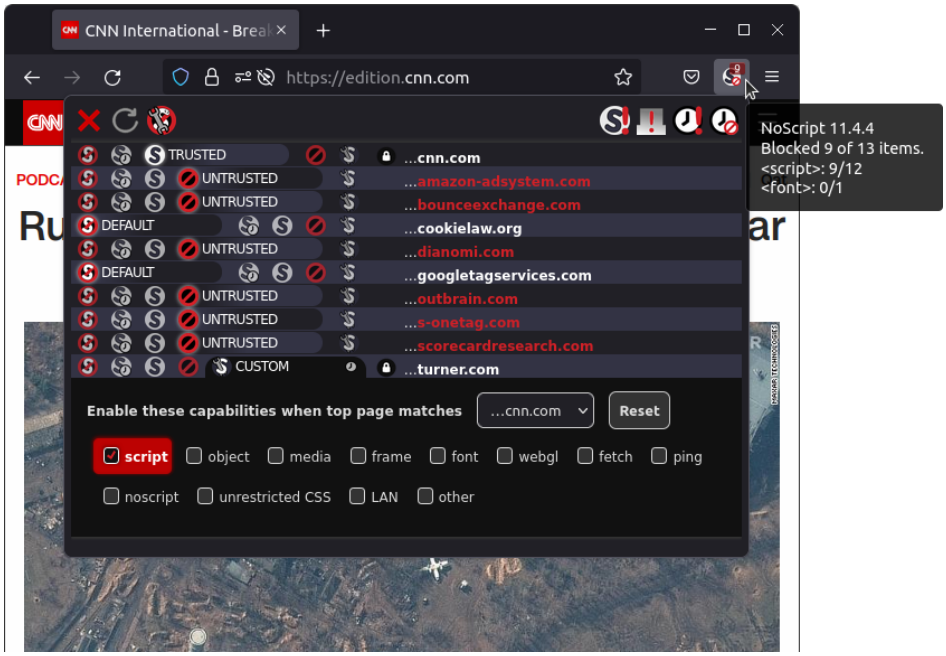


Figure II.3: The NoScript extension allows to selectively block scripts per domain (CC BY-SA 4.0 https://noscript.net/)

II.4.5 Additional Benefits of Content Blocking

As this is the focus of this thesis, we mostly focused on the benefits of content blocking for privacy so far. However, content blocking also has numerous secondary benefits, originating from the fact that blocked content is neither downloaded or executed.

II.4.5.1 Web Security

By preventing the execution of certain scripts, content blocking can improve browser security.

Reduced attack surface By reducing the attack surface of the browser, as specific content is blocked, it is harder to exploit browser and operating system vulnerabilities, diminishing the role of the web browser as a major attack vector. Advertisements on otherwise innocuous websites can indeed be used as an infection vector [147, 153]—then constituting malvertising. The NSA and CIA have for instance made public their use of ad-blockers to mitigate this risk [89].

Cross-Site Scripting (XSS) One of the main web attacks that content blocking may help prevent is one of the most widespread web attacks: Cross-Site Scripting (XSS). XSS can happen when server-side or client-side code injects user-provided inputs directly into the HTML, without proper validation or escaping. Successful XSS attacks allow an attacker to execute JavaScript code they control within the victim's web page, with the same privilege as the website owners. These attacks can be leveraged by attackers to steal the victim's authentication cookie, thus effectively taking control of their account. XSS can also be chained with a CSRF attack, enabling the attacker to perform actions on another website where the victim is logged in. Although XSS is a well-known attack, it is still the most common web attack vector according to HackerOne's 2020 security report [126]. Preventing the execution of specific scripts helps defend against XSS in the browser.

II.4.5.2 Reduced Data Usage, Performance, and Energy Consumption

As one aspect of content blocking is to prevent some specific types of content to be downloaded by the browser, it can lead to reductions in the amount of data transferred by the device. This therefore brings two other benefits. For users connecting to the Internet using metered connections, for instance on pay-as-you-go mobile contracts, using content blockers may help reduce their phone bill [246]. Browser performance, in particular page load speed, can benefit from blocking content, since the blocked content will not be competing for bandwidth. Lastly, as the amount of data transferred can be diminished with content blocking, it can also reduce the energy consumption of the device [246], thus increase its battery lifetime.

II.5 Recent Changes in Cornerstone Features of Browsers

For a long time, browser vendors seem to have been hesitant to change cornerstone features of the client-side web platform, even when these constituted major privacy holes. Many of these earliest features were effectively considered untouchable, for fear of breaking websites.

II.5.1 Request HTTP Headers

Request HTTP headers have been introduced as early as 1992, in the HTTP/1.0-draft specification [64]. This specification introduced the content negotiation headers which are still in use today: `Accept`, `Accept-Encoding`, and `Accept-Language`. Along with them, two other HTTP headers of interest are part of this specification: `User-Agent` and `Referer` (the misspelling of the `Referer` header has been preserved until today, for backward compatibility).

II.5.1.1 User-Agent Header

The `User-Agent` header is a string sent to web servers advertising the browser vendor, product name, and version. It has been useful when browsers were not implementing the same features, especially when not all browsers supported framesets, as web servers could send different versions of HTML documents to different browsers. However, now that browser compatibility has greatly improved, this header has little use and is nonetheless a significant fingerprinting vector [117, 165]. Google is experimenting replacing this header with a set of new headers, called client hint headers, with which browsers could send specific pieces of information when requested [240] with the hope that the resulting entropy would be lower, while still allowing differential serving. If this experiment is successful, the `User-Agent` request HTTP header may be replaced with client hints, and browsers may stop sending it with every request, reducing their fingerprintability.

II.5.1.2 Referrer Header

The `Referer` header serves a completely different purpose: the browser was to optionally send it with every request—e.g., when clicking a link or loading an image—its value being the address of the page which provided the link to the requested resource. This header has been used for tracking, Search Engine Optimization (SEO) and security.

As the `Referer` header used to always contain the complete URL, it could easily be used for user tracking, as websites could thus learn the previous page users were on. Websites were also able to leverage the `Referer` value to learn about the search engine keywords that led to their websites. Indeed, search engines result page URLs used to contain the keywords unencrypted, which were thus received as is by websites when following search result links. In 2013, Google has started encrypting search keywords in URLs to avoid directly leaking them to websites in this manner [229], and websites now have to register to Google Search Console to obtain the search keywords having led to their websites [175].

The `Referer` value is also used to try to protect against Cross-Site Request Forgery (CSRF) attacks [205]. CSRF attacks aim to leverage the fact that a user is logged into a website A to execute actions on this site A on behalf of the user without their knowledge. A successful CSRF attack requires a page B controlled by the attacker triggering a request to the website A and leveraging their account—e.g., ordering a product or posting social media content—effectively impersonating them. The `Referer` has been used to check whether action-performing requests were indeed originating from the same site: since the `Referer` contains the origin domain, it is easy for website A to verify that the request comes from its domain. However, relying on this method only is not robust, as not

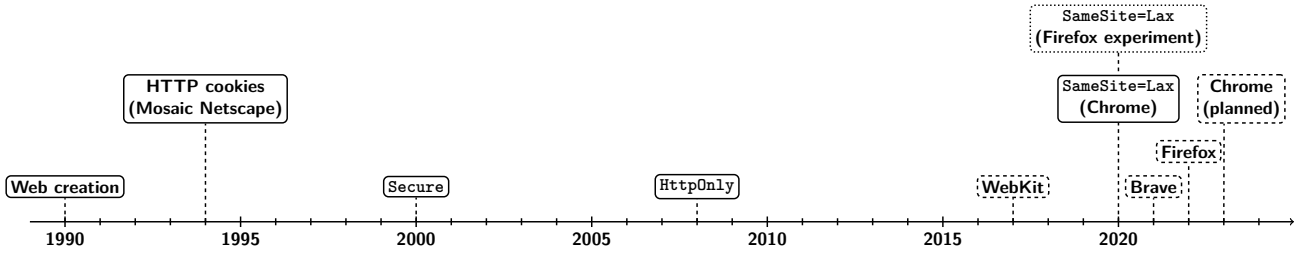


Figure II.4: History of HTTP cookies and their security and privacy features. Dashed boxes represent the deployment in browsers of isolation of cookies for all users in normal windows.

all requests contain the `Referer` and website thus have to allow requests containing no `Referer`. As techniques exist for attackers to avoid sending the `Referer`, this defeats this security strategy [205].

Because of the privacy-invasive nature of this header, it is being replaced and gradually removed. A new request HTTP header has indeed been introduced, `Origin`, which only contains the origin—i.e., the scheme, hostname and port—that caused the request [80]. Contrary to the `Referer` header, it does not contain the URL path or query string. It is in particular sent with all `POST` requests, allowing to replace the `Referer` header for its security purpose. Deploying CSRF tokens—i.e., unguessable values—in HTML forms, which will be submitted along with its other values and then checked by the server, is still the recommended way of preventing CSRF.

In 2017, WebKit started to limit the `Referer` value to the origin only for all cross-origin requests, as part of ITP; before that, it was only stripped for requests to domains classified as tracking by ITP [270]. To be able to gradually retire the `Referer`, an additional response HTTP header has been introduced, `Referrer-Policy`, which allows to specify whether the browser should send the `Referer` header and, if so, what URL parts it should include. In 2021, Firefox, along with other browsers, have changed the default behavior from `no-referrer-when-downgrade` to the stricter `strict-origin-when-cross-origin`, which effectively strips the URL path and query string from referrers of cross-origin requests [162]. Later in the same year, Firefox stopped abiding by lesser strict policies, therefore ensuring that URL paths are never leaked in the referrer in cross-origin requests [134].

While introduced very early in the history of the web, in 1992 [64], browser vendors have thus been hesitant to change the behavior of these headers until a few years ago, even though they posed privacy issues for every web user.

II.5.2 Cookies and Other Client-Side Storage Types

Cookies and other client-side storage mechanisms have also undergone heavy transformations since the inception of cookies, in 1994 [152], but, as summarized in Figure II.4, these transformations only started relatively recently.

II.5.2.1 Cookie Sending Behavior

As we have covered in subsubsection II.1.2.2, the historical behavior was to send the cookies with every request to the domain the cookie was set for, whose path matched the defined path, regardless of the request resource type or whether the request was cross-site [112].

With the introduction of the `SameSite` attribute in 2016 [245], it became possible for websites to restrict this behavior, and only send the cookies for same site requests using `SameSite=None`. In 2020, Chromium has gradually changed the default value of `SameSite` to `Lax` for all users [74], preventing cookies to be sent with normal cross-site requests, such as image requests. Firefox has also been experimenting with changing the default value in 2020 [76].

This is a significant shift in the web platform, which can have a big, positive impact on user privacy.

II.5.2.2 State Isolation

In the same vein, browsers have also rolled out another major change in the web platform: client-side state partitioning. Historically, state—i.e., cookies, `localStorage`, HTTP cache, etc.—was shared between all websites and tabs, websites could thus leverage resources loaded in the HTTP cache by other websites. However, sharing this state made it possible for websites to observe what other websites had, directly or indirectly, written to it, sometimes by using timimg-based attacks. For example, a website A could purposefully trigger the loading of a resource originating from another website B and measure its loading time; it the resource was loaded very quickly, it definitely came from this shared local cache and website A could therefore infer that the user had recently visited website B (and even a specific page of website B by selecting the resource on purpose), leaking part of the user's history [107, 141, 143, 274, 166]. To defeat this kind of attack, some browser vendors have recently deployed thorough state partitioning, keying client-side state to the first-party domain as well, as we have detailed in subsection II.3.1.

That is a major change in a fundamental subsystem of the client side of the web platform, and can significantly help to reduce cross-site tracking.

II.6 Conclusion

Web tracking has seen rapid development since the inception of the web in 1990. It is now widespread, as almost all websites contain third-party trackers that collect information about users and follow them across websites. Trackers also record how visitors use web pages, as if they were watching over the users' shoulders. These observations and recordings are used for targeted advertisements and for tailoring the websites for increased revenue. They constitute private data about users, which can be exploited by insurance companies to learn about their consumers' health conditions, for vote manipulation using micro-targeting relying on user profiling, or even government surveillance. The lack of online privacy can also lead to "chilling effects" and self-censorship, deterring users from educating themselves about sensitive topics.

This calls for strong web tracking protections that users can employ to protect themselves when browsing the web. Different tracking protections are already deployed in browsers and in the form

of browser extensions, implementing different strategies such as state isolation, regular storage deletion, or content blocking. As we have seen, content blocking prevents the downloading or execution of unwanted resources. As other privacy protections are not always able to protect the user from the effects of unwanted scripts, content blocking is a crucial part of client-side tracking solutions. However, it still suffers from several weaknesses due to it mostly relying on filter lists. Because these may be incomplete, lagging behind, or unable to target specific tracking mechanisms, users are left vulnerable to certain kinds of tracking on the web.

The overarching theme of this thesis is to reduce the amount of JavaScript executed in the browser, as JavaScript is the cornerstone of extensive tracking, and undermining this tracking vector would greatly benefit user privacy and security.

In chapter III, we start by investigating page breakage when disabling JavaScript and measuring the dependency on JavaScript of common web page elements. We report on the breakage rate of various page elements when JavaScript is disabled and on how much disabling JavaScript can help protect user privacy. Building on the knowledge acquired from this first contribution, we introduce in chapter IV the concept of User Browsing Intent (UBI) and, focusing on the 'read-only' UBI, we devise a set of repairs to make it more viable to browse the web without JavaScript by default, enabling JavaScript only when required. In chapter V, we switch on the server side, and propose an automated rewriting system to replace interface components that rely on JavaScript with noscript alternatives, and discuss the benefits, especially regarding device energy consumption and security. Lastly, we introduce in chapter VI a signature scheme for identifying JavaScript functions and leverage it for detecting tracking functions from known tracking scripts bundled with functional code, which cannot be blocked with existing content blocking tools.

CHAPTER III

Investigating Page Breakage when Disabling JavaScript

JavaScript is one of the main client-side technologies enabling in-depth user tracking. Some users thus would like to disable it in their browser to improve their privacy—or at least block most page scripts—but it is unclear to them what page elements are missing or broken. These users would benefit from being able to visually locate these broken elements. Moreover, we want to investigate page breakage when JavaScript is disabled and quantify the breakage of page elements and evaluate the commonly-encountered claim that "everything breaks when you disable JavaScript." No standard interface exists to detect page breakage or even the breakage of individual elements: existing literature has thus been reduced to manually evaluate page breakage when trying to reduce the amount of client-side JavaScript for performance reasons [72, 71].

In this chapter, adapted from our paper published in TOIT [111], we propose an automated, heuristic-based, bottom-up JavaScript framework, built in a browser WebExtension, which detects and locates broken page features, when JavaScript is disabled. Through a large-scale web crawl, 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, and that disabling JavaScript reduces the number of tracking requests by 85 % on average.

This contribution 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, relying on the limited amount of information available client-side
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

III.1 Detecting 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.

III.1.1 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.

III.1.1.1 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.

III.1.1.2 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 contribution, 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 A.1 in the appendix, and of popular web component libraries (in Table A.2), aided by the Web Accessibility Initiative — Accessible Rich Internet Applications (WAI-ARIA) widget list [254] 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 [263, 50], 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 III.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 [186], which is gradually getting adopted [50] and implements native image lazy-loading, as shown in Listing III.2. This native behavior of the browser is disabled when JavaScript is disabled, to prevent tracking of the scroll position [186].

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.

```
<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 (III.1) Lazy loading images with JavaScript

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

Listing (III.2) Lazy loading images without JavaScript

Figure III.1: Lazy loading images

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 contribution, 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 III.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 [264].

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 III.3: A valid form

Lone Controls In this contribution, 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 III.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 contribution, 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 `"#"`,
- have a `javascript:` pseudo-protocol `href` that is a no-operation, see Listing III.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 [265].

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 III.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.

[1] An `<a>` element with no `href` but with a `name` is an obsolete, HTML4 way of marking a destination for another anchor [251].

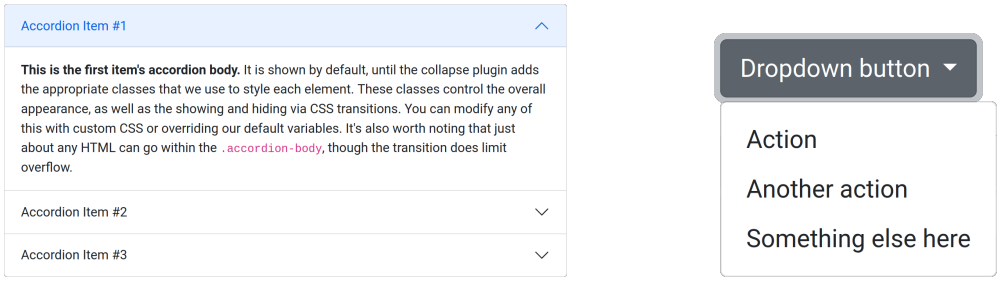


Figure III.2: Examples of disclosure buttons (from the Bootstrap 5 documentation [235, 233])

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 III.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 [233, 235]: 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 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 III.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 III.5: Accordion working without JavaScript

A simple disclosure element can also use the native element pair `<details>` and `<summary>`, see Listing III.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 III.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.

III.1.2 Page Feature Relevance

A web page is composed of different sections with each their own purposes, as can be seen in Figure III.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 `ids`, 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 `ids` 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.

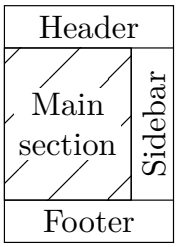


Figure III.3: Common page structure (the sidebar can be on either side of the page)

III.2 Data Collection

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

III.2.1 Crawled Websites and Pages

We used a subset of the Hispar list [58], 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 [59], 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 III.1.1 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 the preference `javascript.enabled` set to `false`, which disables JavaScript globally for this instance, as depicted in Figure III.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

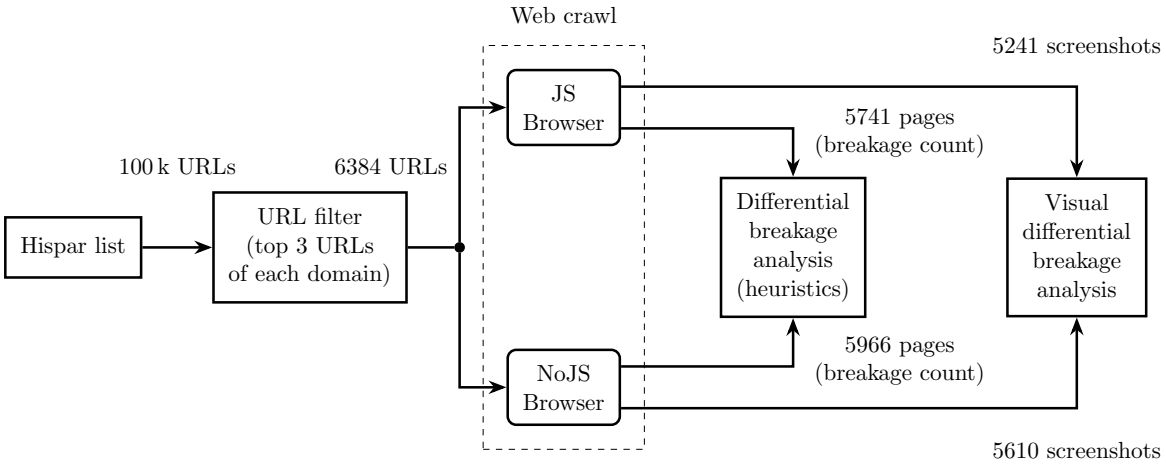


Figure III.4: Dataflow diagram

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

III.2.2 Collection of Website Categories

To discover and highlight breakage disparities between website categories, we adopted the classification from OpenDNS [198], 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 [199]) 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.

Table III.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 %)

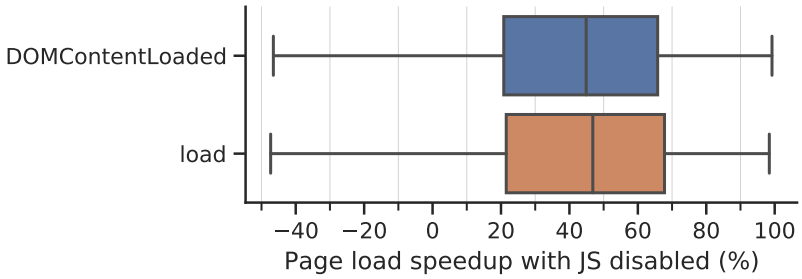


Figure III.5: Speedup when blocking JavaScript

III.3 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).

III.3.1 Dataset Description

Table III.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.

Inability to load and process some of the pages can be attributed to various factors, including region-based blocking [242], 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.

III.3.2 Effect on Page Load Time

Blocking JavaScript brings up a significant page load speedup for most pages, as shown in Figure III.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.

Table III.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

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.

III.3.3 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, `id`s, or classes that relate to the structure of a page.

Table III.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 [52]. 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.

III.3.4 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 :

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

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:

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

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

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

III.3.4.1 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.

III.3.4.2 Disparity Across Website Categories

Figure III.6 reports on the 90^{th} 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 A.1 from the appendix takes into account the whole page for comparison.

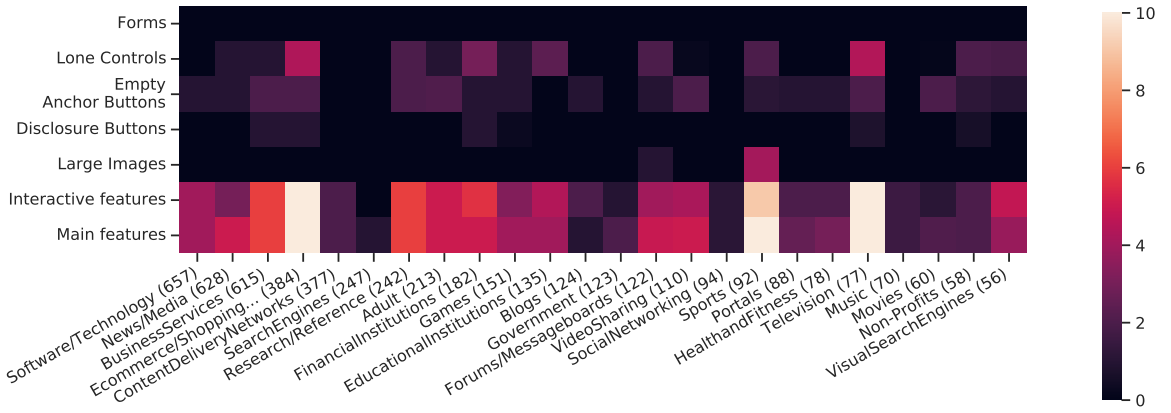


Figure III.6: Color represents the 90$^{\text{th}}$ percentile of differential breakage of visible elements in the *main section*. Lighter shades denote higher differential breakage. Only categories with more than 50 pages in the dataset are plotted, to improve readability. This highlights the disparity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

III.3.4.3 Dependency of Main Page Features on JavaScript

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

Feature Breakage Report Figure III.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 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 III.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 III.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 III.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 [236]), that standardizes

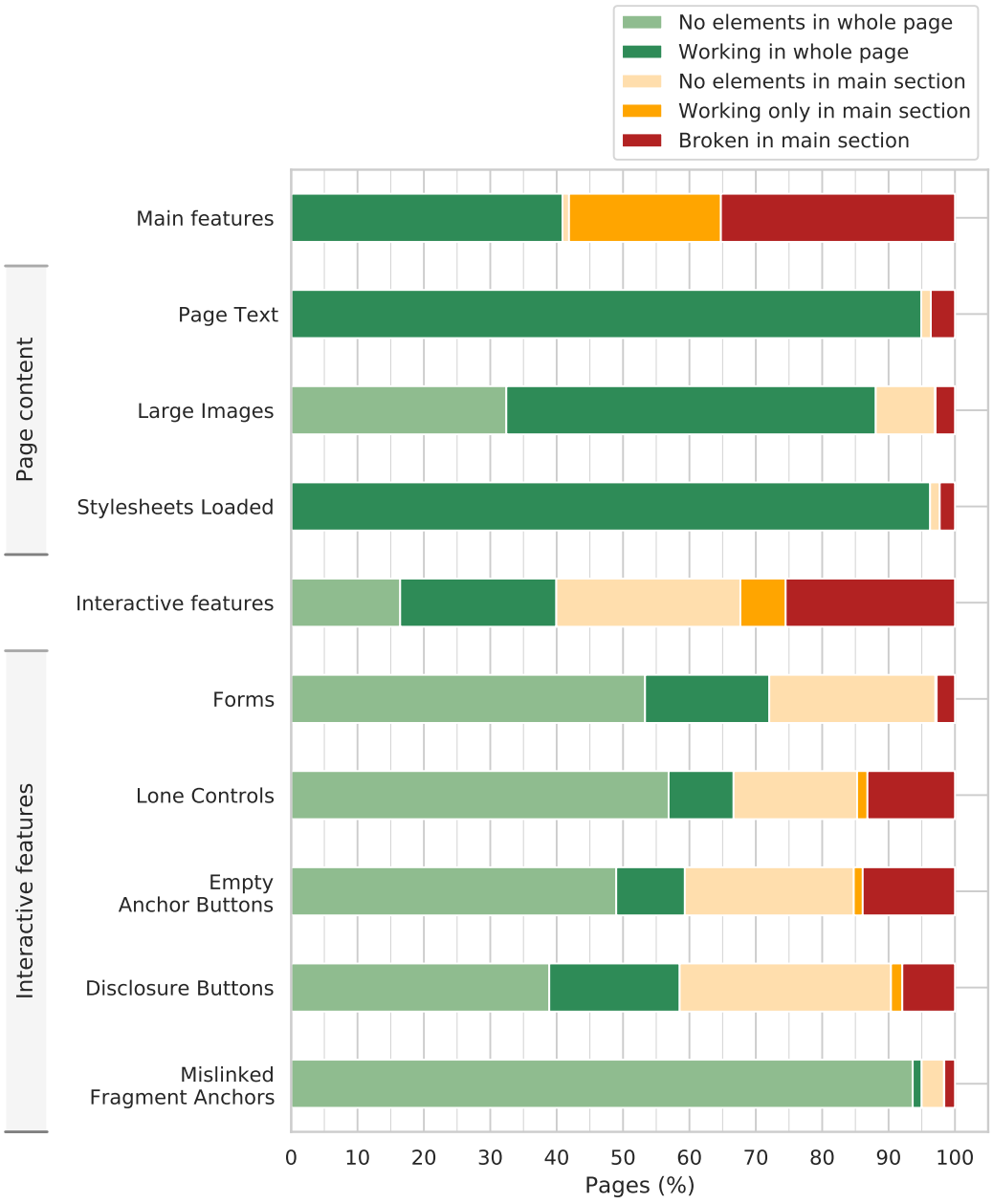


Figure III.7: Shares of pages for each page feature status; when present, a feature can either be working in the whole page, only in the main section or broken in the main section

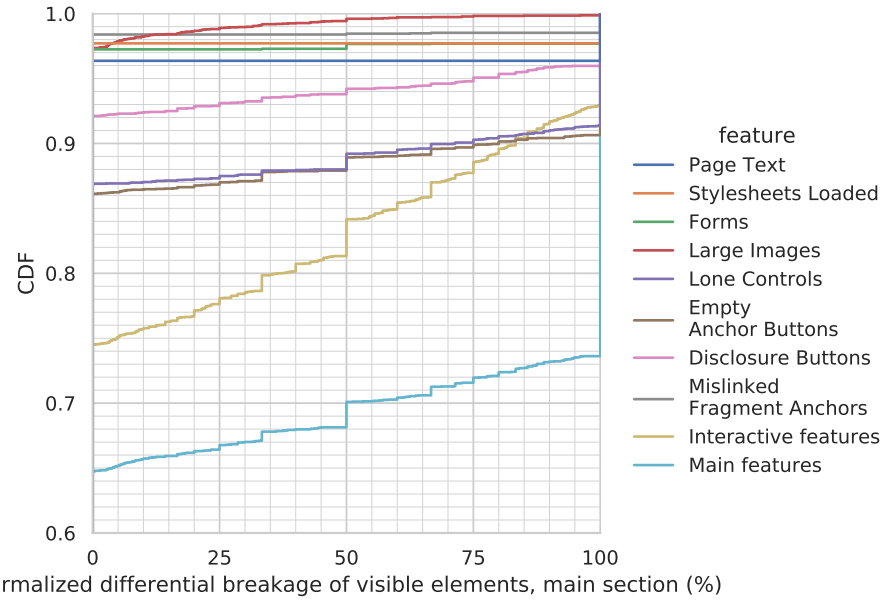


Figure III.8: Normalized differential breakage (DBRn) of visible elements of the *main section*; negative normalized differential breakage is not shown for readability

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.

III.3.5 Website Handling of Non-JavaScript Users

We investigate decisions made by websites that handle users with JavaScript disabled differently than users with JavaScript enabled.

III.3.5.1 Noscript Redirects

Some websites redirect users with JavaScript disabled to a different URL using a construct similar to `<noscript> <meta http-equiv="refresh" content="0; URL=…"></noscript>`, which redirects to the provided URL after 0 s only when JavaScript is disabled. This behavior remains uncommon: around 10 domains of our sample use this to redirect to a URL denoting the fact that JavaScript is disabled.

III.3.5.2 Fine-Grained `<noscript>` Fallbacks

This does not mean other websites do not specifically handle non-JavaScript users at all, instead, they provide fine-grained fallbacks using the `<noscript>` element, which is only interpreted by the browser when JavaScript is disabled.

Out of the 5,434 pages saved for the `[nojs]` crawl, 3,049 pages contain at least one `<noscript>` element, while the page with the highest count of them has 562 such elements. We extracted these `<noscript>` elements and classified their purposes, see Table III.3.

Table III.3: <noscript> purposes

<noscript> purpose	Share of pages having <noscript> tags featuring at least one element of the type (%)	Total element count
Tracking iframe	43.1	1413
Tracking pixel	28.1	1155
Image fallback	18.1	11008
Empty tag	17.2	638
Noscript warning	14.4	519
Style element	7.1	244
Async. style fallback	4.9	263
Next.js CSS nonce	4.1	132
Other	4.0	198
Other stylesheet	3.5	112
Anchor	3.3	363
Other iframe	1.3	54
Text only	0.4	82
Embedded video iframe	0.3	38
HTML comment	0.3	20
Noscript-dedicated style	0.2	7
Meta tag	0.2	8
Video tag	0.2	14
Advertisement iframe	0.1	9
Tracking social iframe	0.1	44
IE fallback	0.1	34

We classified the `<noscript>` elements by iteratively grouping them according to their children tags, manually inspecting them, and using the EasyPrivacy list [6] as a reference for tracking pixels. The "other" category contains elements that do not fall in any of the other categories, mostly because they are comprised of several children and would not fit in a single category.

Tracking Elements The dominant purpose—by far—is tracking, relying on tracking pixels and `<iframe>`s. Of course, much less data can be collected from the user's browser than with JavaScript, but these constructs are enough to at least count page views and set/read cookies, still enabling basic user tracking. All tracking `<iframe>`s from the dataset connect to a Google-owned domain: either `www.googletagmanager.com`, `*.fls.doubleclick.net` or `www.google-analytics.com`. Out of the 1,821 domains having at least one page from the dataset including a `<noscript>` tag, 47 % of them implement a noscript tracking `<iframe>` or a tracking pixel, of which 76 % use the `<noscript>` tag only for these.

Lazy-Loaded Image Fallbacks The second most common purpose is to provide image fallbacks for lazy-loaded images that require JavaScript. A few websites, including e.g. `www.bbc.co.uk`, `www.spiegel.de`, `www.heise.de` and `www.goal.com`, still set `loading="lazy"` on these noscript image fallbacks, even though it will be ignored by browsers to prevent tracking of the scrolling position when JavaScript is disabled. The large total count of these elements is explained by the fact that websites that bother implementing lazy-loading in JavaScript (and thus providing noscript fallbacks) are websites that make heavy use of images (such as news and shopping sites), and thus have a large number of noscript fallbacks per page.

Noscript Warning Message Some websites display a message inviting the user to enable JavaScript. Among these, some of them still provide other noscript fallbacks, thus not always actually requiring JavaScript.

```
<!-- Other common values for the media attribute are "none" or "only x",
  which are invalid media queries -->
<link rel="stylesheet" href="..." media="print"
  onload="this.onload=null;this.media='all'">

<link rel="preload" href="..." as="style"
  onload="this.onload=null;this.rel='stylesheet'">
```

Listing III.7: Asynchronous stylesheet loading

Async-Loaded Stylesheet Fallbacks Loading a CSS stylesheet blocks page rendering, thus delaying the first content paint. Some websites circumvent this by inlining critical CSS—i.e., CSS required within the initial viewport—and asynchronously loading other stylesheets when the initial page load is completed. This is usually implemented either using constructs similar to the ones presented in Listing III.7 or using a dedicated JavaScript library, such as loadCSS [122]. At the time of writing, there was no way of asynchronously loading stylesheets without JavaScript.

Noscript Dedicated Stylesheet Finally, a few websites embed a dedicated stylesheet, specifically intended to be used when JavaScript is disabled, to make the page usable. This stylesheet can be inlined or remote, often named `noscript` in this case. It is a short stylesheet that makes sure content that is normally hidden by default and then shown with JavaScript—e.g., as accordions or fade-in elements (elements hidden by default and whose appearance is animated)—is actually visible by default, by setting appropriate CSS properties for these elements—e.g., `opacity: 1` or `visibility: visible`.

III.4 Limitations

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

III.4.1 Measurement Framework Limitations

III.4.1.1 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.

III.4.1.2 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.

III.4.2 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 [51], 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.

III.5 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.

III.5.1 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.

III.5.2 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 [150], 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 [206].

Table III.4: 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

III.5.3 Benefits and Viability of Aggressive JavaScript Blocking

On top of the numerous privacy and security benefits introduced earlier, disabling JavaScript brings additional changes of different natures.

III.5.3.1 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 [187] of Firefox to classify each URL in real-time. The classification relies on Firefox Enhanced Tracking Protection and the built-in Disconnect lists [47] to indicate if a URL is either first or third party and whether it is involved in tracking or not.

Table III.4 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.

III.5.3.2 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.

III.5.3.3 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 [49].

Reducing the amount of transferred data has several benefits, including faster page loads for most pages (see subsection III.3.2), 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.

III.5.3.4 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 [246].

III.5.3.5 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 [129], NoScript [171] and the unmaintained uMatrix [128], 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.

III.5.4 Website Usage of JavaScript Features With No Fallback

III.5.4.1 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 [235]. Unlike the implementation from Listing III.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 [237], 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 [277] or Semantic UI [4].

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 [138], Vue.JS [278], AngularJS [118], or Svelte [11], 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 [120], 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 [193] (for React) or NuxtJS [7] (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.

III.5.4.2 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 [125, 88], 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) [127, 119].

III.6 Conclusion

In this contribution, 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.

In the next chapter, we will leverage the acquired knowledge of page breakage when JavaScript is disabled to design a set of repairs to common types of breakage, so that it is possible to keep JavaScript disabled on a larger share of webpages and still be able to access the wanted information.

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.

CHAPTER IV

Bridging the Gap Between the User and the Browser with User Browsing Intent

Previous works which investigated improving privacy through in-browser content blocking have focused on making the entirety of the page work, leaving some trackers unblocked. In this contribution, we claim that not all page features are required to work depending on the purpose of a visit. Hence, we introduce the concept of User Browsing Intent (UBI), which acknowledges that the user may visit a web page with different intents. For instance, a user may only want to read a newspaper article, including the related illustrations. Other examples include different expected visit frequencies: if the user only expects to visit the web page once or very occasionally, this allows the browser to delete all (including first-party) client-side storage related to the page at the end of its visit. On the opposite, if the user expects to browse the page regularly, the browser should avoid encumbering the website usage—possibly adjusted by the user's trust towards the website. In this contribution, we focus on the 'read-only' UBI: the user only wants to be able to read the page, with no interaction. Targeting this intent allows us to block more content, irrelevant in view of the UBI, enabling a much more robust tracking-content blocking strategy.

In light of this realization, we introduce and evaluate a tool blocking JavaScript by default, bringing significant privacy improvements, and applying a limited set of hand-crafted targeted repairs, with the aim of making more pages compliant with the 'read-only' UBI, i.e., readable by the user, even if interactive behaviors do not work. In the context of the 'read-only' UBI, the tool can be evaluated using only page screenshots, with a semi-manual classification of these, by comparing versions of pages where JavaScript is enabled, disabled, and disabled with our additional repairs.

We find that, using our tool, more than 62 % of web pages in our sample comply with the 'read-only' UBI if the user tolerates only minor information loss, and that more than 77 % of pages are compliant if they tolerate the loss of some non-central sections. In addition, we find that our targeted repairs make at least 27 % more pages compliant in both cases, making it viable to browse with JavaScript disabled by default with our repair WebExtension and to enable JavaScript only when needed.

Research Questions In this chapter, we address the following research questions:

RQ1 Does the concept of User Browsing Intent (UBI) help improve user privacy by blocking more tracking content?

RQ2 To what extent a limited set of targeted repairs improves the 'read-only' UBI?

Contributions Answering the above research questions brings the following contributions:

1. Introducing the concept of User Browsing Intent, allowing to block more tracking content
2. Sharing the implementation of a WebExtension to block JavaScript and automatically repair common client-side breakage
3. Evaluating this tool on a large-scale web crawl that includes landing and internal pages, with a semi-manual classification of page screenshots

IV.1 Introducing the User Browsing Intent

In this section, we present existing content blocking strategies and motivate our approach.

IV.1.1 Current State of Content Blocking Solutions

Currently deployed content-blocking solutions mostly rely on filter lists to define content that should be blocked [132, 188, 66]. Filter lists are an efficient way of blocking content, but they suffer from several issues. Being crowd-sourced, they can work well on popular websites, but may be incomplete for less popular websites, or in regions with fewer contributors [218]—e.g., in regions whose language have few speakers. They may also be lagging behind the current website behavior, especially if the website is regularly updated while not being heavily scrutinized. On top of this, current filter lists are limited in the content they can target [73] and thus content blocking tools are limited in what they can block. Indeed, JavaScript is now often bundled in a single file (or split into a few smaller script bundles), that contains both functional and tracking content. No content blocking tool is currently able to selectively target and block these tracking bundle sections in a generic manner, thus exposing the user to tracking, even when the required functional content is only used for page functionality the user would not be using during their visit. This has led to a plateau in new content blocking approaches, which try to complement filter lists with automated approaches [73] or try to automatically rewrite individual scripts [220], but this latter approach currently suffers from serious scalability issues regarding the distribution of the rewritten scripts, as noted by the authors.

IV.1.2 More Aggressive Blocking with the User Browsing Intent

We believe that, to further improve client-side web privacy through content blocking, the current gap between the user and the browser should be bridged so the browser can make informed content-blocking decisions on behalf of the user, and effectively play its role as a *user agent*. We introduce the concept of User Browsing Intent, to reflect the intent motivating a user to visit a given web

page. More specifically, we focus on the 'read-only' UBI in this contribution: the user only wants to read the content of a page, without interacting with it. For instance, the user wants to read a newspaper article, a blog post, or a company website. Thus, by acknowledging that the entirety of the page is not always required to satisfy the user, it becomes possible to block web content more aggressively, as long as the page stays compliant with the user's UBI. Other examples of UBIs include visiting a web page only once or very occasionally—allowing the browser to delete all (including first-party) client-side storage related to the page at the end of its visit—or, on the opposite, communicate to the browser that regular visits are expected and it should avoid encumbering the website usage—possibly adjusted by the user's trust towards the website. As not all webpages can be made compliant with the 'read-only' UBI, or when the user's UBI is not the 'read-only' UBI, this more aggressive content blocking strategy can be disabled in one click, thus bringing significant privacy and security benefits when possible, while also only minimally increasing browsing friction.

IV.2 Blocking JavaScript and Repairing Specific Breakage Cases

This section presents the design of our WebExtension, which blocks JavaScript and repairs some types of common breakage induced by this blocking. The WebExtension is tasked with the following:

1. Blocking JavaScript by injecting a Content Security Policy (CSP) header in the document HTTP response
2. Injecting WebExtension content scripts to repair common types of breakage resulting from the JavaScript blocking

IV.2.1 Blocking JavaScript with a WebExtension

To improve the privacy protection coverage, and in the context of the 'read-only' UBI, our WebExtension needs to block JavaScript completely. The only method available from a WebExtension is to inject a custom Content Security Policy (CSP), similarly to what the uBlock Origin [132] and NoScript [172] WebExtensions do. Thus, our WebExtension injects the following CSP response HTTP header, which forbids the browser from downloading and executing any JavaScript code, including inline scripts (scripts within `<noscript>` tags, in DOM1 event attributes such as `onclick`, or using the `javascript:` pseudo-protocol in `href` attributes): `Content-Security-Policy: script-src 'none'`. The WebExtension also blocks CSP reports to avoid sending extraneous reports to website owners and prevent additional privacy leaks.

In addition, HTML features a `<noscript>` tag, whose content is interpreted and rendered by the browser only if JavaScript is disabled *globally* in the browser. These tags are thus not rendered when JavaScript is simply forbidden using a CSP. Unlike uBlock Origin and NoScript, we do not inject `<noscript>` fallbacks—i.e., inject the contents of `<noscript>` tags into `<span>` tags so they are interpreted by the browser—as part of our JavaScript blocking process. Indeed, many of these `<noscript>` tags are actually used for tracking, by either adding a tracking pixel to the page or adding a tracking `<iframe>`, or show a banner asking the user to enable JavaScript. Some `<noscript>` tags are however selectively injected as part of one of our repairs, see Figure IV.2.2.

As the WebExtension is solely responsible for the blocking of JavaScript, it is able to disable the blocking when asked by the user.

IV.2.2 Scope of Targeted Repairs

As simply blocking JavaScript breaks some page features and may prevent content from being downloaded and displayed—even when caring only about the 'read-only' UBI—we introduce a set of targeted, in-browser repairs to fix common types of client-side breakage. We selected five types of breakage to investigate: lazy-loaded images, lazy-loaded stylesheets, fade-in elements, preloaders and `<noscript>` tags. We focused on these because they could be fixed by relying only on knowledge available client-side and we anticipated they would cover common breakage occurrences.

Lazy-Loaded Images

Lazy-loaded images are images whose loading is deferred after the initial page load. Even though the HTML standard now contains a native specification of lazy-loaded images—using the `loading` attribute set to `"lazy"` [266] on `<img>` tags—and though it is now supported by all evergreen browsers, many websites still implement the lazy-loading of images using JavaScript, often leveraging an open-source library, such as *lazysizes* [106].

The markup for a minimal lazy-loaded image looks like the following:

```
<img data-src="image.jpg">
```

A `data-*` attribute stores the image URL and a script detects when the image comes close to the viewport and then copies the attribute value to the `src` IDL attribute.

When JavaScript is disabled, lazy-loaded images implemented using JavaScript are not loaded.

Lazy-Loaded Stylesheets

Lazy-loaded stylesheets are stylesheets that are loaded or applied to the document using JavaScript.

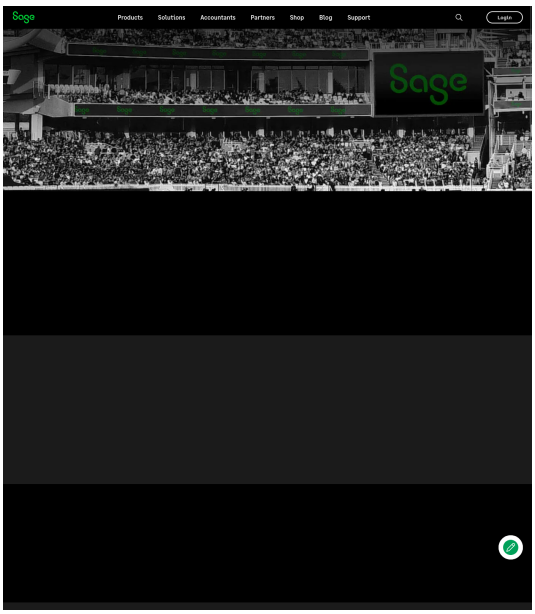
This can be useful as loading and parsing a stylesheet is synchronous: the browser will pause parsing the HTML document until the stylesheet is downloaded, parsed, and applied, to avoid a flash of unstyled content. Even though browsers implement a preload scanner to preemptively parse the rest of the HTML while synchronous scripts and stylesheets are loading—so they are able to download discovered resources, such as images or other scripts, in parallel—it may still be beneficial in some cases to lazy-load stylesheets.

A lazy-loaded stylesheet is often implemented as follows:

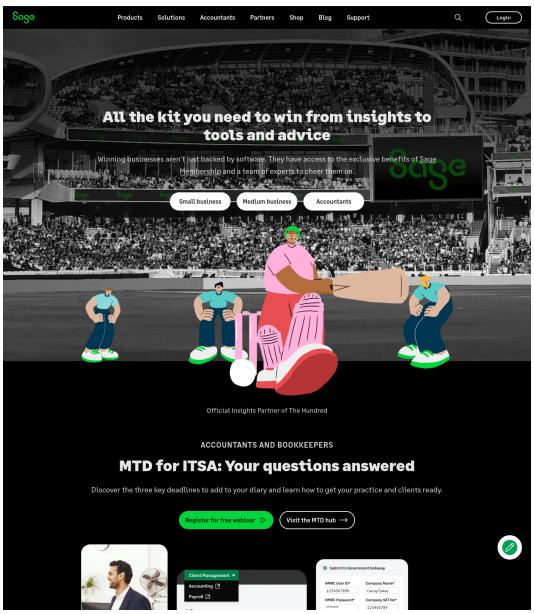
```
<link rel="preload" as="style" onload="this.rel='stylesheet'"
      href="style.css">
```

This asks the browser to asynchronously load the stylesheet with a high priority, then, when it is loaded, applies it to the page by changing the `rel` attribute.

When JavaScript is disabled, the lazy-loaded stylesheets are not loaded and page styles (or part thereof) are missing.



(a) Screenshot when JavaScript is disabled



(b) Screenshot when JavaScript is disabled, but fade-in elements are repaired

Figure IV.1: Screenshots of the homepage of www.sage.com, having fade-in elements, which make the content unreadable without JavaScript, fixed by our WebExtension

Fade-in Elements

Fade-in elements are elements initially hidden—usually by setting their opacity to zero—which are then made visible using JavaScript when the user scrolls the page and the elements enter the viewport. Figure IV.1 shows a page having fade-in elements with JavaScript disabled and with our repairs.

When JavaScript is disabled, fade-in elements are not shown on screen.

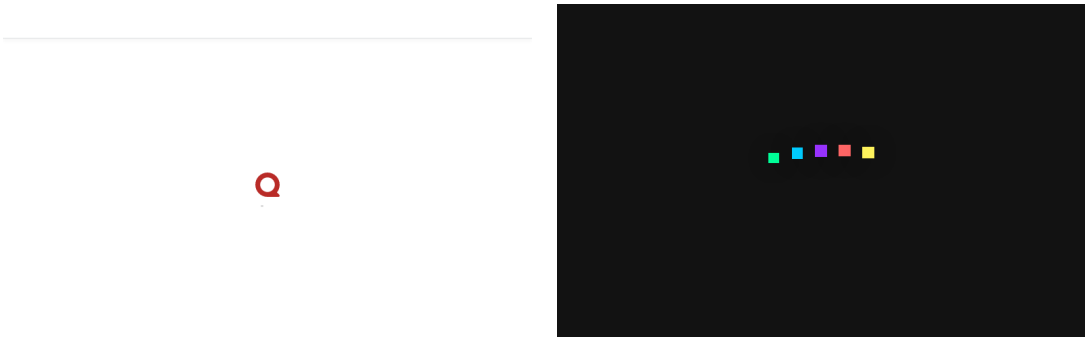
Preloaders

Preloaders are elements covering the entire viewport until the page is fully loaded. The preloader, which often features a spinner, is then hidden with JavaScript and unveils the page content it was concealing. A page normally contains at most one preloader. Figure IV.2 presents two examples of pages having preloaders.

When JavaScript is disabled, the page content is concealed by the preloader and thus cannot be accessed.

`<noscript>` Tags

A `<noscript>` tag [267] is a standard HTML tag whose content is only interpreted by the browser when JavaScript is *globally* disabled in the browser. It is often used to implement noscript fallbacks of elements whose behavior implementations require JavaScript or to show a banner asking the user to enable JavaScript because the website relies on it.



(a) Screenshot of the homepage of www.quora.com when JavaScript is disabled

(b) Screenshot of the homepage of giphy.com when JavaScript is disabled

Figure IV.2: Screenshots of pages having preloaders, which completely prevent accessing the page content without JavaScript

```
{
  "preloaders": [
    { "target": "body > #preloader" },
    { "target": "body > #splash" },
    {
      "target": "body > div.loader",
      "size": {
        "mustCoverViewport": true,
        "tolerance": { "value": 20, "unit": "px" }
      }
    }
  ]
}
```

Listing IV.1: Excerpt of the JSON repair configuration for 'preloaders'. The `target` property contains a CSS selector of elements to hide. The size of these can additionally be checked to make sure they cover the entire viewport.

IV.2.3 Repairing Breakage Induced by JavaScript Blocking

Leveraging the identification of these repairs, this section explains how our WebExtension proceeds to first detect the elements to repair before effectively repairing them, using our hand-crafted declarative configuration. The WebExtension injects content scripts—WebExtension scripts which run in the context of the web pages in the browser—that detect and repair each type of breakage. Each repair type has its own configuration, which makes it easy to update and extend the coverage of the repairs if needed. This configuration is built using our knowledge of the causes of breakage, on previous large-scale crawls, and on reading the documentation and source code of open-source libraries implementing the JavaScript mechanisms, such as *lazysizes* [106]; references to the libraries and archived versions of pages featuring broken elements are added to the configuration when possible. The configuration, as part of the WebExtension, is tested using continuous integration on a

selected list of web pages, to catch possible regressions introduced when modifying it. Excerpts of the configuration can be found in Listing IV.1, in Listing B.2 in the appendix and in the companion repository, see section .

IV.2.3.1 Detecting Elements to Repair

The detection of elements to repair leverages a combination of CSS selectors and dedicated checking mechanisms, relying on regular expressions and on reading the currently computed styles using `window.getComputedStyle()`. The exact criteria used to detect the elements to repair are defined in declarative JSON files which can easily be updated and could be pushed to users, similarly to filter lists. As soon as the page is loaded, the WebExtension searches the page DOM for elements needing to be repaired.

IV.2.3.2 Repairing Elements

Once the elements of a type of breakage have been detected, they can be repaired using the repair methods defined in the JSON configuration. Repairs usually involve applying some CSS properties—e.g., to force-show some hidden elements—or copying the values of some IDL attributes—e.g., to set the value of the `src` `<img>` attribute from another attribute containing the image's URL. As the exact cause of breakage varies within a single type of repair—e.g., not all fade-in elements work in the same way—each cause of breakage has its own repair method.

IV.3 Evaluating Repairs

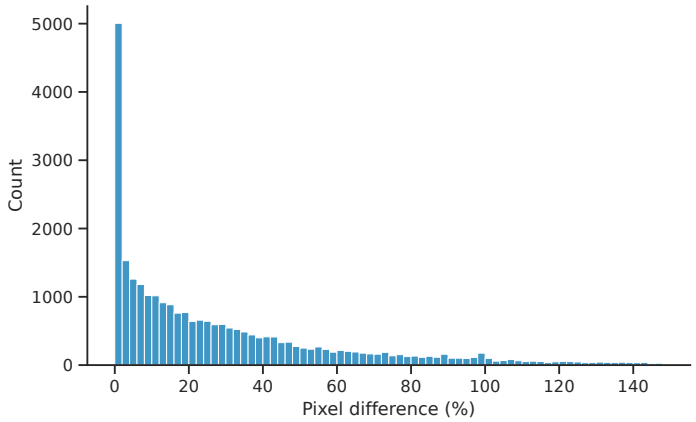
This section details how we evaluated our WebExtension in view of the 'read-only' UBI.

IV.3.1 Methodology

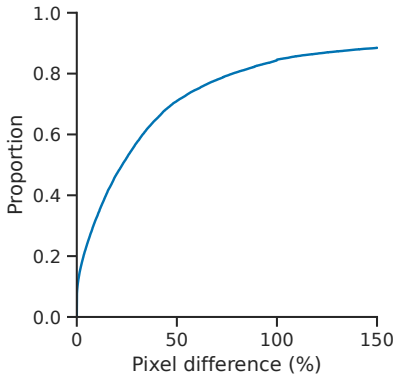
In this contribution, we focus on the 'read-only' UBI, that is, the user only wants to read the page and get a grasp of its content, but not interact with it. This means it is sufficient to visually compare page screenshots to evaluate whether the page is compliant with this particular UBI. We thus run a large-scale web crawl to gather the screenshots of 30,728 landing and internal pages in three different versions: with JavaScript enabled, disabled, and with JavaScript disabled with our repairs. We then semi-manually rated a 3,958-page sample following a visual differential analysis between these three versions.

IV.3.1.1 List of Visited Pages

For this evaluation, we needed to crawl a set of of landing and internal pages on popular and less popular websites. We selected the Tranco list, which provides a list of ranked-by-popularity domain names, and aims to stay stable by aggregating different domain-ranking sources [204]. To obtain a crawl sample containing popular websites as well as less popular ones, we kept the top 5,000 domains and added a random sample of 5,000 domains from the top 100,000 domains (excluding the already selected top 5,000 domains).



(a) Histogram of the pixel difference percentage between the JS and NoJS-UBI screenshots (truncated at 150 % for readability)



(b) CDF of the pixel difference percentage between the JS and NoJS-UBI screenshots (truncated at 150 % for readability)

Figure IV.3: Distribution of the pixel difference percentage between the JS and NoJS-UBI screenshots in our 30,728-page sample

As the Tranco list is a list of domain names, we first visited the landing page of the website by appending `http://` before the domain name. The crawl browser then visited this page and extracted a random sample of up to 4 URLs among the `<a>` tags linking to the same origin as the current page. As not all landing pages contain 4 links to the same origin, and because not all domain names from the Tranco list point to websites, we obtained 30,728 pages, among which 72.1 % were internal pages.

The crawl ran on a server we host, from June 28, 2022 to July 2, 2022, using the Tranco list from June 23, 2022.

IV.3.1.2 Data Collected

The crawler is built by automating Firefox using the WebDriver protocol. It spawns three browser instances:

- JS: the default configuration (with JavaScript enabled)
- NoJS: JavaScript is disabled using a purpose-built WebExtension which injects CSP headers (see subsection IV.2.1)
- NoJS-UBI: JavaScript is disabled in the same way, and our repair WebExtension is installed

JavaScript is always disabled using the aforementioned WebExtension, never globally in the browser.

The crawler visits each URL in the three browser instances at the same time and collects the following data:

- The count of repairs for each repair type in the NoJS-UBI browser

- The count of cookie annoyances (e.g., cookie overlays and cookie banners) in each browser, see below
- The HTTP requests sent by the page, including their tracking classification from Firefox Enhanced Tracking Protection in each browser, see below
- A full-page screenshot in each browser

To detect whether a page contains cookie annoyances—e.g., cookie banners or cookie overlays—we relied on the EasyList Cookie list [77], a crowd-sourced list containing CSS selectors matching such cookie annoyances. We built a WebExtension using this list of CSS selectors to count the number of cookie annoyances and send it to the crawler.

We also collect the list of HTTP requests sent by each page, including its tracking classification from Firefox Enhanced Tracking Protection [188], using another purpose-built WebExtension.

IV.3.1.3 Analysis Methods

The main part of the evaluation consists in rating whether the NoJS-UBI version complies to the 'read-only' UBI and, if it is, whether the page was already compliant without our repairs. In view of the 'read-only' UBI, comparing full-page screenshots of the page is sufficient to determine whether the page is compliant with this particular UBI. We thus opted for a hybrid approach: we first automatically compared full-page screenshots pixel-to-pixel, then put the pages into bins according to their pixel-to-pixel difference, randomly sampled the bins, and manually rated the page samples.

Automated Comparison We first automatically compared the NoJS-UBI full-page screenshots to the JS ones using a dedicated tool we built. It compares the two input screenshots pixel-to-pixel, while discarding the differences due to vertical content shift, and outputs an image highlighting the pixel differences, along with the pixel difference percentage. To discard differences due to vertical content shift, it first splits the images into 16 px-high strips and detects identical strips. Pixels from these identical strips—thus having vertically shifted—are then excluded from the pixel-to-pixel comparison. An example of the resulting difference image can be seen in Figure B.1, in the appendix. The pixel difference percentage between two images A and B is then computed as follows:

$$\text{pixel diff} = \frac{\#\text{different pixels}}{\#\text{shortest image pixels}}$$

where:

- #different pixels is the count of pixels different between image A and image B. If one of the image is taller than the other, all extra pixels are counted as being different, unless they were from a strip detected as shifted.
- #shortest image pixels is the pixel count of the image having the smallest pixel height.

The pixel difference percentage can thus be greater than 100 %, e.g., when image B is taller than image A and the pixel difference count is greater than the pixel count of image A. The pixel difference distribution between JS and NoJS-UBI full-page screenshots of our 30,728-page sample is shown in Figure IV.3.

Manual Labeling As relying only on this automated comparison would result in many false positives—pages classified as non-compliant with the 'read-only' UBI although they are—we opted for sampling the screenshot set and manually classifying them, based on the amount of information lost between the JS and the NoJS-UBI screenshots. As the amount of information loss is not linearly correlated to the pixel difference, we first put the screenshots into the following bins, according to their pixel difference percentage:

$$[0, 1, 3, 5, 10, 20, 35, 50, 65, 80, 100, 150, 200, +\infty]$$

We then sampled these bins, keeping 13 % of the screenshots, totaling 3,958 screenshots.

For each page, four images were displayed for the manual classification: the JS screenshot, the NoJS screenshot, the NoJS-UBI screenshot, and the color-coded difference image computed in the automated comparison stage, between the JS and NoJS-UBI screenshots, see Figure B.1 in the appendix. The order in which pages where shown was randomized, and the pixel difference percentage was not shown. Two grades were attributed: one for the information loss between the JS and NoJS-UBI screenshots, and the other for the potential repairs applied, between the NoJS and NoJS-UBI screenshots. One of the author thus labeled the 3,958 pages by following two different scales. The scale used for grading the information loss is the following:

0. The JS or the NoJS-UBI page is an error page due to the crawl itself, and would not appear during normal navigation—usually a bot detection page.
1. The NoJS-UBI page is utterly broken.
    - Examples: The page is blank, contains only skeleton placeholders, or the page content is concealed by a preloader that has not been hidden.
2. The NoJS-UBI page features major information loss compared to the JS page.
    - Examples: The central sections of the page are missing, important images are missing: e.g., product images in a product search result page, product screenshots are missing, some missing search results are missing, a video that is the main page focus is missing. The page language is incorrect. All page styles are missing.
3. A small number of non-central sections—sections considered as secondary by the page, based on its layout—are missing from the NoJS-UBI page compared to the JS page.
    - Examples: Some of the following sections are missing: homepage user reviews/Trustpilot evaluation, 'Trusted by' section, 'You may also like'/related posts section, embedded Twitter feed. Numerous generic stock images are missing.
4. Some minor information is lost between the JS page and the NoJS-UBI page.
    - Examples: The footer contains an `[email protected]` mention in footer (due to Cloudflare email address obfuscation [75]), some stock/generic images are missing with very few information, a small text banner is missing, redundant text—text appearing multiple times on the page—is missing.

5. The NoJS-UBI page feature no information loss compared to the JS page, taking into the account the exception list below.

The following elements are not considered as participating in information loss when missing:

- advertisement banners
- cookie banners/cookie overlays
- newsletter forms
- chatbot buttons
- feedback buttons
- 'secure' certification button
- social network sharing buttons
- custom web fonts
- go-to-top buttons
- donation request banners

The scale used for grading the automated repairs is the following:

0. The JS, NoJS, or the NoJS-UBI page is an error page due to the crawl itself, and would not appear during normal navigation—usually a bot detection page.
1. The repairs degraded the page, in view of the 'read-only' UBI.
2. The repairs did not modify the page.
3. The repairs fixed some—but not all—information loss breakage.
4. The repairs fixed all information loss breakage.

IV.3.2 Experimental Results

In this section, we report on the results obtained from the crawl and the semi-manual classification.

IV.3.2.1 Repair Frequency

Our repair WebExtension records the counts of elements repaired on each page. The shares of pages among the 30,728 pages crawled where the repairs have been applied to at least one element are detailed in Table IV.1. Note that the 'lazy-loaded images' repair may be over-reporting as it may try to repair unbroken elements. Figure IV.4 reports on the counts of elements which have had repairs applied to them, per page. It highlights that the targeted elements do not have the same counts of occurrences on a page: a page has at most one preloader but can have many lazy-loaded images. Correlation between repairs having modified at least one element is shown in Figure IV.6. Positive correlation between 'lazy-loaded images' and 'lazy-loaded stylesheets' can be attributed to website developers trying to defer the loading of as much content as possible. Note also that not all `<noscript>` tags are injected by our repairs, on purpose, so websites may still be adding noscript fallbacks to lazy-loaded images even though this correlation is not highlighted by the matrix.

Table IV.1: Shares of pages among the 30,728 pages where repairs have been applied to at least one element

	Share of internal pages (%)	Share of homepages (%)	Total share of pages (%)
Lazy-loaded images	26.12	28.14	26.58
Fade-in elements	15.62	17.37	16.01
Lazy-loaded stylesheets	8.44	7.88	8.32
Noscript tags	8.31	9.74	8.63
Preloaders	0.80	0.75	0.79
Any	44.95	48.96	45.86

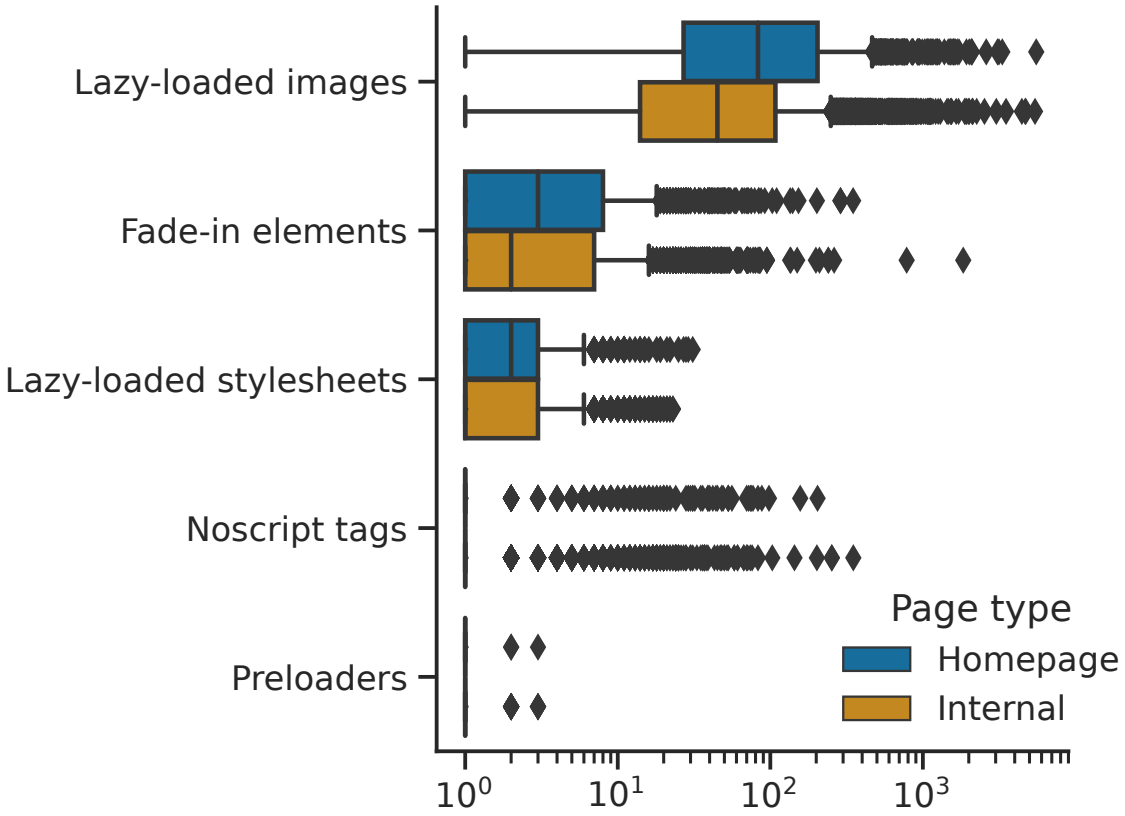


Figure IV.4: Tukey boxplots of the counts of repaired elements per page, when at least one element has been repaired: when a page contains a lazy-loaded image, it usually contains many of them, unlike preloaders, which usually appear only once on a page when present.

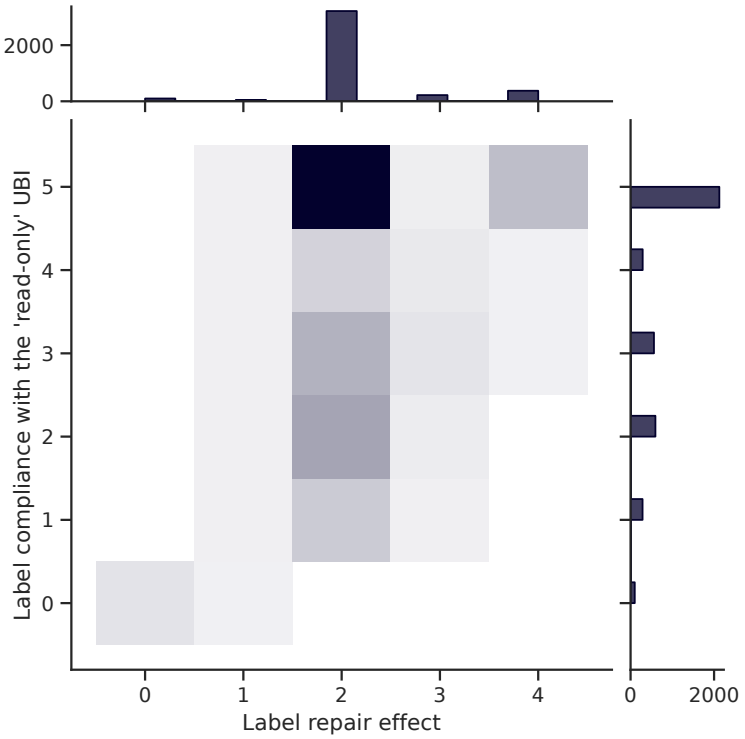


Figure IV.5: Bivariate plot of the manual labeling; the grades are defined in subsubsection IV.3.1.3

Table IV.2: Shares of pages compliant with the ‘read-only’ UBI and the indicated tolerance

	NoJS (%)	NoJS-UBI (%)	Improved by repairs by (%)
Minor information loss (4) + (5)	49.04	62.62	27.70
Loss of some non-central sections (3) + (4) + (5)	60.54	77.14	27.43

IV.3.2.2 Validation of Targeted Repairs

Using the semi-manual rating methodology described before, we found that, with our repairs, more than 62 % of pages were compliant with the ‘read-only’ UBI when tolerating only minor information loss; if the user also tolerates losing some non-central sections, more than 77 % of pages of our sample comply with the ‘read-only’ UBI, as shown in Table IV.2. The results of the manual labeling are shown in Figure IV.5. The repairs degraded the page compared to the NoJS version (without the repairs) for less than 1.27 % of pages. This is most often caused by the ‘fade-in elements’ repair, which may incorrectly force-show some elements meant to stay hidden, e.g., dropdown menu lists.

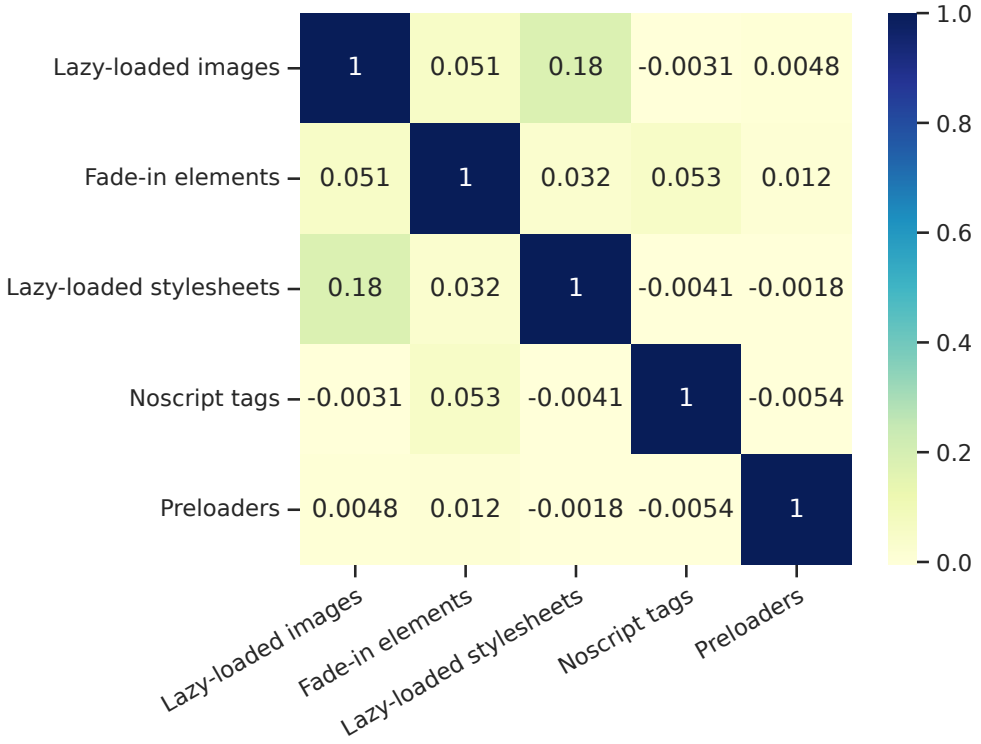


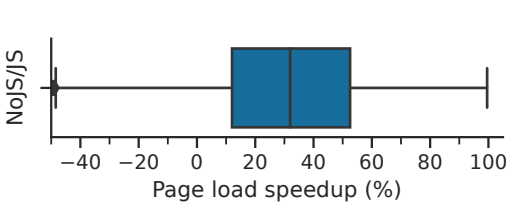
Figure IV.6: Correlation matrix of repairs modifying at least one element

IV.3.2.3 Benefits

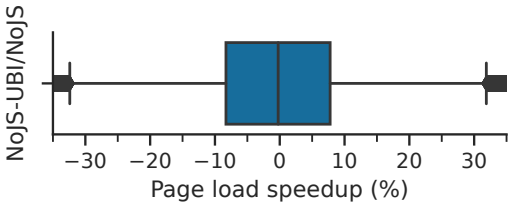
Browsing without JavaScript, when made easier with our repair WebExtension, has multiple benefits, including improving client-side privacy and security, and potentially improving the browsing comfort.

Privacy On top of significantly reducing the browser attack surface, having JavaScript disabled significantly improves user privacy. Indeed, as shown in Table IV.3, the number of third-party tracking requests, as classified by Firefox Enhanced Tracking Protection and matching the Firefox ETP `any_basic_tracking` flag [190], is significantly smaller with JavaScript disabled. On each individual page, the number of third-party tracking requests was reduced by 97.7 % on average, with a median of 100.0 % and a standard deviation of 18.82. Remaining, unblocked trackers are mostly tracking pixels—tiny images used to count page hits and collect some information about the user—and tracking `<iframe>`s. Both do not depend on JavaScript to be sent, but the amount of data they are able to collect is greatly reduced when JavaScript is disabled. In addition, our WebExtension can be used in conjunction with existing filter-list-backed blocking tools, so that these requests are also prevented.

Even though the set of tracking requests detected as such should already be blocked by content blockers—as Firefox Enhanced Tracking Protection [188] relies on the Disconnect list for its tracking classification—blocking JavaScript helps protect the user privacy even further. It indeed also



(a) Speedup between the JS and NoJS pages (truncated for readability): disabling JavaScript very often significantly reduce page load times.



(b) Page load time difference between the NoJS and NoJS-UBI pages (truncated for readability): our repairs do not slow down the page loads.

Figure IV.7: Tukey boxplots of the load time differences (measured with the 'load' event) when disabling JavaScript and when applying our repairs to the NoJS pages, in our 30,728-page sample

prevents the execution of trackers unknown to filter lists and of trackers that cannot be targeted with existing content-blocking solutions.

Browsing Comfort In addition to bringing privacy benefits, disabling JavaScript can also improve the browsing comfort, especially page load times and the reduction of page annoyances. As the Figure IV.7a shows, disabling JavaScript brings a median page load time speedup—measured with the 'load' event—of more than 31 %, while only very rarely slowing down the loading.

Moreover, many websites rely on JavaScript to show cookie banners and cookie overlays to the user, asking for content before processing the user's personal data. Based on the EasyList Cookie list, we found that 35.19 % of pages from our sample contained a cookie banner or overlay. Among those, 53.27 % of them were not shown when JavaScript was disabled.

IV.4 Discussion and Limitations

In this section, we discuss this contribution's limitations and possible usage.

IV.4.1 Public Pages

In this contribution, we focused on testing our repair WebExtension on public pages, including landing and internal pages, but we did not evaluate it on pages requiring authentication to be accessed. However, as these pages are likely to involve user interaction, the 'read-only' UBI may be less relevant for those pages.

IV.4.2 Usage and Privacy–Usability Trade-off

We found that a large part of the pages are compliant with the 'read-only' UBI with our repairs applied. As not all pages are compliant, the usage workflow would be the following: the browser loads the pages with JavaScript disabled by default, with our repair WebExtension, and then, if the page is not compliant with the 'read-only' UBI or if the user's User Browsing Intent is different, the user would enable JavaScript for this page in one click. This brings a significant privacy and security improvement on a large share of pages, while involving only a single action on the others. Usage

Table IV.3: Counts of third-party tracking requests with JavaScript enabled and disabled, with our repairs. Some Script and XHR requests are still present without JavaScript because JSON objects loaded with `<script>` tags and `<link>` preloads are classified as such.

Request type	JS	NoJS-UBI
Script	229970	8
Image	195154	1588
XHR	157193	7
Iframe	50146	393
Beacon	19400	3
Stylesheet	1997	99
Font	381	7
WebSocket	348	0
Media	334	3
Image set	174	361
Other	44	33
Object	29	0
CSP report	2	1

friction could even be decreased by automatically detecting the change of the user's UBI based on their behavior on the page (clicking on a button or typing characters on the keyboard), which we leave to future work. Whether JavaScript is enabled can also be remembered by the browser for each page, avoiding having to enable JavaScript each time the user repeatedly visits a page requiring it.

IV.4.3 Unfixed Breakage and Maintenance

Breakage unfixed by our repair WebExtension we observed in our crawl can be split into two categories: breakage that is fixable client-side and breakage that is not. To be fixable client-side, the missing content must either (1) be present in the HTML document—e.g., 'fade-in elements' or `<noscript>` tags—but hidden, (2) referenced by a URL extractable from the document—e.g., 'lazy-loaded images' or 'lazy-loaded stylesheets'—or (3) be standard enough to be inferred from some content, e.g., if a client-side library uses some particular format of URLs, this external knowledge can be leveraged to manually develop a repair. On the contrary, some types of breakage cannot be fixed client-side without knowing the server-side infrastructure, e.g., XHR requests leveraging some complex server API cannot be replaced.

If new occurrences of one of the breakage types covered by our WebExtension are discovered in the future, the WebExtension configuration can easily be updated and could be distributed separately, as filter lists currently are. Furthermore, the number of libraries used by websites and relevant to our repairs—e.g., *lazysizes* [106]—is small enough to be easily monitored for change, and these libraries rarely undergo major changes, making the maintenance burden very manageable.

IV.5 Conclusion

In this contribution, we introduced the concept of User Browsing Intent and focused on the 'read-only' UBI to propose a solution blocking JavaScript by default, while providing a limited set of targeted targets, aimed at repairing common types of breakages fixable client-side. We evaluated our repair WebExtension on a large page sample, semi-manually labeling the page conformance with the 'read-only' UBI. We found that 77 % of pages are compliant with this particular UBI if the user tolerates the loss of some non-central sections, and that our repair WebExtension increases the number of compliant pages by more than 27 %. Future work should investigate improving user privacy in the context of other User Browsing Intents.

Availability

We make available our repair WebExtension, including the continuous integration testing configuration. We also make available the entire crawl infrastructure, with a reusable Rust crawl framework, along with our data analysis notebook. These can be found in the following repository:
https://archive.softwareheritage.org/browse/origin/https://gitlab.inria.fr/Spirals/privacy-ubi-artifact.

CHAPTER V

Reducing Interface Components Dependency on JavaScript Server-Side

Existing development tooling and previous works have investigated the automated removal of JavaScript code unused in a page. However, they are only able to remove dead code, i.e., code that is shipped to the browser but not executed.

In this chapter, which extends our paper published in WWW Companion 2022 [110], we propose an automated server-side solution to replace common interface components with alternatives not relying on JavaScript for their interactivity, making it possible to remove the component JavaScript library. We show that most User Interface (UI) components provided by popular frameworks, such as Bootstrap [236], 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 CSP, 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 [72]. The contributions covered by this chapter 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
5. Manually validating these noscript alternatives on a corpus of 100 webpages

V.1 Rewriting HTML with Noscript Alternatives

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

V.1.1 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.

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* [215, 90], 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 V.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`.

```
<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>
```

Listing V.1: Noscript alternative of a dropdown button

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 [255] and derived the set of pseudo-classes that can be used to access the state of HTML elements and other mechanisms without JS. Table V.1 reports on the mechanisms

Table V.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

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.

V.1.2 Rewriting UI Components with JSRehab

V.1.2.1 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 V.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.

V.1.2.2 Generating noscript alternatives

Even though the *checkbox hack* has been well-known for more than a decade [215], 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 hand-writing the required HTML and CSS. Moreover, this first point particularly stands out when compared to interface component frameworks—such as Bootstrap [236] or ZURB Foundation [276]—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 V.1.2, we succeeded

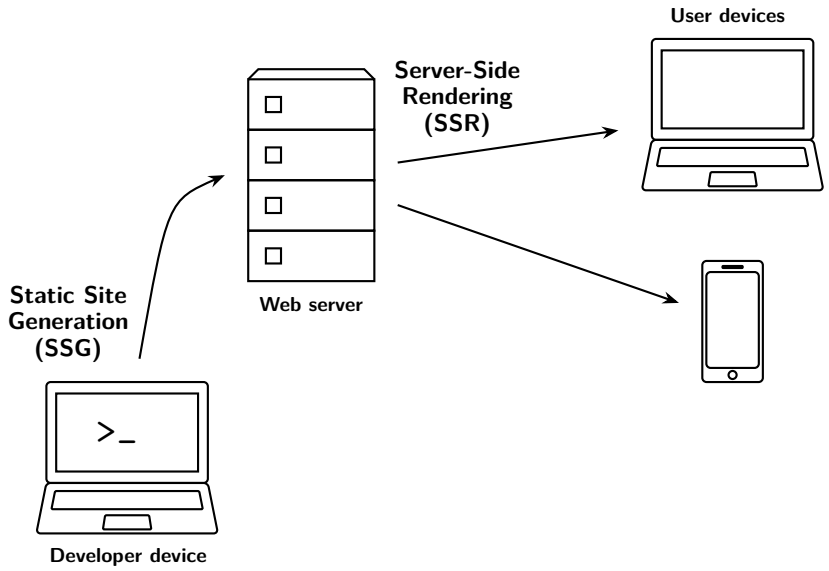


Figure V.1: Intended deployment setup of JSRehab: it is meant to be used in an SSG or in an SSR context

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 V.2 and the JSRehab plugin repository[1] contains detailed documentation about each component.

We opted to use an HTML preprocessor called PostHTML [8] and create our own JSRehab plugin to carry out the transformation, as pictured in Figure V.2. By using existing transformation tooling, we also benefit from its integration into the web ecosystem, including bundlers, such as Webpack [13] and rollup.js [9], and web server frameworks, such as Express [228]. We can also generate noscript alternatives in both SSG and SSR contexts (see Figure V.1), by injecting them only once in the former or whenever the page is rendered in the latter case.

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

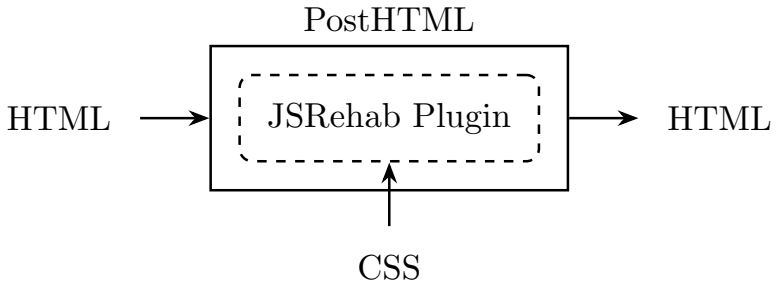


Figure V.2: JSRehab is built as a plugin for the existing PostHTML HTML preprocessor; it processes input HTML, along with optional additional stylesheets, to produce HTML with noscript alternatives

Table V.2: Bootstrap components having a built-in JavaScript behavior [232];

Bootstrap component (latest version)	noscript alternative	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*

V.1.3 HTML & CSS Limitations

Crafting noscript alternatives requires particular care due to various default behaviors and browser limitations, that we highlight in this section.

V.1.3.1 CSS Limitations

Some of these limitations are inherent to CSS, while others could be fixed by browsers supporting a few more pseudo-classes. In particular, when crafting the noscript alternatives, it is often needed to select elements based on the state of other elements, e.g., to make a dropdown menu appear when its hidden checkbox is checked, as in Listing V.1. However, until support for `:has()` [256] becomes widespread, CSS only makes it possible to select elements *following* other elements in tree order [262], imposing that elements whose state should be accessed must be placed *before* all elements reading their state. This constraint is taken into account in our rewriting rules, but may exhibit some limitations in particular cases.

Other CSS limitations refer to missing features, preventing the noscript alternatives from accessing some information, such as whether elements are currently within the viewport—e.g., tooltips cannot take the viewport into account to reposition themselves and make sure they are completely visible—and it is currently impossible to implement a scrollspy component, a table of contents highlighting the current section being read.

V.1.3.2 Scroll & Focus Limitations

Noscript alternatives make extensive use of `<label>`s associated with hidden checkboxes or radio buttons. Current specifications indicate that `<label>`s are not focusable elements, hence their associated input must be hidden in a way that still makes them focusable, using `opacity: 0` and a `position` value that removes them from the normal document flow. When activating the `<label>`, the associated input will be activated accordingly, but the input states (e.g., `:checked`, `:focus`, `:active`) are not propagated back to the associated `<label>` [17], meaning the `<label>` styling of these states must be handled by the HTML preprocessor by generating CSS similar to Listing V.2's. The preprocessor thus needs to be provided with the page stylesheets to be able to copy the style that would be applied to original components to the generated `<label>`s. This is supported by our JSRehab plugin, as shown in Figure V.2.

```
<style>
  #chkbox0:active + label,
  #chkbox0:focus + label {
    /* Dropdown button styling when focused */
  }
  #chkbox0:active + label {
    /* Dropdown button styling when activated */
  }
</style>
```

Listing V.2: Styling a `<label>` depending on a checkbox state

In addition, browsers scroll the page when an element receives focus, so that the element is within the viewport. This can cause undesirable scrolls when an out-of-viewport input receives focus, if the input has been absolutely positioned to remove it from the normal document flow, as is usually advised in "checkbox hack" guides [90]. This can fortunately be mitigated by using `position: fixed` instead, as in Listing V.1, which will keep the input in the viewport at all times, preventing unwanted scrolls.

Finally, no HTML or CSS features allow to conditionally prevent scrolling of the page background when a modal is open, nor to trap the focus within a modal or a dropdown menu.

V.1.4 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 [257] which do not specify implementation details, only high-level requirements, such as the Success Criterion 2.1.1 Keyboard [258], indicating only that the page must be operable with a keyboard with no time-sensitive input.

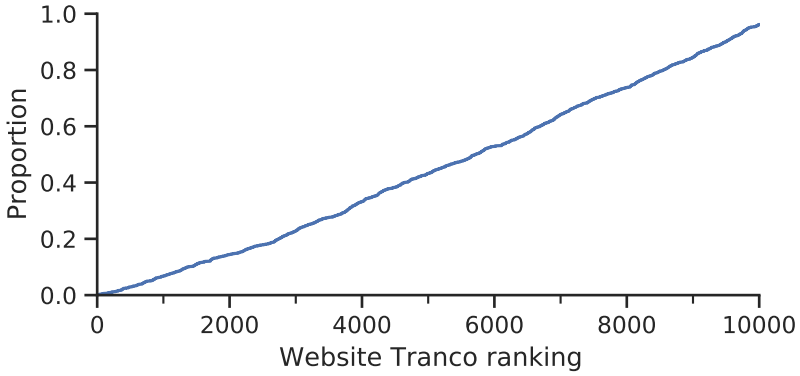


Figure V.3: CDF of Bootstrap's JS usage according to website ranking; Bootstrap's JS usage distribution is uniform, except for the very best ranked websites

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 [253]—such as `aria-checked`—which could not be toggled without JS.

V.2 Validating the Noscript Alternatives

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

V.2.1 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 [203] 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, as can be seen in Figure V.3, 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, as can be observed on Figure V.4, that the collapse, dropdown, and modal components are by far the most common—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.

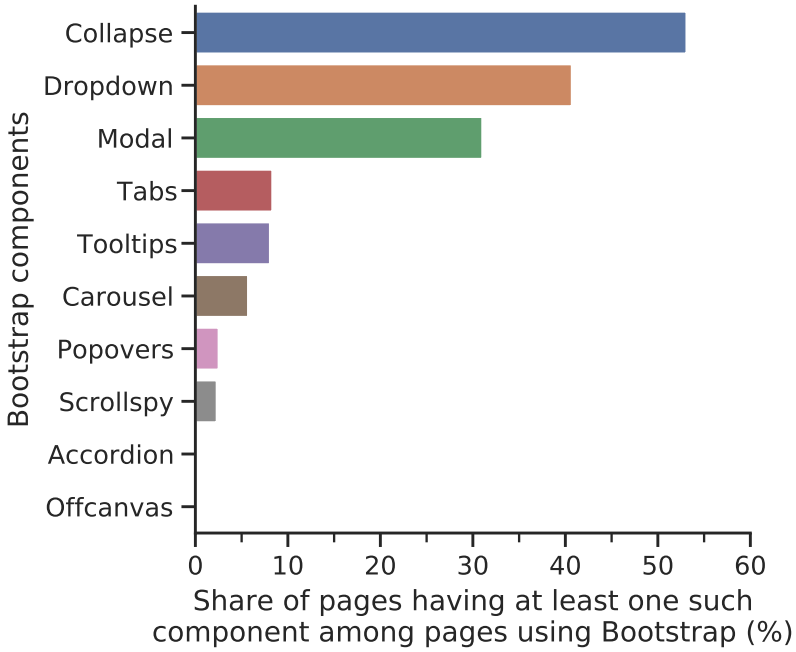


Figure V.4: Popularity of Bootstrap's components; accordion and offcanvas components have only been recently introduced with Bootstrap 5, and are not widely used

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.

V.2.2 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 V.5.

V.2.2.1 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).

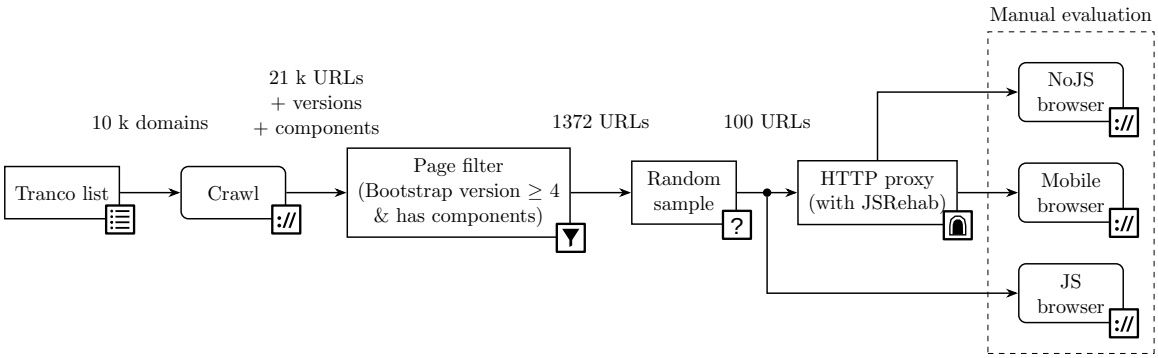


Figure V.5: Dataflow diagram of our empirical evaluation setup

Table V.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

V.2.2.2 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 V.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.

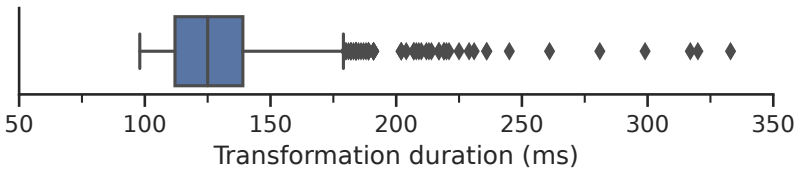


Figure V.6: Distribution of transformation durations on the corpus of tested pages

V.2.3 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.

V.3 Results

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

V.3.1 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 V.7, 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 V.6. The median rewriting delay is lower than 125 ms on

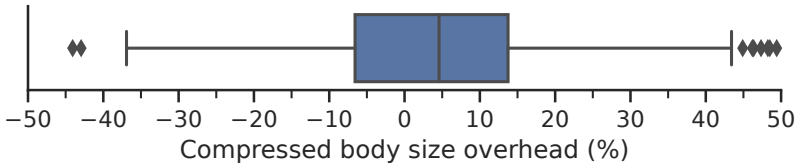


Figure V.7: Distribution of the compressed body size overhead on the set of tested pages (some outliers are omitted to improve readability)

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.

V.3.2 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 V.4.

Pages not being fully testable include error pages, which were not the intended pages, Single Page Applications (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 [231], 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.

V.3.3 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 [227].

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

Table V.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

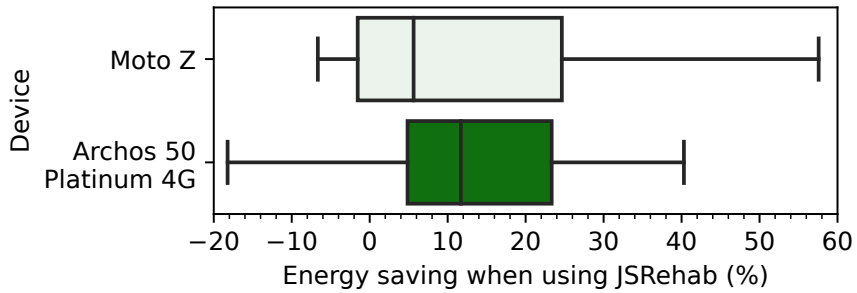


Figure V.8: Energy savings of mobile devices when loading web pages without Bootstrap's JS and rewritten with JSRehab; outliers are excluded for readability

not possible on every page, we focused on the top 31 pages that include a file `bootstrap.min.js`, according to PublicWWW [19]. 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 V.8, 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.

V.4 Discussion

V.4.1 Expected Benefits

V.4.1.1 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 59.5 % of the crawled domains were using outdated and vulnerable versions of Bootstrap, as the Figure V.9 highlights. This lack of security fixes can open users to security problems.

V.4.1.2 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 [148, 133], 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 [72] 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 [72], as component interactivity scripts would be considered as *essential*.

V.4.2 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 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; an example of configuration for Webpack can be seen in Listing C.1, in the appendix.

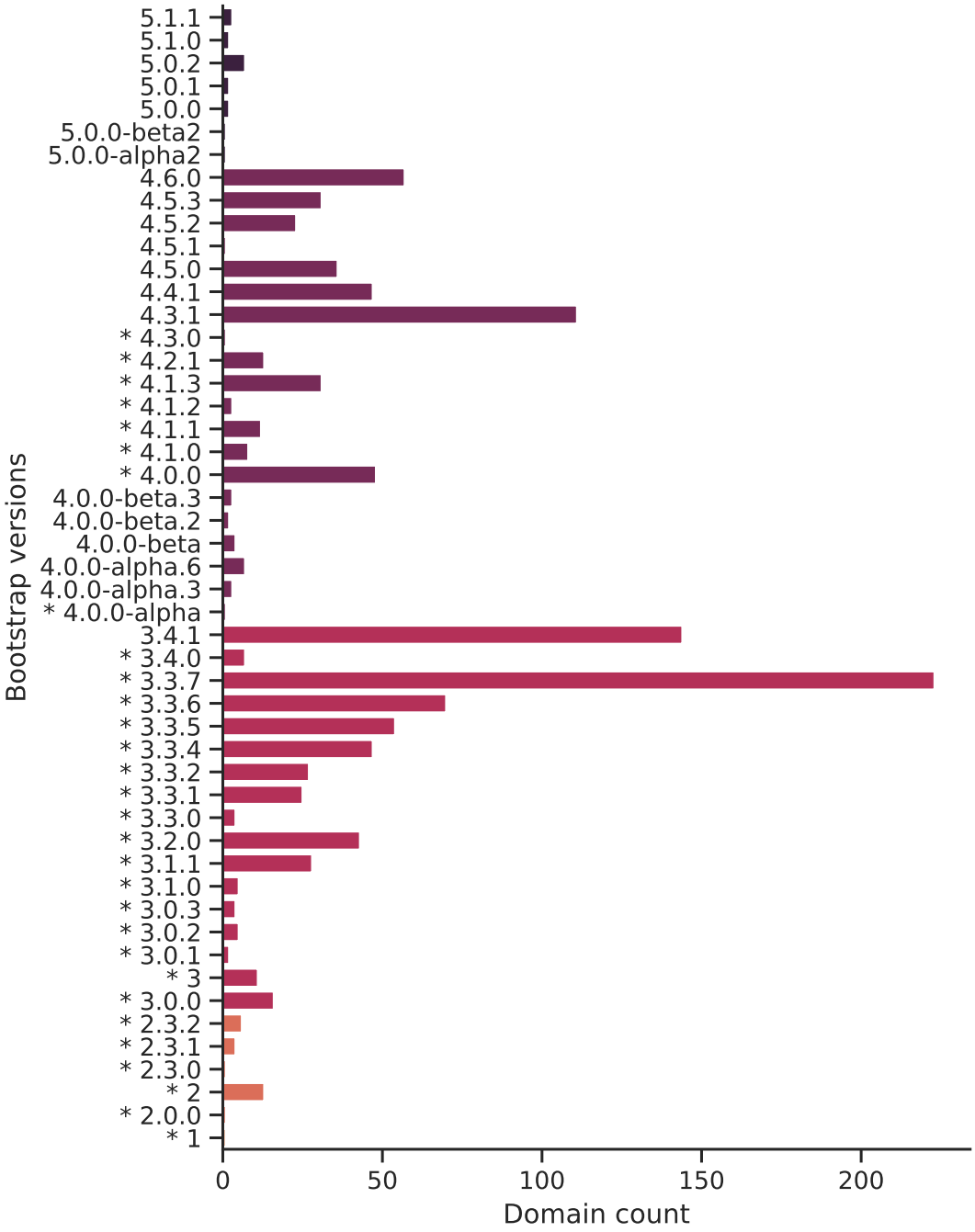


Figure V.9: Popularity of Bootstrap's JS versions on the crawled pages; versions marked with a star (∗) are known to be vulnerable to at least one XSS vulnerability as listed by [168]

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.

V.4.3 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 [276, 10], 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 support `:has()` [256] or to implement new pseudo-classes such as `:in-viewport` [12].

V.5 Conclusion

In this contribution, 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.

CHAPTER VI

Detecting and Blocking Individual JavaScript Functions for Privacy

Filter lists have been the subject of many previous works. However, they are still unable to block only parts of external scripts. This raises concerns as script bundling—i.e., using a JavaScript bundler to resolve the dependencies between authored scripts and automatically generate a single file containing both first-party code and third-party libraries—has been the development best-practice for many years. It indeed would not work to add these script bundles to filter lists, since they also contain functional code, potentially resulting in severe page breakage. In the following, we call 'bundles' scripts resulting from the bundling of several scripts and containing functional code. They may also contain bundled tracking code.

A more fine-grained blocking solution is therefore needed, able to specifically target and block parts of external scripts. Hence, we introduce a function signature technique that makes it possible to detect tracking functions from known tracking libraries in other scripts, especially first-party script bundles. Our function signatures are built on the function Abstract Syntax Tree (AST), where local identifiers—e.g., local variable names—are erased, to increase signature robustness while keeping chances of false positives low. Working at function level and applying this erasure transformation contribute to make the signatures robust: the tracking functions may appear at any location in the bundle and the signatures are location-independent. Relying on a well-defined and understood syntactic unit—functions—makes the signatures easy to reason about and limit false positives when classifying functions that should be blocked.

Using our function signature technique with a large-scale web crawl, we were able to detect yet-unknown scripts bundling tracking functions along with functional code. We find that at least 4.37 % of unique scripts contain such tracking functions, in scripts unknown to filter lists, while 22.7 % of visited domains were loading such a script on their homepage. In light of these measurements, we propose a hybrid blocking strategy to block specific functions in downloaded scripts in the browser, along with a list-inspired approach for scripts of significant size, thus providing a robust, future-proof and efficient solution.

Research Questions In this chapter, we address the following research questions:

RQ1 How often are tracking functions from known tracking libraries bundled in other scripts?

RQ2 How to design function signatures to detect these tracking functions?

RQ3 How to leverage these function signatures for in-browser blocking?

Contributions Answering the above research questions brings the following contributions:

1. Introducing a robust function signature generation technique
2. Quantifying how much tracking functions from known tracking libraries are bundled in other scripts
3. Proposing a hybrid in-browser blocking strategy leveraging the generated tracking functions signatures

VI.1 Motivation

VI.1.1 Blocking and Usability

In order to improve their online privacy, a lot of users are turning to ad blocking solutions to limit the number of trackers they encounter when they browse the web. By simply using an extension or installing a browser with this type of protection built-in, web users become free from visible ads and hidden trackers. In 2021, it was estimated that an average of 37 % of the Internet population were using ad blocking solutions with this number growing each year [93].

While very convenient for online browsing, ad blockers may become a nuisance when visiting certain webpages. Because they block scripts completely without considering their content, ad blockers may end up blocking useful code that is needed for the webpage to behave properly. This breakage is particularly exacerbated with modern coding practices where developers bundle different scripts on a page in a single file. In that case, if an ad blocker wants to protect the user, it would have to block the bundle of scripts at the risk of breaking the other libraries that may also be present in it. As a remedy to this problem, special 'unbreak' filter lists [21] are being maintained to index sites where some tracking scripts should not be blocked, but the existence of such a list shows the limit of current approaches when it comes to ad blocking. There is a need for a more fine-grained solution that blocks specific parts of a script and not its entire content.

VI.1.2 Related Work

In the literature, Chen et al. performed dynamic analysis of scripts running in the browser [73]. Their approach generates signatures of JavaScript behaviors based on the event loop of the JavaScript engine. They use known tracking scripts to build a database of signatures identifying tracking behavior. They then use these behavioral signatures to detect tracking behavior in other scripts on the web. While their approach is successful at identifying tracking in obfuscated and bundled scripts, their signatures are based on operations that are so low-level that it becomes hard to map the signatures to actual code. Moreover, their study is mostly focused on detection, and it would be difficult to leverage these signatures for in-browser blocking, since they rely on script *behavior* and in particular on detecting the tracking behavior, which, therefore, has already happened when they are able to detect it as such.

Smith et al. developed SugarCoat [220], which aims to create safe alternatives of tracking scripts that cannot be blocked. After analyzing the behavior of a script in the browser, SugarCoat automatically injects specially crafted JavaScript statements to intercept accesses to sensitive sources like `localStorage` or cookies, before restoring the original API implementation after the privacy-sensitive accesses originating from that script, in case that API is used by other scripts. They use existing exceptions in filter lists, maintained by the community, to focus their efforts on a smaller and well-identified list of scripts. It also raises serious scalability and legal concerns about the distribution of these scripts: the entire modified scripts (which may be as large as a few megabytes [67]) need to be distributed with the browser, and these copyrighted scripts are distributed without the permission of the original authors.

Amjad et al. have aimed to investigate mixed scripts [54]—scripts containing both tracking and functional code. Their classification relies on the tracking request density—i.e., the ratio of count tracking requests over the count of functional requests—tracking requests being identified using filter lists. Even though filter lists are considered high-precision [73]—i.e., having few false positives—this methodology leads them to consider scripts with a small number of tracking requests—compared to the number of functional requests—as functional, thus making it possible to 'dilute' tracking with more functional requests. They identified the need for a solution able to block individual functions and, more specifically, to selectively block tracking functions within mixed scripts.

Amjad et al. have continued their exploration of mixed scripts and have again noted the necessity of being able to block individual functions [55]. They first dynamically locate functions directly initiating tracking requests—as classified by filter lists. Using offline processing, they then rename the tracking functions with a different name, so that calling these functions results in a `ReferenceError`, effectively preventing the execution of the tracking functions. However, and contrary to what they assume, this may well lead to functionality breakage, as that exception will only be catched by the first `catch` block in the call stack and will terminate the JavaScript program if none exists. This is actually what called for the deployment of shims in Firefox Enhanced Tracking Protection [184] and of scriplets in uBlock Origin [130]: they replace tracking scripts with no-operations to avoid functional breakage induced by functional code calling functions from blocked scripts. Moreover, their approach cannot translate into a solution able to block tracking functions from unseen mixed scripts: they could only generate byte offsets of functions to rename for scripts seen during a crawl. This limitation originates from the fact that they are not able to re-identify the functions based on their content—i.e., their source code—and have to rely on their behavior. Furthermore, as tracking scripts are very likely channeling all their requests through a single function, only responsible for making these requests, it is unclear what the benefit of blocking these functions is, compared to using filter lists to block the request itself.

With JsRipper, as described in more details in section VI.2, we follow in the footsteps of these studies by identifying pieces of tracking code in any scripts on the web. We go further by introducing a signature scheme able to identify functions based on their source code only, and robust to minification and name mangling. We then propose an in-browser, on-the-fly solution allowing to selectively remove the functions responsible for tracking, and only them.

VI.1.3 Bundling and Minification

In this section, we present relevant features of bundlers and minifiers which guided the design of our function signatures, and call for specific properties to obtain robust signatures.

VI.1.3.1 Bundling Technology

A JavaScript bundler is a development tool that gathers all the modules and dependencies of a project and outputs a single file that is optimized for network transfer, and suitable for use in a browser. One of the most important features of a bundler is that it resolves the dependency graph and outputs different dependencies into one single file. Bundlers were first introduced in 2010 [24], when browsers did not support native JavaScript modules, and they thus provided a convenient way to use npm packages [196] in the browser, when the ecosystem was just starting to bloom. It had also the benefit of minimizing the number of requests, before HTTP/2 decreased the overhead of issuing numerous requests to the same host.

Before modules and bundlers existed, libraries used to expose their API as a global object, accessible to every other scripts. For instance, jQuery exposes the global functions `jQuery()` and `$()`. This practice resulted in global namespace pollution and possible conflicts between libraries which depended on different versions of other libraries.

Now, one of their most interesting features is their ability to removed unused items from dependencies, a technique known as tree shaking (see below). Ultimately, the result is still the same: functions from both functional and tracking dependencies will find themselves in the same script, and therefore cannot be blocked with current, location-based content blocking mechanisms such as filter lists, which treat scripts as the blocking unit. If the code features multiple entry points, some bundlers are able to generate multiple bundles, one for each entry point, decreasing the amount of code that need to be downloaded by the browser.

After the output file has been generated, it is processed by a minifier, which will reduce his size for faster network transfer (see below). Depending on which bundler is used, the assembly of the different dependencies can differ, especially as minification features vary from a minifier to another one.

In 2023, there are five major bundling tools, namely Webpack [22], esbuild [261], Rollup [20], Parcel [18], and Turbopack [249] with the 2022 Web Almanac [260] reporting that Webpack and Parcel are respectively present in 15 % and 2 % of the most visited 10,000 pages. In addition, they have also found that bundlers are mostly used on popular websites, from the top 10,000.

VI.1.3.2 JavaScript Module Types

Bundlers resolve dependencies between different JavaScript modules utilized as part of a project. There are two major types of JavaScript used on the web today: ECMAScript (ES) modules and CommonJS modules. CommonJS modules have been introduced by Node.js, and provide a global function, `require()`, allowing to import other modules based on a file path or the name of an npm module [109]. `require()` being a regular function, these imports are dynamic, making the static analysis of these difficult. To export items, libraries use the `module.exports` construct, `module` being a magic object introduced by the execution environment. However, CommonJS modules are not

```
// lib.js
function say(msg) {
  console.log('Hello!' + msg);
}

module.exports = { say };

// index.js
import { say } from './lib.js';

say('hello');
say('world');
```

Listing (VI.1) Input files: the library file uses CommonJS module conventions to export its function, and the application file uses ES imports to import it

```
(() => {
  var r = {
      992: r => {
        r.exports = {
          say: function(r) {
            console.log(r)
          }
        }
      }
    },
    e = {};

  function o(t) {
    var a = e[t];
    if (void 0 !== a) return a.exports;
    var n = e[t] = {
      exports: {}
    };
    return r[t](n, n.exports, o), n.exports
  }
  o.n = r => {
    var e = r && r.__esModule ?
      () => r.default : () => r;
    return o.d(e, {
      a: e
    }), e
  }, o.d = (r, e) => {
    for (var t in e) o.o(e, t) && !o.o(r, t)
        && Object.defineProperty(r, t, {
      enumerable: !0,
      get: e[t]
    })
  }, o.o = (r, e) =>
    Object.prototype.hasOwnProperty.call(r, e), (() => {
      "use strict";
      var r = o(992);
      (0, r.say)("hello"), (0, r.say)("world")
  })()
})();
```

Listing (VI.2) Bundle output by Webpack 5 (prettified for readability): it contains the imported function (lines 5–7), the runtime boilerplate (lines 11–34), and the application logic (lines 36–37)

Figure VI.1: Minimal example of a bundle produced by Webpack 5 for a CommonJS module

supported by browsers. As explained above, this has led to the initial development of bundlers, these resolving the dependencies during the bundling stage and outputting a single bundle encompassing all the imported modules.

Introduced with ES2015, ES modules introduced two keywords: `import` and `export`, and are supported by browsers. Being declarative, they are easier to analyze and, because of this, were first to be supported for tree shaking [41].

Because of the different nature of ES modules and CommonJS modules, bundlers cannot treat them the same way. Bundlers are able to 'inline' items imported from ES modules and generate a script as if the imported items—e.g., classes, functions, constants—had been written as-is in the output file, a process referred to as 'scope hoisting' [33], since it removes the need for a function surrounding the function and providing a dedicated scope for isolation. On the contrary, CommonJS modules require a small 'runtime' which exposes the `module` object, presented above, to the imported script, and allow dependents to query the module. Figure VI.1 shows an example of the runtime boilerplate inserted in the output bundle by Webpack when a CommonJS module is imported: each export scope is wrapped in an arrow function expression which receives the magic `module` object used to export items (line 3).

This example also highlights that exported items are scoped to the script within an Immediately Invoked Function Expression (IIFE): it is therefore impossible to redefine specific functions from outside the bundle, which was possible when dependencies were exposed as global variables.

VI.1.3.3 Tree Shaking

In its early days, Webpack included the whole dependency in the output module, even when only a small portion of it was actually used in the dependent script. This was responsible for sizable bundles, comprised of a large part of dead code. Modern bundlers are able to detect these unused imports, and to avoid including such dead code in the output bundle. This process is known as 'tree shaking', a term popularized by Rollup [32], referring to the tree structure of the dependency graph. Regarding the detection of bundled tracking dependencies, this means that we cannot rely on detecting entire dependencies, but must instead run a more fine-grained analysis, at the level of individual imported items, i.e., classes and functions, and require a signature scheme able to recognize these individual items in isolation.

VI.1.3.4 Minification versus Obfuscation

Minification and obfuscation are often mistaken for one another, however they are very different processes, employed with different goals in mind. Minification makes the output script as small as possible for network transfer, and is thus concerned with improving performance. To this end, it uses different strategies, especially comments and whitespace removal, value rewriting—e.g., writing `!0` instead of `true` to save two characters—equivalent syntax rewriting, and name mangling to make identifiers shorter (see below). Minification is a normal step in the build process of any production website, and Moog et al. have found it used on around 90 % of websites from the Alexa Top 10 k.

Obfuscation is aimed to make it hard for someone to read and understand the script behavior, making its reverse engineering costly, sometimes at the cost of transfer and execution performance.

Many type of obfuscation exist for JavaScript, a task made easier by the fact that JavaScript is a dynamic language and support dynamic evaluation from a string, using `eval()`. Given its performance impact and the additional build tooling required, script obfuscation is rarely found on the web, as measured by Skolka et al. [219].

Figure VI.2 provides an example of a JavaScript function that implements the Pythagorean theorem with the original, minified and obfuscated versions.

VI.1.3.5 Name Mangling

Name mangling is the process of changing identifiers for shorter names, while keeping the script functionality. For instance, local variables and functions defined and used in the same module can be renamed to save some characters. terser [63], the minifier used by default by Webpack 5, uses frequency analysis on the input script to determine the characters used for generating shorter names. It will thus fist scan the input script to determine the most frequent letters, and generate short names using these, starting with one-letter names, then two-letter names, etc. subsection D.0.1, in the appendix, highlights this behavior.

This means that names chosen during name mangling are dynamic, therefore our function signatures need to be able to handle this and recognize functions whose only local identifiers differ as the same functions, as long as the rest of the AST of the functions is the same.

```
function pythagorean(sideA, sideB) {
    return Math.sqrt(
        Math.pow(sideA, 2) +
        Math.pow(sideB, 2)
    );
}
```

Standard

```
function pythagorean(n,o){return Math.sqrt(Math.pow(n,2)+Math.pow(o,2))}
```

Minified

```
function a(){var k=['2BsmQWL','5230647togUuc','210608WxJTcE','580968uHNTLF','
    5165237BBYddg','81MQzJhp','30xqIxoj','1891248XukuUz','673266SpCvQg','2741010
    lohNTR','sqrt','pow'];a=function(){return k;};return a();}function b(c,d){var
     e=a();return b=function(f,g){f=f-0x13d;var h=e[f];return h;},b(c,d);}(
    function(d,e){var i=b,f=d();while(!![]){try{var g=-parseInt(i(0x146))/0x1*(
    parseInt(i(0x13e))/0x2)+parseInt(i(0x143))/0x3*(parseInt(i(0x140))/0x4)+-
    parseInt(i(0x147))/0x5+-parseInt(i(0x141))/0x6+-parseInt(i(0x142))/0x7+-
    parseInt(i(0x145))/0x8+-parseInt(i(0x13f))/0x9*(-parseInt(i(0x144))/0xa);if(g
    ===e)break;else f['push'](f['shift']());}catch(h){f['push'](f['shift']())
    ;}}}(a,0xd5070));function c(d,e){var j=b;return Math[j(0x148)](Math['pow'](d
    ,0x2)+Math[j(0x13d)](e,0x2));}
```

Obfuscated by [146]

Figure VI.2: JavaScript code for the Pythagorean theorem. The three pieces of code are all equivalent.

VI.2 Building Function Signatures and Statically Detecting Tracking Functions

In this section, we present our function detection method and our crawl methodology. We want to be able to recognize bundled tracking functions originating from known tracking libraries, such as Google Analytics or Adobe Launch. This allows us to obtain a lower bound of tracking script bundling and enables function-level blocking by substituting recognized tracking functions with benign shims in the browser. We work at function-level because this is a well defined code unit, small enough so that it should have a single responsibility, which makes it possible to replace it with benign code.

VI.2.1 Generating Function Signatures

We aim to be able to recognize tracking functions bundled in other scripts. As the transformation applied by different bundlers differs, and because the variable names given by the minification stage may differ depending on the location within the bundle, our recognition method must ignore these local identifiers. We thus devised a function signature generation technique to recognize these tracking functions when bundled, while aiming to keep false positives rare so that the signatures could be directly used for in-browser blocking.

Given the (minified) source code of a JavaScript function, its signature is generated with the following steps:

1. Parse the function source code and generate an AST.
2. Erase the local variable identifiers.
3. Serialize the AST and hash the result.

All these steps need to be deterministic so that the same function source code generates the same hash, making it possible to compare the function against a bank of known function signatures.

Step (1) We use the parser of SWC [98] to generate an AST of the given JavaScript function source code.

Step (2) We then visit the resulting AST and erase identifiers of local bindings (variables and function parameters) to make the signature minifier-agnostic and location-independent, as they can vary depending on the minifier used and the function location in the generated bundle. These identifiers are simply replaced by the empty string so that functions having different local identifiers produce the same result after erasure. Other identifiers are not modified. In particular, web APIs identifiers must be retained to match functions of interest and avoid false positives, e.g., `document.cookie` or `localStorage` APIs must be preserved during the signature generation process.

```
// Function from
// https://www.google-analytics.com/analytics.js
function(a) {
    var b = [], c = M.cookie.split(";");
    a = new RegExp("^\\s*" + a + "=\\s*(.*?)\\s*$");
    for(var d = 0; d < c.length; d++){
        var e = c[d].match(a);
```

```
        e && b.push(e[1]);
    }
    return b;
}

// Representation of the function with local identifiers
// erased (identifiers are here replaced with underscores
// for readability)
function(_) {
    var _ = [], _ = M.cookie.split(";");
    _ = new RegExp("^\\s*" + _ + "=\\s*(.*?)\\s*$");
    for(var _ = 0; _ < _.length; _++){
        var _ = _[_].match(_);
        _ && _.push(_[1]);
    }
    return _;
}
```

Listing VI.3: Example of local identifier erasure within a function. The erasure is actually applied on the AST, not on a text representation of the source code as shown here

It is unlikely in real-world scenarios that the bundling and minification processes would modify the AST of functions beyond renaming local bindings. Other transformations include whitespace removal, dead code elimination, and equivalent expression substitution (e.g., replacing `true` with `!0`), but these behave similarly across bundlers and minifiers, or are ignored in the AST (e.g., whitespace). Moreover, Webpack is currently the most used bundler on the web [260], by far, minimizing the risk of false negatives due to a different bundling/minification process. An example of local identifier erasure can be found in Listing VI.3.

Step (3) Finally, we serialize the modified AST and hash the result using the SHA-256 cryptographic hash function, minimizing the risk of hash collisions. The function signature of the tracking function shown in Listing VI.3 is the following SHA-256 hash:
`5772f21c0261d01ad84e68b628b752d7e2e2d0e09046d7abbcd8c74bab75fc4e`.

VI.2.2 Collecting Scripts

As we want to detect tracking functions from known tracking scripts in other scripts, we need to build a bank of tracking function signatures. We thus ran a web crawl on the top 10 k Tranco [204] to collect these scripts. Specifically, we crawled the homepages of the top 10 k Tranco and downloaded all external scripts found in the page. We opted for restricting our crawl to the top 10 k as this range of websites is the one using bundlers the most [260]. The crawler was built by automating Firefox with WebDriver. For each homepage, we collected the list of external scripts exposed in the `document.scripts` API and downloaded them in the crawler process, bypassing Firefox tracking content blocking mechanisms and the browser Same-Origin Policy (SOP) so that we would download tracking scripts which could otherwise be blocked. As the functions originating from tracking scripts will be matched against minified scripts, collecting the tracking scripts as used in the wild (instead of using vendor-provided versions or even upstream source code for open-source tracking

libraries) allows us to easily obtain minified versions of these tracking scripts. The crawl was run on October, 4 2022 using the Tranco list from the same day.

VI.2.3 Detecting Bundled Tracking Functions

We aimed to detect tracking functions from tracking scripts that also appear in other scripts which cannot be blocked as a whole since they also contain functional code. Leveraging our function signature mechanism, we employed a two-stage analysis. We first automatically detect all functions originating from known tracking scripts in other candidate scripts, then filtered these functions using text-based heuristics, and finally manually classified tracking functions, which can then be found back in other candidate scripts. We focus on function declarations—functions defined with the `function` keyword—and leave out arrow function expressions—defined with `() => {}`, since arrow functions are used in cases which would not be interesting to detect or block, while potentially increasing the function count significantly. Moreover, bundlers use function declarations instead of arrow functions in their boilerplate for performance reasons, since the arrow functions capture their scope, bringing an additional runtime cost.

VI.2.3.1 Signature-Based Large-Scale Automated Analysis

Having obtained the script corpus detailed in the previous section, we generated a database of signatures of every function in each external script. The AST was generated for each script to discover all functions and was reused to generate the signature of each function. We also stored the associated source code span of the function (the byte range where the function is defined). For each script, its SHA-256 is also stored, whether it was a first-party script in our crawl, and whether it is in the EasyList [105] or EasyPrivacy [6] filter lists, in which case we call them tracking scripts.

We then selected the functions appearing in tracking scripts and in other first-party scripts, not found in filter lists. As we mainly focus on bundles in this contribution, we restricted our analysis to first-party scripts, as bundles are most likely to be served as first-party for performance reasons (to avoid opening a connection to a different server to load the bundle, whose loading speed is often critical). We thus obtained a list of functions from tracking scripts appearing in other scripts which are likely bundles.

VI.2.3.2 Manual Labeling

To be able to detect tracking scripts bundled in other scripts, and to enable blocking bundled tracking functions, we manually classified the tracking functions also appearing in first-party scripts, based on their source code. As not all functions found in tracking are privacy-harmful, we first filtered these functions based on their source code, looking for the specific privacy-relevant APIs [220] listed in Table VI.1. This filtering significantly reduces the number of functions to manually classify, while preserving functions whose blocking would significantly improve user privacy [220]. Then, we classified functions actively taking parting in tracking, relying on our domain knowledge and knowledge of web APIs and client-side tracking practices. The classified functions play two roles: (1) they are then used to identify tracking scripts bundled in other scripts and (2) they can be used to

Table VI.1: Regular expressions used to recognize privacy-relevant APIs; \b denotes a word boundary

Persistent client-side storage
`\.cookie\b`
`localStorage`
`\bStorage\b`
Direct network access
`XMLHttpRequest`
`\bfetch\b`

build a bank of function signatures actively participating in tracking whose blocking in the browser would improve user privacy. These functions must be distinctive enough to avoid false positives—i.e., mis-detecting a function from a script as being a tracking function—and actively taking part in tracking themselves, e.g., accessing privacy-relevant APIs.

VI.3 Results

This sections reports on the results of our crawl and our analysis of tracking functions bundled into other scripts.

VI.3.1 Script Statistics

In this section, we detail various general statistics about the crawl and the analyzed scripts. When crawling the homepages of the top 10 k Tranco as explained in the previous section, and because not all domains in the Tranco list point to websites, we obtained the scripts of 6,702 unique domains. In total, we collected 41,924 unique scripts, based on their content hash. Among these, 1,025 unique scripts were found in EasyList or EasyPrivacy lists.

We then generated the AST of each script and detected every function in each, then generated the function signature of each function, totaling 38,401,137 function signatures and 11,552,421 unique function signatures. This indicates that some functions appear in multiple scripts on the web, but these are mostly very short functions, constitute code boilerplate, including code generated by automated tools, such as bundlers, as seen in subsection VI.1.3. The distribution of (uncompressed) file sizes of these unique external scripts can be seen in Figure VI.3. As scripts are compressed in-flight, the size on-the-wire of these scripts is significantly smaller. The median file size of these individual scripts is 20 kB, while the 90th percentile is 400 kB. Figure VI.4 shows the count of unique scripts per domain, the median is 7 scripts. The distribution of function counts in each script can be found in Figure VI.5, the median count of functions per script is 100 functions and the 90th percentile is 2,000 functions. The number of tracking scripts having few functions is smaller, since implementing a tracking script requires a certain minimal count of functions. The distribution of the byte size of function source code can be seen in Figure VI.6; when collecting functions from the AST we do not

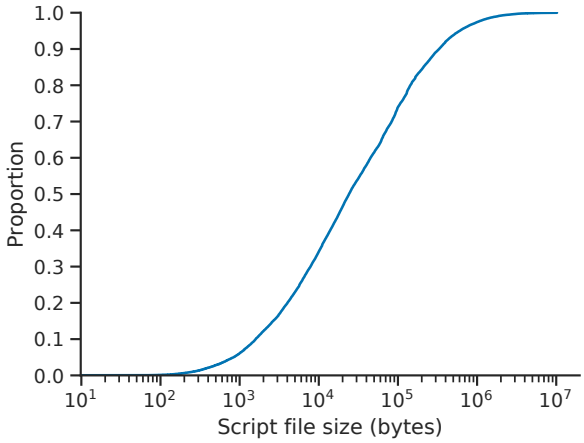


Figure VI.3: File size of external scripts

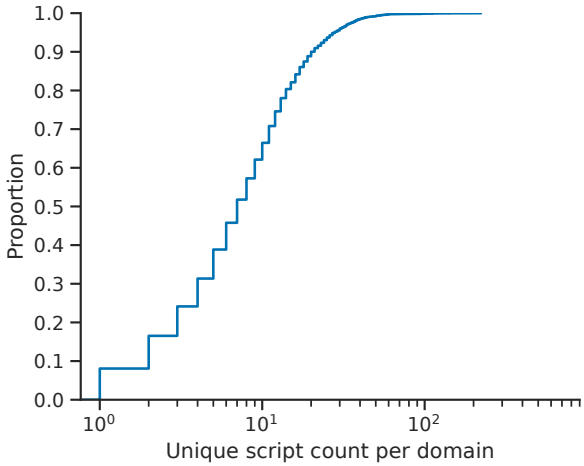


Figure VI.4: Script count per page

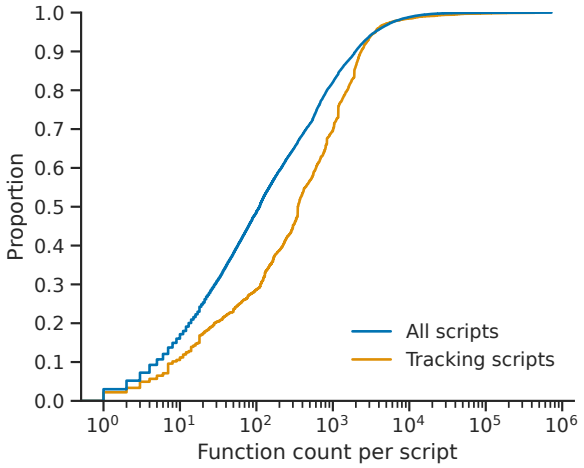


Figure VI.5: Count of functions per script

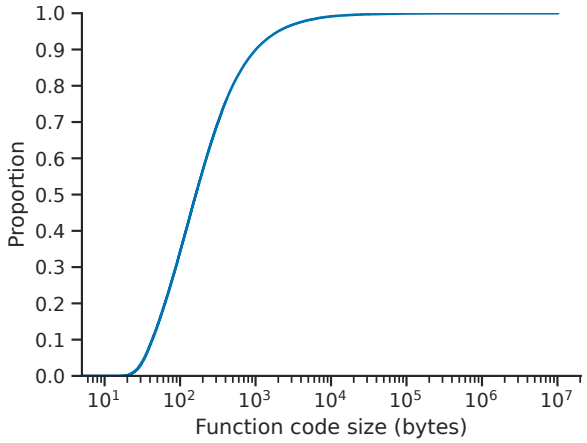


Figure VI.6: Source code size of functions

exclude nested functions, that is, we do keep every function declaration, including top-level IIFEs—JavaScript function defined to be called immediately, which are useful to avoid polluting the global namespace. This figure shows that half the functions are less than 200-byte long: they are thus very likely to have a single responsibility—e.g., not mixing functional and tracking code—but may also be less identifiable in themselves.

VI.3.2 Bundled Tracking Functions

After detecting the functions from tracking scripts also found in first-party scripts, we filtered these functions based on their source code, keeping only the functions matching the regular expressions detailed in Table VI.1, thus focusing on functions accessing privacy-relevant APIs. We call these remaining functions 'candidate functions'. We then manually marked functions having self-contained privacy-harming behaviors, e.g., storing or accessing cookies or `localStorage`, while making sure the functions were distinctive enough as not to mis-detect similar non-tracking functions. We sorted the candidate functions according to their count of appearance in other scripts and limit our manual classification to the top 150 functions as the classification benefit rapidly diminishes.

```
// Function from
// https://www.google-analytics.com/analytics.js
// which is distinctive enough in isolation to be safely
// classified as tracking.
// This function returns an array of cookies built from
// the native cookie API, which exposes a string where
// key=value pairs (i.e., cookies) are separated by a
// semicolon.
function(a) {
    var b = [], c = M.cookie.split(";");
    a = new RegExp("^\\s*" + a + "=\\s*(.*?)\\s*$");
    for(var d = 0; d < c.length; d++){
        var e = c[d].match(a);
        e && b.push(e[1]);
    }
```

Table VI.2: Tracking scripts whose tracking functions are most often bundled (truncated to top 5)

Tracking script	Bundle count
https://weby.aaas.org/weby_bundle_v2.js	313
https://branch.io/js/all.js	251
https://d.mail.cafepress.com/track.v2.js	118
https://script.hotjar.com/modules.21c2ce197b1deec7582e.js	105
https://lightning.adultswim.com/launch/7be62238e4c3/22d196a3e151/launch-2fa6614adbd9.min.js	97
https://assets.adobedtm.com/e0903a2aaadb93ceed6a5acaaacbb9b9846eaa41/satelliteLib-88084863a26dad129e2d755e9777f20485407022.js	89

```
    return b;
}

// Function from
// https://asg.phukienthoitranggiare.com/Cqp6VQ5.js
// which cannot marked as tracking as it is not
// distinctive enough in isolation and would cause
// false positives otherwise
function(t) {
    return localStorage.getItem(t);
}
```

Listing VI.4: Examples of candidate functions (prettified for readability)

Listing VI.4 shows an example of function origination from a tracking script which can easily be marked as tracking in itself and an example of another function also found in a tracking script which cannot be classified as tracking in isolation, otherwise risking producing false positives when detecting this function in other scripts or risking breaking pages if the function signature is leveraged for in-browser blocking.

Out of the 150 functions originating from tracking scripts, we classified 47 functions as actively participating in tracking—i.e., accessing privacy-relevant APIs—and being distinctive enough to avoid false positives. We then queried the list of scripts having at least one function matching a signature of a function manually marked as tracking. We found that 4.37 % of scripts unknown to filter lists contained a function marked as tracking—totaling 978 scripts—and that 22.7 % of domains loaded at least one script containing such functions. An excerpt of the tracking scripts the most often found in other scripts unknown to filter lists can be found in Table VI.2. The complete list can be found in Table D.1. Note that the URLs may not reference the 'upstream' tracking script, but simply an occurrence of it in the wild. For instance, the tracking script in the fifth row of this table is a script of Adobe Launch that the website is re-hosting. Some examples of scripts containing functions manually marked as tracking can be seen in Table VI.3, most of these are indeed script bundles, others are tracking scripts unknown to filter lists. Table D.2, in the appendix, also lists the tracking scripts where the tracking functions originate from.

Table VI.3: Examples of scripts containing tracking functions, these scripts are not part of filter lists

Scripts containing tracking functions
https://tags.tiqcdn.com/utag/intuit/brand-icom/prod/utag.js
https://www.sanofi.com/.resources/sanofi-lm-platform/themes/sanofi-platform/dist/common~2022-09-28-22-03-50-000~cache.js
https://assets.adobedtm.com/9aafaf1151ac/682bb4885835/launch-be686e885d32.min.js
https://www.infoplease.com/sites/infoplease.com/files/js/js_sSle-sUI_xeB_8F5RW4Nq8rRQbKgBG9MDKvSNyHX8ww.js
https://www.grammy.com/_next/static/chunks/484-ed227be4db7efa1f.js
https://s1cdn.vnecdn.net/vnexpress/restruct/j/v3797/v3/production/lazyload.js
https://assets.adobedtm.com/331fbea29f79/a5b25a446515/launch-e80baf9c0255.min.js
https://assets.poetryfoundation.org/assets/scripts/vendors~main-9b6becb7b8.js
https://static.phemex.com/s/3rd/sd/sd-1.17.2.min.js
https://assets.adobedtm.com/8a93f8486ba4/5492896ad67e/launch-b1f76be4d2ee.min.js
https://metamask.io/121cd9c2bdc4dd8c8ec9ead858719809d6d18de3-a80ac18016e9cb1f8728.js

VI.4 Blocking Tracking Functions In-Browser

In this section, we present a proposed system for leveraging the function signatures for in-browser tracking function blocking.

VI.4.1 Leveraging Function Signatures for In-Browser Blocking

We have used our detection system to discover script bundles, having functional code mixed with tracking code originating from known tracking libraries. Since they contain a large part of functional code, likely to be required for the page to work, these script bundles cannot simply be added to filter lists.

As we have designed our tracking signatures around a syntactic unit—functions—this makes it possible to also leverage them for blocking unwanted code execution in a fine-grained manner, unlike previous works relying on behavioral signatures [73]. As our static analysis relies on script source code, this allows to identify the origin of potential tracking behavior, by focusing on privacy-relevant APIs. Indeed, it becomes possible to substitute the function body of detected tracking functions bundled in other scripts with benign code, thus preventing the tracking behavior while avoiding breaking the page. In addition, since the unit on which signatures work on is well-known, they are easy to reason about, enabling to contain the number of false positives, making them a viable ground truth for in-browser blocking.

VI.4.2 Function Substitution Strategies

VI.4.2.1 Signature-Only Blocking

We first devised a first strategy to leverage these signatures for tracking blocking where the function signatures are used as-in in the browser. On top of making it possible to prevent tracking behaviors

which are currently impossible to avoid without also blocking functional code, this has the additional benefit of providing a robust solution, as websites and scripts containing tracking may evolve. Indeed, since our signatures only rely on the local source code of the tracking functions, they are immune to bundle modifications which would change the location of these functions within the script. In addition, not being linked to particular scripts—unlike filter lists, which commonly rely on URLs to block—this also provides a robust solution for future scripts integrating tracking functions.

To deploy this tracking function detection and blocking strategy in-browser, it is required to be able to generate the function signatures in the browser, by leveraging hooks to rewrite the scripts when downloaded by the browser but before they are parsed and executed. Furthermore, as our signature generation process involves serializing the generated AST, the in-browser signature generation must use the same parser so that the AST is exactly the same given a function source code. As our function signature library is written in Rust, we leveraged its ability to compile to WebAssembly which can then be called from JavaScript and integrated in a WebExtension, making it possible to implement this strategy without modifying the browser. It is possible for a WebExtension, in a browser implementing the `webRequest.filterResponseData()` API [83], e.g., Firefox, to hook into the HTTP response stream and rewrite the received response. When stream chunks of an HTTP response for a script are received, they are first concatenated in a single buffer. In the case of scripts, this is unlikely to degrade performance, as the browser is required to check the hash of full script content for scripts using the Subresource Integrity security feature [82]. The buffer is then parsed using the same parser as the one used for generating the signatures, and signatures are generated for all functions discovered in the script. If some functions matched known function signatures, distributed to the WebExtension in the same way filter lists currently are, these tracking functions can then be substituted with benign shims, preventing the tracking behavior while preserving the page functionality. Not all functions require shims, and their body can then simply cut out by leveraging the byte span preserved in the AST, which makes it possible to obtain back the byte range within the source code file. Other function bodies do need to be replaced by appropriate shims to avoid breaking the page. Page breakage would otherwise happen when the blocked function returns a value used by the caller function in a way that would raise an exception—thus halting the script execution—if absent, i.e., `undefined`. Shims are thus required to return dummy values, e.g., an empty array in the case of the function from Listing VI.3. As the classification of tracking functions already requires a manual step, these shims can be hand-written at the same time and distributed along the function signatures. We leave the automated generation of these shims, from the function source code, to future work.

VI.4.2.2 Filter-List Inspired Strategy

Another possible deployment strategy for blocking detected tracking functions is to generate script rewriting rules offline, substituting byte ranges with the same shims as discussed in the previous section. This alleviates the browser from the need to parse all incoming scripts and generate functions signatures for all discovered functions. However, this strategy is also more brittle and could result in lower effective blocking coverage, as these rewriting rules need to be generated for each individual script bundling tracking functions.

VI.4.2.3 Hybrid Strategy

When implementing the first strategy, we have identified potential performance issues when dealing with scripts of significant size (> 2 MB uncompressed), which result in delayed page loads, while the script is being processed by the WebExtension. We thus propose a hybrid approach where scripts whose size is above a specific threshold are not processed to generate function signatures but are instead handled along the second strategy. This hybrid approach thus brings robust blocking of tracking functions in current and future bundles while also handling scripts of significant size and keeping the size of the rewriting list reasonable—as it requires one rule per *script*—since these scripts are rare on the web as can be seen in Figure VI.3 and as measured by the Web Almanac [259].

VI.5 Limitations and Discussion

VI.5.1 Static Analysis and Obfuscation

As our analysis method is purely static, it inherits the usual limitations from statically analyzing JavaScript source code. In particular, we assume that scripts contain functions fine-grained enough so that tracking functions can be individually blocked. This is a safe assumption for most scripts, particularly in the context of bundles where factory functions may be introduced as boilerplate by bundlers, thus introducing clear boundaries at the module-level. However, this assumption can be defeated by obfuscation techniques, by either merging functional and tracking functions, making it impossible to block tracking at the function-level without breakage, hiding the function declarations with dynamic techniques—e.g., using the `eval()` JavaScript function, allowing to execution code from a string—or to hide the accesses to privacy-relevant APIs, as we first filter out functions without these accesses, by relying on the regular expressions from Table VI.1. It should be noted that Skolka et al. observed that less than 1 % of scripts of their 424,023-script sample were obfuscated [219], moderating this limitation. In addition, our solution is capable of blocking parts of a bundle that would be obfuscated, as the bundling process would segregate this obfuscated code in a function, which is our blocking unit. It is extremely unlikely that an entire bundle be obfuscated, because of the performance hit this would entail.

Furthermore, as our analysis works at the function level, tracking functions must be distinctive enough in themselves so they can be safely blocked, with low risks of false positives. We did observe such distinctiveness in the wild, during our manual classification. In particular, we observed many different implementations of the `document.cookie` string parsing function shown in Listing VI.3, confirming that implementations are diverse enough to separate functions from tracking script from other functions. However, we did also observe functions directly accessing privacy-relevant APIs which were not distinctive enough to be safely blocked, without risking blocking functions from functional code, as shown in Listing VI.4. To address these cases, it would be possible to widen the range of candidate functions, by including the functions calling these non-distinctive functions, and considering blocking these instead.

VI.5.2 Third-Party Scripts

During our manual classification, we restricted ourselves to functions originating from tracking scripts that were also found in *first-party* scripts unknown to filter lists. This choice was motivated by the fact that we specifically focused on script bundles, which we assumed to be served as first-party scripts, for performance reasons. The tracking functions discovered, after manual classification, were then searched in every other scripts (not restricted to first-party scripts). Our analysis methodology could be extended to consider third-party scripts as well when initially searching for functions from tracking scripts shared in other scripts. This may discover other tracking functions shared in other scripts, whose deployment practices are different—e.g., using caching, third-party Content Delivery Networks (CDNs) such as CloudFront [216].

VI.5.3 Inline Scripts

In this contribution, we focused on detecting tracking functions in *external scripts*. This is motivated by our focus on bundles which are the most likely to embed tracking functions with functional code. Indeed, it is very unlikely that bundles are inserted as inline scripts—i.e., within `script` tags—as it is critical for performance that bundles benefit from browser HTTP caching, to avoid loading the same bundle across different pages of the same website. Our function signatures could still be used to detect and block tracking functions in inline scripts, by rewriting the HTML document using the same WebExtension hook used for script rewriting.

VI.5.4 Bundlers Usage Trend

Finally, we have shown that tracking functions are already found in other scripts unknown to filter lists in a significant share of websites. Bundlers have been common practice for several years and their usage should become event more prevalent as bundlers are integrated in all-in-one server-side frameworks, such as Next.js [248] or NuxtJS [87], which automatically generate and serve script bundles. This trend could eventually significantly diminish the current abilities of URL-based content blockers, leaving users unprotected from web tracking.

VI.6 Conclusion

In this contribution, we introduced a robust function signature technique with the aim of detecting tracking functions originating from tracking scripts known to filter-lists in other scripts, especially script bundles. With a large-scale web crawl, we have quantified the occurrences of such tracking functions in other scripts, and found that 4.37 % of unique scripts contained tracking functions from known tracking libraries, and that 22.7 % of websites in our sample loaded such a script. We finally proposed a hybrid, in-browser blocking strategy, leveraging our function signatures, making it possible to identify tracking functions inside bundles.

Availability

We make available our function signature generation library, the crawl infrastructure and our proof-of-concept hybrid blocking strategy WebExtension. They can be found in the following repository: https://archive.softwareheritage.org/browse/origin/https://gitlab.inria.fr/Spirals/jsripper-artifact.

CHAPTER VII

Conclusion

Web tracking has seen steady development since the inception of the web in 1990. It is now widespread and leverages numerous web tracking vectors to follow users in their browsing and observe their behavior on individual web pages. Web tracking has raised serious privacy concerns and has called for the development of web tracking defenses that users could employ in their browsers to protect themselves. Such defenses have existed for a long time, but websites and the web platform are always changing, leading to a cat and mouse game between web trackers and tracking protections.

In this thesis, we have focused on one kind of tracking protection: content blocking, which entails blocking parts of web pages unwanted by users. This defense can be used in conjunction with other protections, such as clients-side state isolation or regular storage contents deletion. It supplements these other protections and can in particular address same-page tracking, which is often left out by other defenses. We presented four contributions which aim to reduce the amount of JavaScript executed client-side, thus greatly reducing the ability of web pages to track users.

VII.1 Contributions

VII.1.1 Investigating Page when Disabling JavaScript

We have investigated page breakage when browsing with JavaScript disabled. To this end, we developed a framework to detect breakage of individual page elements and common element constructs. By crawling 6,384 pages, we leveraged that framework to quantify how much blocking JavaScript breaks pages and impedes browsing, separating elements in the main section of the page, for which the user is more likely to have reached the page, and elements around it. We detailed the breakage rate of each type of studied elements in the main section and around it. We found that 43 % of web pages are not strictly dependent on JavaScript and that 67 % are likely to be usable as long as the user is only interested in the main section. By measuring the number of tracking requests with and without JavaScript, we showed that browsing without JavaScript greatly improves user privacy. We finally discussed the viability of browsing the web without JavaScript for privacy.

VII.1.2 Bridging the Gap Between the User and the Browser with User Browsing Intent (UBI)

Previous works regarding content blocking were mostly concerned with making the entirety of the page work, even if that means allowing some trackers. Acknowledging that the user does not always require the entirety of the page to be working, we introduced the concept of User Browsing Intent (UBI), and focused on the 'read-only' UBI, where the user only wants to read the page content and not interact with it. Building on the acquired knowledge about page breakage when JavaScript is disabled, we designed a limited set of client-side repairs that target common page breakage types to fix them without the page's provided JavaScript. Using a semi-manual comparison of 3,958 page screenshots with and without our repairs, we classified them based on the amount of information lost when disabling JavaScript. We found that our in-browser repairs make more than 27 % more pages compliant with the 'read-only' UBI. More than 62 % of pages are compliant with the 'read-only' UBI if the user only tolerates minor information loss, and more than 77 % of pages are compliant if the user also tolerates the loss of some non-central sections. We also measured the number of tracking requests disabling JavaScript with our repairs enabled, and found that 97.7 % of them are prevented on average, bringing significant privacy improvements. If the page is not compliant with the 'read-only' UBI or if the user UBI is not the 'read-only' UBI, it is easy to enable back JavaScript.

VII.1.3 Reducing Interface Components Dependency on JavaScript Server-Side

Switching to server-side, we proposed a server-side plugin to explore the potential benefits of replacing interface components that usually rely on JavaScript with noscript alternatives. We targeted the most popular web interface component framework, Bootstrap, and developed a server-side plugin that manipulates HTML as a middleware. Deploying this plugin, JSRehab, allows to remove the Bootstrap library on the pages where all components can be replaced and, if Bootstrap was the only piece of JavaScript used on those pages, to apply a much stricter Content Security Policy (CSP) policy, improve security. We also measured the energy consumption on mobile devices when removing the Bootstrap library and found gains of at least 5 % while introducing only a very small size overhead to the HTML payload.

VII.1.4 Detecting and Blocking Individual JavaScript Functions for Privacy

Back to client-side, we have investigated a more fine-grained and resilient script blocking strategy than existing filter lists. Focusing on script bundles, which result from the bundling of multiple scripts after dependency resolution, we aimed to be able to block tracking functions bundled with functional code. Since blocking the whole script would break the page—as this would also block functional code—these scripts are not currently blocked by filter lists; such tracking functions are thus currently slipping through the net. In this contribution, we have devised a JavaScript function signature scheme for detecting these bundled tracking functions: these signatures rely on the function AST and erase the local identifiers for increased robustness. With a large-scale web crawl, we collected 41,924 unique scripts and computed the signature of each 11,552,421 unique function

signatures. Using common filter lists to detect known tracking scripts, along with manual classification, we collected a set of tracking functions whose body is identifying enough in itself to avoid false positives. Using the signatures of these tracking functions we found that 4.37 % of unique scripts—that were not known to filter lists—contained these tracking functions and that 22.7 % of domains of our sample contained such a script. As these function signatures allow to locate the unwanted functions, it becomes possible to target them for replacing with shims, effectively making them no-operation functions and preventing their tracking behavior. We have thus finally proposed a hybrid function-blocking strategy, relying on these function signatures.

VII.2 Future Work

VII.2.1 Keeping Filter Lists Lean

Even if filter lists suffer from several issues, they are still useful to block widespread tracking, especially on slowly changing websites, which may not be trying to evade them actively. However, as we have seen, they tend to only grow larger and larger [222], leading to potential issues, especially on mobile devices with limited resources and because of Google's Manifest V3 [94] limitations. Future work could thus further investigate keeping filter lists lean, as explored in [222]. In that work, Snyder et al. reduced the number of filter rules by more than 90 % by crawling landing and internal pages from 10,000 sites from the top Alexa and observing which rules were used the most [222]. It would be useful to run this testing continuously and generate light versions of lists accordingly.

Additionally, future work could investigate the detection and removal of obsolete and redundant rules. For instance, a continuous integration system could regularly check that the rules are still useful on the page for which they have been added to the list. Such continuous integration system would require that a browsing scenario be written for each rule, and that scenario would be performed automatically, checking that the rule is still triggered. It could also run specific checks to verify that the page is not broken, similarly to end-to-end website automated tests.

VII.2.2 Increasing Fine-Grained JavaScript Blocking Ability

JsRipper provides JavaScript function signatures allowing to locate unwanted functions so they can be blocked. To this end, it relies on each function's AST, whose local identifiers are erased for increased robustness. Future work could investigate extending this technique to be able to target functions that are not distinctive enough in themselves. JsRipper indeed only relies on function bodies to identify them. Analyzing how the function is used in the script, statically or dynamically, could be investigated to further increase the share of functions such tool is able to target for blocking. Further processing, based on graph analysis or machine learning, could help decide whether the function should be blocked.

VII.2.3 Assisting Shim Generation

As explained in subsection II.4.2, some content blocking tools allow to replace specific scripts with shims, to avoid page breakage. Shims are meant to provide the same public API as the scripts they

replace, but their functions are effectively no-operations, rendering their execution benign. Shims provided by Firefox Enhanced Tracking Protection and uBlock Origin are hand-written, which limits their number and thus their coverage. Smith et al. have investigated a different approach where privacy-sensitive APIs—e.g., cookies or `localStorage`—are temporarily replaced with no-operations by rewriting the scripts containing both functional and tracking code [220]. However, this technique requires distributing the resulting, sometimes large, scripts in their entirety, which raises concerns about the scalability of this approach, along with legal questions since the browser is effectively distributing modified copyrighted scripts without the authors permission. Future work could investigate automatically generating the shims which are currently hand-written, allowing this technique to scale and could help generate the shims needed by JsRipper when swapping the tracking function bodies with no-operations to avoid breakage.

VII.2.4 Improving the Privacy–Usability Trade-off

A more general perspective would be to investigate how to further improve the privacy–usability trade-off, and make advanced privacy protections accessible to more people. Multiple browser vendors have recently rolled out radical changes for the web platform, which significantly improve user privacy while having a very limited negative impact on browsing, as we have presented in section II.5. Future work could explore how to limit browsing impediment of other privacy preserving defenses against browser fingerprinting, such as in-depth configuration uniformity. Firefox for instance already ships with such in-depth defense, gated by the `privacy.resistFingerprinting` preference, but enabling it inevitably results in some inconvenience, directly originating from its behavior: for example the browser timezone is set to UTC [46] and canvas data readings are completely randomized, resulting in colored patterns if shown to the user. Exploring how to reduce these usage barriers would drastically increase the adoption of these protections.

VII.3 Insights and Perspectives

In this section, we discuss insights gained while working on in-browser content blocking for privacy, remaining general challenges, and what future we can envision it having.

VII.3.1 Loose Coupling between Page Elements and Behavior

One of the main underlying challenges of content blocking, especially regarding blocking scripts, is, surprisingly, the loose coupling between the DOM elements and their scripted behavior: i.e., statically connecting an element—e.g., a button—to the relevant portions of scripts responsible for its behavior is basically impossible. The markup, styling, and behavior are indeed provided by three different languages (HTML, CSS, and JavaScript) whose design aimed for loose coupling. Loose coupling is usually desired in computer science but, in this case, it makes it difficult for the user to protect themselves as locating—and thus blocking—the relevant scripts or portions thereof is not possible.

Interestingly, this loose coupling has been acknowledged as a reason for maintenance issues in the web development community and has stimulated the development of new methods and tools that

deliberately increase the coupling by keeping the markup, styling, and scripts together. Component frameworks, like React [138] and Vue.JS [278], have pioneered this idea of keeping the markup and behavior implementation together. More recently, the same strategy has been applied to styling, using different methods: scoped CSS [40], atomic CSS [35], and CSS-in-JS [25, 34]. Even if the methods differ, their goals are the same: keeping styling close to other constituents of the component—thus increasing the coupling between these. Keeping the constituents of a component closer makes it easier to reason about them as part of a single component and treat them as such. In particular, when deleting a component, the markup, scripts, and styling of this component are all deleted, avoiding remnants of code to be left behind and littering the codebase. This highlights the usefulness of tighter coupling between the constituents of components for the development phase. However, information about this coupling—i.e., what portions of scripts work with a particular markup portion—is lost when rendering these components for distribution to clients and thus cannot be used for content blocking in the browser.

If it were possible to recover this coupling information in the browser, it could allow fine-grained content blocking, at the interface component level.

VII.3.2 New Resource Types and Tighter Weaving

Another major challenge of content blocking is the always evolving web landscape and the inception of more interleaved resources, making their individual blocking much harder. As we have seen in chapter VI, script bundling is now common practice, even if still adopted by popular websites mostly. As scripts responsible for different mechanisms of the page—e.g., component rendering, interactivity, user tracking, etc.—are now part of the same script, content blocking tools having as blocking unit the whole script are unable to protect against unwanted portions of these scripts. We tackled with issue with JsRipper in chapter VI, which offers promising strategies for blocking only portions of scripts. However, interleaving of scripts and resources can only become tighter, rendering this task even more difficult in the future.

Furthermore, new types of resources, and especially of executable resources, have appeared and may continue to be added to the web platform which may threaten user privacy in new ways or make their blocking harder. After `asm.js`, WebAssembly has been integrated into browsers with two major goals: executing faster than JavaScript and providing a low-level format as target for compiled programming languages such as C++ or Rust. WebAssembly being a binary format, with low-level instructions, it is even more challenging to inspect and process to protect the user from the potential privacy harm of its executable behavior. These could threaten user privacy in new ways if they become the norm on the web.

VII.3.3 Non-Alignment of Interests of Producers and Consumers

User privacy, especially in the browser, has ultimately always been a cat and mouse game: website owners deploy privacy-invasive technologies, then privacy defenses are developed and deployed and, in turn, websites set up new strategies to circumvent these protections. As websites owners usually have no economic incentive to respect users' privacy and instead actually benefit from targeted advertisement and privacy-invasive handling of personal data with direct financial rewards, it

is hard to see this chase stopping anytime soon. As most client-side defenses can usually be circumvented by websites with enough effort, the fact that most websites did not have stronger reactions against content blocking—e.g., rates of active evasion from filter lists have been reported to be very low [222]—can only be explained by the current lack of pecuniary incentive—i.e., the loss of income for these websites is currently not worth actively fighting back. However, this may be about to change with the more generalized deployment of built-in privacy protections in browsers, and increasing awareness towards online privacy. One could try to propose new web APIs for content blocking, but it is hard to envision this as being a viable direction. Indeed, this would require some cooperation from website owners and, since browsers are expected to preserve backward compatibility, websites would have no interest in embracing these APIs.

This means that, other than *via* regulation and its enforcement, or with financial incentives, the ultimate privacy defense remains the user and lies with them as a consumer.

APPENDIX A

Appendices to the Breaking Bad Study

Table A.1: JavaScript reliance of standard HTML elements [185].
(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)	

Element	JavaScript reliance	JavaScript use cases
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)	
`<figure>`	(0)	
`<hr>`	(0)	
`<li>`	(0)	
`<ol>`	(0)	
`<p>`	(0)	
`<pre>`	(0)	
`<ul>`	(0)	
Table content		
`<caption>`	(0)	
`<col>`	(0)	
`<colgroup>`	(0)	
`<table>`	(0)	
`<tbody>`	(0)	
`<td>`	(0)	
`<tfoot>`	(0)	
`<th>`	(0)	
`<thead>`	(0)	
`<tr>`	(0)	

Element	JavaScript reliance	JavaScript use cases
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)	
`<em>`	(0)	
`<i>`	(0)	
`<kbd>`	(0)	
`<mark>`	(0)	
`<q>`	(0)	
`<rp>`	(0)	
`<rt>`	(0)	
`<ruby>`	(0)	
`<s>`	(0)	
`<samp>`	(0)	
`<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

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

Element	JavaScriptJavaScript use cases re-liance
`<textarea>`	(2*)

Table A.2: 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 [234]	Foundation 6 [277]	Tailwind Ele-ments [5]	Semantic UI [4]
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)

Component	Bootstrap 5	Foundation 6	Tailwind Elements	Semantic UI
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)
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

Component	Bootstrap 5	Foundation 6	Tailwind Elements	Semantic UI
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)
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

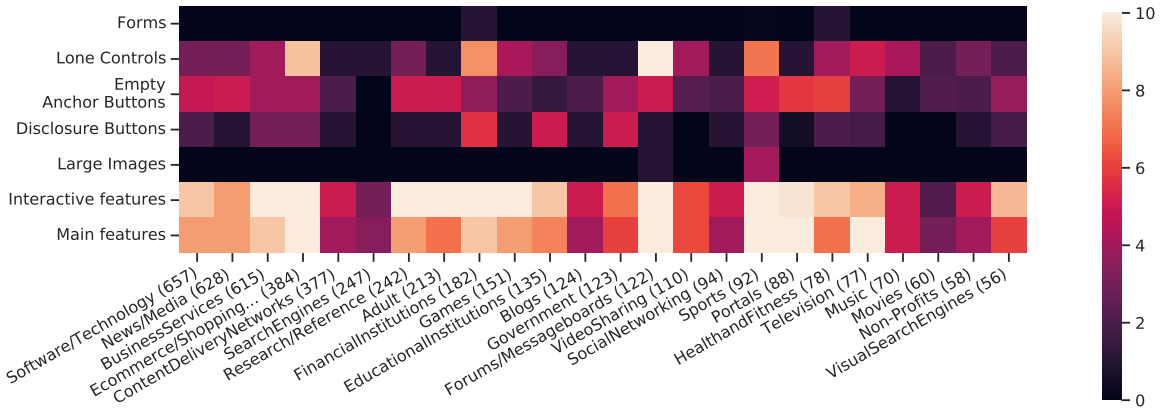


Figure A.1: Color represents the 90$^{\text{th}}$ percentile of differential breakage of visible elements in the *whole page*. Lighter shades denote higher differential breakage. Only categories with more than 50 pages in the dataset are plotted, to improve readability. This highlights the disparity of breakage across website categories.

APPENDIX B

Appendices to the UBI Study

```
{
  "lazyStylesheets": [
    {
      "target": "link[onload][rel='stylesheet'],
        link[media='only x'], link[media='none'],
        link[onload][as='style'],
        link[onload][type='type/css']",
      "applyToLoad": [
        { "prop": "rel", "value": "stylesheet" },
        { "prop": "media", "value": "all" },
        { "prop": "onload", "value": null }
      ]
    }
  ],
  "scriptLoaded": [
    {
      "target": "head > script",
      "regexp": "loadCSS\\s*\\(\\s*[\"'][^\"']+)[\"']",
      "usedBy": [
        "https://github.com/filamentgroup/loadCSS/"
      ]
    }
  ]
}
```

Listing B.1: Excerpt of the JSON repair configuration for 'lazy-loaded stylesheets'. Two types of lazy-loaded stylesheets can be handled: those using a `<link>` element and those using a script in the `<head>`. In both cases, the `target` property contains a CSS selector of elements to consider. In the former case, the `applyToLoad` property lists the IDL attributes to set for the stylesheets to be loaded. In the latter case, the script is parsed using a regular expression to extract the stylesheet URL to load, which is then loaded by injecting a `<link>` element.

```
{
  "fadeinElements": [
    {
      "target": "div:not(:empty), header, footer, section,
        article, main, .js-fadein, .js-fadein-hero,
        .fade, .fadein, .appear",
      "fadeinStyle": [
        { "prop": "opacity", "test": "===",
          "value": "0" },
        { "prop": "display", "test": "!==",
          "value": "none" },
        { "prop": "z-index", "test": "===",
          "value": "auto" },
        {
          "or": [
            {
              "prop": "transition", "test": "!==",
              "value": "all 0s ease 0s"
            },
            {
              "prop": "animation",
              "test": "!==",
              "value":
                "0s ease 0s 1 normal none running none"
            }
          ]
        }
      ],
      "showStyle": [
        { "prop": "opacity", "value": "1" },
        { "prop": "visibility", "value": "visible" }
      ]
    },
    {
      "target": ".fusion-animated, .elementor-invisible",
      "fadeinStyle": [
        { "prop": "visibility", "test": "===",
          "value": "hidden" }
      ],
      "showStyle": [
        { "prop": "visibility", "value": "visible" }
      ]
    }
  ]
}
```

Listing B.2: Excerpt of the JSON repair configuration for 'fade-in elements': the `target` property contains a CSS selector of elements to consider, which are then filtered using the computed style conditions found in the `fadeinStyle` property. The CSS styles encoded in the `showStyle` property are finally applied to the remaining elements.

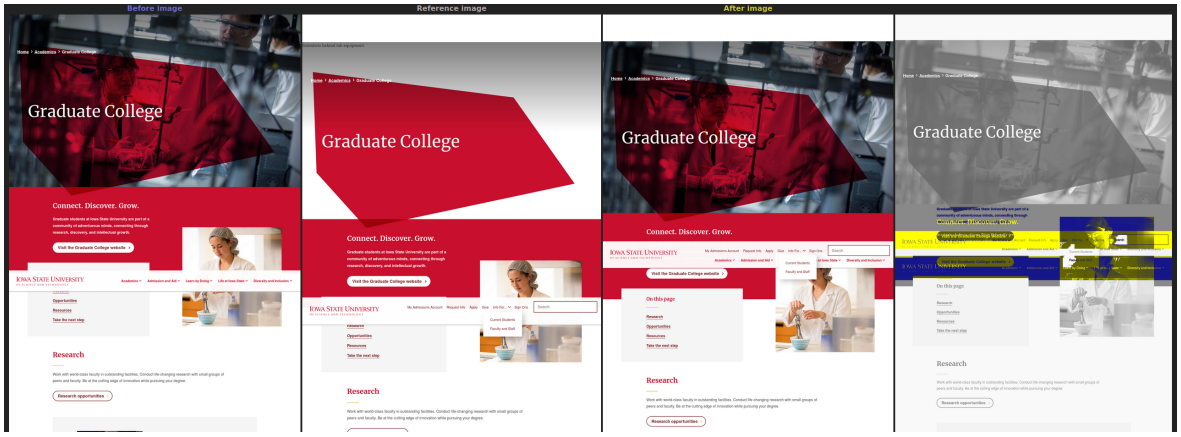


Figure B.1: Screenshot of the manual labeling tool (truncated vertically for readability), from left to right: the JS screenshot, NoJS screenshot, NoJS-UBI screenshot, and the color-coded difference image.

APPENDIX C

Appendices to the JSRehab Study

```
// Other imports
import posthtmlBootstrapNoscriptFallbacks
  from "posthtml-bootstrap-noscript-fallbacks";

const config = {
  // Other configuration
  module: {
    rules: [
      // Other rules
      {
        test: /\.html$/,
        use: [
          "extract-loader",
          {
            loader: "html-loader",
            options: { minimize: true },
          },
          {
            loader: "posthtml-loader",
            options: {
              plugins: [
                posthtmlBootstrapNoscriptFallbacks({
                  cssFiles: [
                    // Paths to stylesheets
                  ],
                }),
              ],
            },
          }
        ]
      },
    ],
  },
};
export default config;
```

Listing C.1: Example `webpack.config.js`

APPENDIX D

Appendices to the JsRipper Study

D.0.1 Frequency Analysis in Name Mangling

```
function components() {
  const element0 = document.createElement('div');
  element0.textContent = Math.random();

  const element1 = document.createElement('div');
  element1.textContent = Math.random();

  const element2 = document.createElement('div');
  element2.textContent = Math.random();

  return [element0, element1, element2];
}

for (const component of components()) {
  document.body.appendChild(component);
}
```

Listing D.1: Example script containing multiple local variables

```
(() => {
  for (const t of function() {
      const t = document.createElement("div");
      t.textContent = Math.random();
      const e = document.createElement("div");
      e.textContent = Math.random();
      const n = document.createElement("div");
      return n.textContent = Math.random(), [t, e, n]
    }()) document.body.appendChild(t)
})();
```

Listing D.2: Bundle produced by webpack, minified by Terser (prettified for readability), given the source code from Listing D.1: note the local variables `t`, `e`, `n`, used by Terser for being the most frequent letters in the original source code

```
function components() {
  const element0 = document.createElement('div');
  element0.textContent = Math.random();

  const element1 = document.createElement('div');
  element1.textContent = Math.random();

  const element2 = document.createElement('jjjjjjjjjjjj-jjjjjjjjjjjjjjjjj');
  element2.textContent = Math.random();

  return [element0, element1, element2];
}

for (const component of components()) {
  document.body.appendChild(component);
}
```

Listing D.3: Example source code where a long string of ‘j’ is introduced

```
(() => {
  for (const j of function() {
      const j = document.createElement("div");
      j.textContent = Math.random();
      const t = document.createElement("div");
      t.textContent = Math.random();
      const e = document.createElement("jjjjjjjjjjjj-jjjjjjjjjjjjjjjjj");
      return e.textContent = Math.random(), [j, t, e]
    }()) document.body.appendChild(j)
})();
```

Listing D.4: Bundle produced by webpack, minified by Terser (prettified for readability), given the source code from Listing D.3: note that now the letter ‘j’ is the most frequent letter and is thus used by Terser as the first local variable name

Table D.1: Tracking scripts whose tracking functions are most often bundled

Tracking script	Bundle count
https://weby.aaas.org/weby_bundle_v2.js	313
https://branch.io/js/all.js	251
https://d.mail.cafepress.com/track.v2.js	118
https://script.hotjar.com/modules.21c2ce197b1deec7582e.js	105
https://lightning.adultswim.com/launch/7be62238e4c3/22d196a3e151/launch-2fa6614adbd9.min.js	97
https://assets.adobedtm.com/e0903a2aaadb93ceed6a5acaaacbb9b9846eaa41/satelliteLib-88084863a26dad129e2d755e9777f20485407022.js	89
https://assets.adobedtm.com/dac62e20b491e735c6b56e64c39134d8ee93f9cf/satelliteLib-6b47f831c184878d7338d4683ecf773a17973bb9.js	80
https://cdn.exitbee.com/xtb.min.js	69
https://tags.tiqcdn.com/utag/autodesk/lib-target-flicker-free/prod/utag.sync.js	68
https://sc-static.net/scevent-gtm.min.js	68
https://a.pub.network/aljazeera-com/pubfig.min.js	62
https://cdn-akamai.mookie1.com/LB/LightningBolt.js	48
https://s3.amazonaws.com/cdn.aimtell.com/trackpush/trackpush-dev.min.js	38
https://www.ezojs.com/ezoic/sa.min.js	36
https://tags.news.com.au/prod/tad/tad.js	35
https://dumpster.cam4.com/v2/directory.js	28
https://a.pixiv.org/yufulight-cdn/apt.js	27
https://www.clickcease.com/monitor/stat.js	25
https://cdn.segment.com/analytics.js/v1/VvaFLFUpi8AdaC0xRvjHFUwuKSg3OxAe/analytics.min.js	16
https://cdn.optimizely.com/public/17026340012/s/square_enix_master.js	15
https://www.datadoghq-browser-agent.com/datadog-logs-v4.js	14
https://www.datadoghq-browser-agent.com/datadog-logs-us.js	11
https://cdn.jsdelivr.net/npm/sa-sdk-javascript@1.23.1/sensorsdata.min.js	11
https://matomo.1984.is/matomo.js	10
https://www.google-analytics.com/analytics.js	4
https://cdn.firstpromoter.com/fprom.js	3
https://static1.51ctocdn.cn/edu/sa-sdk-js/sensorsdata.min.js	2
https://static.sensorsdata.cn/sdk/1.12.5/sensorsdata.min.js	1
https://deadspin.com/x-kinja-static/assets/new-client/adManager.4d36048548b67ff27c83.js	1
https://plausible.io/js/plausible.js	1

Table D.2: Examples of scripts bundling tracking functions, these scripts are not part of filter lists

Scripts containing tracking functions	Known tracking scripts
https://tags.tiqcdn.com/utag/intuit/brand-icom/prod/utag.js	['https://dumpster.cam4.com/v2/directory.js']
https://www.sanofi.com/.resources/sanofi-lm-platform/themes/sanofi-platform/dist/common~2022-09-28-22-03-50-000~cache.js	['https://assets.adobedtm.com/e0903a2aaadb93ceed6a5acaaacbb9b9846eaa41/satelliteLib-88084863a26dad129e2d755e9777f20485407022.js']
https://assets.adobedtm.com/9aafaf1151ac/682bb4885835/launch-be686e885d32.min.js	['https://assets.adobedtm.com/dac62e20b491e735c6b56e64c39134d8ee93f9cf/satelliteLib-6b47f831c184878d7338d4683ecf773a17973bb9.js']
https://www.infoplease.com/sites/infoplease.com/files/js/js_sSle-sUI_xeB_8F5RW4Nq8rRQbKgBG9MDKvSNyHX8ww.js	['https://script.hotjar.com/modules.21c2ce197b1deec7582e.js']
https://www.grammy.com/_next/static/chunks/484-ed227be4db7efa1f.js	['https://branch.io/js/all.js']
https://s1cdn.vnecdn.net/vnexpress/restruct/j/v3797/v3/production/lazyload.js	['https://www.ezojs.com/ezoic/sa.min.js']
https://assets.adobedtm.com/331fbea29f79/a5b25a446515/launch-e80baf9c0255.min.js	['https://assets.adobedtm.com/e0903a2aaadb93ceed6a5acaaacbb9b9846eaa41/satelliteLib-88084863a26dad129e2d755e9777f20485407022.js']
https://assets.poetryfoundation.org/assets/scripts/vendors~main-9b6becb7b8.js	['https://script.hotjar.com/modules.21c2ce197b1deec7582e.js']
https://static.phemex.com/s/3rd/sd/sd-1.17.2.min.js	['https://cdn.jsdelivr.net/npm/sa-sdk-javascript@1.23.1/sensorsdata.min.js']
https://assets.adobedtm.com/8a93f8486ba4/5492896ad67e/launch-b1f76be4d2ee.min.js	['https://assets.adobedtm.com/dac62e20b491e735c6b56e64c39134d8ee93f9cf/satelliteLib-6b47f831c184878d7338d4683ecf773a17973bb9.js']
https://metamask.io/121cd9c2bdc4dd8c8ec9ead858719809d6d18de3-a80ac18016e9cb1f8728.js	['https://cdn.exitbee.com/xtb.min.js']

Bibliography

[1] 2013. *EasyList — About*. https://easylist.to/pages/about.html

[2] 2013. *WebKit Bugzilla — Optionally partition cache to prevent using cache for tracking*. https://bugs.webkit.org/show_bug.cgi?id=110269

[3] 2019. *MozillaWiki — Security/Tor Uplift*. https://wiki.mozilla.org/Security/Tor_Uplift

[4] 2019. *Semantic UI Documentation*. Retrieved 2021-03-17 from https://semantic-ui.com/introduction/getting-started.html

[5] 2021. *About Tailwind Elements*. Retrieved 2021-08-10 from https://tailwind-elements.com/

[6] 2021. *EasyPrivacy List*. Retrieved 2021-05-07 from https://easylist.to/easylist/easyprivacy.txt

[7] 2021. *NuxtJS Homepage*. Retrieved 2021-04-26 from https://nuxtjs.org/

[8] 2021. *PostHTML*. https://github.com/posthtml/posthtml

[9] 2021. *rollup.js*. https://rollupjs.org/guide/

[10] 2021. *Semantic UI*. https://semantic-ui.com/

[11] 2021. *Svelte Website*. Retrieved 2021-04-26 from https://svelte.dev/

[12] 2021. *The Web We Want — I want a CSS pseudo-selector for elements that are in the viewport*. https://webwewant.fyi/wants/63/

[13] 2021. *webpack*. https://webpack.js.org/

[14] 2022. *Breaking Bad: No-JS Breakage Detection Framework*. https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://gitlab.inria.fr/Spirals/breaking-bad.git&path=crawl_nojs/how-broken

[15] 2022. *Bugzilla — [meta] Support IndexedDB in Private Browsing Mode (with encrypted disk storage)*. https://bugzilla.mozilla.org/show_bug.cgi?id=1639542

[16] 2022. *Bugzilla — Support ServiceWorkers in Private Browsing Mode*. https://bugzilla.mozilla.org/show_bug.cgi?id=1320796

[17] 2022. *Make <label> elements reflect CSS pseudoclasses on associated form element.* https://github.com/w3c/csswg-drafts/issues/397

[18] 2022. *Parcel.* https://parceljs.org/

[19] 2022. *PublicWWW — "bootstrap.min.js".* Retrieved 2022-01-24 from https://publicwww.com/websites/%22bootstrap.min.js%22/

[20] 2022. *Rollup.* https://rollupjs.org/

[21] 2022. *uBlock Unbreak filters — uBlock Origin.* https://github.com/uBlockOrigin/uAssets/blob/master/filters/unbreak.txt

[22] 2022. *webpack.* https://webpack.js.org/

[23] 2023. *Breaking Bad: Web Crawler.* https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://gitlab.inria.fr/Spirals/breaking-bad.git&path=crawl_req

[24] 2023. *Browserify.* https://browserify.org/

[25] 2023. *Emotion.* https://emotion.sh/

[26] 2023. *JSRehab: Bootstrap Version Detection WebExtension.* https://gitlab.inria.fr/jsrehab/js-ui-framework-detection-webext

[27] 2023. *JSRehab: PostHTML Plugin Rewritting Interface Components with Noscript Alternatives.* https://gitlab.inria.fr/jsrehab/posthtml-bootstrap-noscript-fallbacks

[28] 2023. *JSRehab: Web Crawler.* https://gitlab.inria.fr/jsrehab/crawl-bootstrap

[29] 2023. *JsRipper: JavaScript Function Signature Generator.* https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://gitlab.inria.fr/Spirals/jsripper-artifact&path=jsripper/jsripper-function-processing

[30] 2023. *JsRipper: Proof of Concept Blocking WebExtension.* https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://gitlab.inria.fr/Spirals/jsripper-artifact&path=jsripper/jsripper-webext

[31] 2023. *JsRipper: Web Crawler.* https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://gitlab.inria.fr/Spirals/jsripper-artifact&path=crawl_jsripper

[32] 2023. *Rollup — Tree shaking.* https://rollupjs.org/introduction/#tree-shaking

[33] 2023. *Scope hoisting — Parcel.* https://parceljs.org/features/scope-hoisting/

[34] 2023. *styled-components.* https://styled-components.com/

[35] 2023. *Tailwind CSS — Rapidly build modern websites without ever leaving your HTML.* https://tailwindcss.com/

[36] 2023. *UBI: Repair WebExtension.* https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://gitlab.inria.fr/Spirals/privacy-ubi-artifact&path=ubi-a-webext

[37] 2023. *UBI: Screenshot Diffing Tool.* https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://gitlab.inria.fr/Spirals/privacy-ubi-artifact&path=browser-screenshot-diff

[38] 2023. *UBI: Screenshot Labeling Web-Based Tool.* https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://gitlab.inria.fr/Spirals/privacy-ubi-artifact&path=image-labeling

[39] 2023. *UBI: Web Crawler.* https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://gitlab.inria.fr/Spirals/privacy-ubi-artifact&path=crawl_ubi-a

[40] 2023. *Vue.js — SFC CSS Features.* https://vuejs.org/api/sfc-css-features.html

[41] 2023. *Webpack 5 release (2020-10-10) — CommonJs Tree Shaking.* https://webpack.js.org/blog/2020-10-10-webpack-5-release/#commonjs-tree-shaking

[42] 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

[43] Gunes Acar, Marc Juárez, Nick Nikiforakis, Claudia Díaz, Seda F. Gürses, Frank Piessens, and Bart Preneel. 2013. FPDetective: dusting the web for fingerprinters. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM, 1129–1140. https://doi.org/10.1145/2508859.2516674

[44] AdGuard. 2022. *AdGuard Knowledgebase — How to create your own ad filters.* https://kb.adguard.com/en/general/how-to-create-your-own-ad-filters

[45] Furkan Alaca and Paul C. van Oorschot. 2016. Device fingerprinting for augmenting web authentication: classification and analysis of methods. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, Stephen Schwab, William K. Robertson, and Davide Balzarotti (Eds.). ACM, 289–301. http://dl.acm.org/citation.cfm?id=2991091

[46] 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

[47] AliceWyman, Michele Rodaro, Joni, Marcelo Ghelman, Lamont Gardenhire, Jeff, Angela Lazar, PGGWriter, Samuelegrice@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

[48] Marshall Allen. 2018. *Health Insurers Are Vacuuming Up Details About You — And It Could Raise Your Rates*. https://www.propublica.org/article/health-insurers-are-vacuuming-up-details-about-you-and-it-could-raise-your-rates

[49] 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

[50] 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

[51] 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

[52] Web Almanac and contributors. 2022. *HTTP Archive Web Almanac — Markup*. Retrieved 2022-05-30 from https://almanac.httparchive.org/en/2021/markup#main

[53] Mshabab Alrizah, Sencun Zhu, Xinyu Xing, and Gang Wang. 2019. Errors, Misunderstandings, and Attacks: Analyzing the Crowdsourcing Process of Ad-blocking Systems. In *Proceedings of the Internet Measurement Conference, IMC 2019, Amsterdam, The Netherlands, October 21-23, 2019*. ACM, 230–244. https://doi.org/10.1145/3355369.3355588

[54] Abdul Haddi Amjad, Danial Saleem, Muhammad Ali Gulzar, Zubair Shafiq, and Fareed Zaffar. 2021. TrackerSift: untangling mixed tracking and functional web resources. In *IMC '21: ACM Internet Measurement Conference, Virtual Event, USA, November 2-4, 2021*, Dave Levin, Alan Mislove, Johanna Amann, and Matthew Luckie (Eds.). ACM, 569–576. https://doi.org/10.1145/3487552.3487855

[55] Abdul Haddi Amjad, Zubair Shafiq, and Muhammad Ali Gulzar. 2023. Blocking JavaScript without Breaking the Web: An Empirical Investigation. https://doi.org/10.48550/ARXIV.2302.01182

[56] Nampoina Andriamilanto, Tristan Allard, and Gaëtan Le Guelvouit. 2020. "Guess Who?" Large-Scale Data-Centric Study of the Adequacy of Browser Fingerprints for Web Authentication. In *Innovative Mobile and Internet Services in Ubiquitous Computing - Proceedings of the 14th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS-2020), Lodz, Poland, 1-3 July, 2020 (Advances in Intelligent Systems and Computing, Vol. 1195)*, Leonard Barolli, Aneta Poniszewska-Maranda, and Hyunhee Park (Eds.). Springer, 161–172. https://doi.org/10.1007/978-3-030-50399-4_16

[57] Nampoina Andriamilanto, Tristan Allard, Gaëtan Le Guelvouit, and Alexandre Garel. 2022. A Large-scale Empirical Analysis of Browser Fingerprints Properties for Web Authentication. *ACM Trans. Web* 16, 1 (2022), 4:1–4:62. https://doi.org/10.1145/3478026

[58] 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

[59] 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

[60] Andrés Arrieta, Bennett Cyphers, Alexei Miagkov, and Daly Barnett. 2020. *Electronic Frontier Foundation — Privacy Badger Is Changing to Protect You Better*. https://www.eff.org/deeplinks/2020/10/privacy-badger-changing-protect-you-better

[61] Babak Amin Azad, Oleksii Starov, Pierre Laperdrix, and Nick Nikiforakis. 2020. Web Runner 2049: Evaluating Third-Party Anti-bot Services. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24-26, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12223)*, Clémentine Maurice, Leyla Bilge, Gianluca Stringhini, and Nuno Neves (Eds.). Springer, 135–159. https://doi.org/10.1007/978-3-030-52683-2_7

[62] Pouneh Nikkhah Bahrami, Umar Iqbal, and Zubair Shafiq. 2022. FP-Radar: Longitudinal Measurement and Early Detection of Browser Fingerprinting. *Proc. Priv. Enhancing Technol.* 2022, 2 (2022), 557–577. https://doi.org/10.2478/popets-2022-0056

[63] Mihai Bazon and contributors. 2023. *terser*. https://github.com/terser/terser

[64] Tim Berners-Lee. 1992. *Request Headers in the HTTP protocol*. https://www.w3.org/Protocols/HTTP/HTRQ_Headers.html

[65] Sarah Bird, Ilana Segall, and Martin Lopatka. 2020. Replication: Why We Still Can't Browse in Peace: On the Uniqueness and Reidentifiability of Web Browsing Histories. In *Sixteenth Symposium on Usable Privacy and Security, SOUPS 2020, August 7-11, 2020*, Heather Richter Lipford and Sonia Chiasson (Eds.). USENIX Association, 489–503. https://www.usenix.org/conference/soups2020/presentation/bird

[66] Brave Software, Inc. 2022. *Brave Shields — FAQs*. https://brave.com/shields/#shields-faq

[67] Brave Software, Inc. 2023. *sugarcoat-resources Repository*. https://github.com/brave-experiments/sugarcoat-resources/tree/development

[68] Martin Brinkmann. 2013. *gHacks Tech News — Self-Destructing Cookies for Firefox*. https://www.ghacks.net/2013/01/15/self-destructing-cookies-for-firefox/

[69] Tomasz Bujlow, Valentín Carela-Español, Beom-Ryeol Lee, and Pere Barlet-Ros. 2017. A Survey on Web Tracking: Mechanisms, Implications, and Defenses. *Proc. IEEE* 105, 8 (2017), 1476–1510. https://doi.org/10.1109/JPROC.2016.2637878

[70] Darion Cassel, Su-Chin Lin, Alessio Buraggina, William Wang, Andrew Zhang, Lujo Bauer, Hsu-Chun Hsiao, Limin Jia, and Timothy Libert. 2022. OmniCrawl: Comprehensive Measurement of Web Tracking With Real Desktop and Mobile Browsers. *Proc. Priv. Enhancing Technol.* 2022, 1 (2022), 227–252. https://doi.org/10.2478/popets-2022-0012

[71] 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

[72] 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

[73] Quan Chen, Peter Snyder, Ben Livshits, and Alexandros Kapravelos. 2021. Detecting Filter List Evasion with Event-Loop-Turn Granularity JavaScript Signatures. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1715–1729. https://doi.org/10.1109/SP40001.2021.00007

[74] Chromium. 2021. *The Chromium Projects — SameSite Updates.* https://www.chromium.org/updates/same-site/

[75] Inc. Cloudflare. 2022. *What is Email Address Obfuscation? – Cloudflare Help Center.* https://support.cloudflare.com/hc/en-us/articles/200170016-What-is-Email-Address-Obfuscation-

[76] Mike Conca. 2020. *Mozilla Hacks — Changes to SameSite Cookie Behavior – A Call to Action for Web Developers.* https://hacks.mozilla.org/2020/08/changes-to-samesite-cookie-behavior/

[77] Easylist contributors. 2022. *Easylist Cookie List.* https://secure.fanboy.co.nz/fanboy-cookiemonster.txt

[78] MDN contributors. 2022. *MDN — Geolocation API.* https://developer.mozilla.org/en-US/docs/Web/API/Geolocation_API

[79] MDN contributors. 2022. *MDN — MediaDevices.getUserMedia().* https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia

[80] MDN contributors. 2022. *MDN — Origin.* https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Origin

[81] MDN contributors. 2022. *MDN — Storage Access API.* https://developer.mozilla.org/en-US/docs/Web/API/Storage_Access_API

[82] MDN contributors. 2022. *Subresource Integrity — Mozilla Developer Network.* https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity

[83] MDN contributors. 2022. *webRequest.filterResponseData() — Mozilla Developer Network*. webRequest.filterResponseData()

[84] MDN contributors. 2022. *webRequest.onBeforeRequest*. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest/onBeforeRequest

[85] MDN contributors. 2023. *MDN — ETag*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/ETag

[86] MDN contributors. 2023. *MDN — Using Service Workers*. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers.

[87] NuxtJS contributors. 2022. *The modern property*. https://nuxtjs.org/docs/configuration-glossary/configuration-modern/

[88] 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/

[89] Joseph Cox. 2021. *Vice — The NSA and CIA Use Ad Blockers Because Online Advertising Is So Dangerous*. https://www.vice.com/en/article/93ypke/the-nsa-and-cia-use-ad-blockers-because-online-advertising-is-so-dangerous

[90] 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/

[91] Savino Dambra, Iskander Sánchez-Rola, Leyla Bilge, and Davide Balzarotti. 2022. When Sally Met Trackers: Web Tracking From the Users' Perspective. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 2189–2206. https://www.usenix.org/conference/usenixsecurity22/presentation/dambra

[92] Amit Datta, Jianan Lu, and Michael Carl Tschantz. 2019. Evaluating Anti-Fingerprinting Privacy Enhancing Technologies. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia (Eds.). ACM, 351–362. https://doi.org/10.1145/3308558.3313703

[93] Statista Research Department. 2022. *Adblocking penetration rate in selected countries/territories worldwide as of 3rd quarter 2021 — Statista*. https://www.statista.com/statistics/351862/adblocking-usage/

[94] Chrome Developers. 2021. *Chrome Developers — Manifest V2 support timeline*. https://developer.chrome.com/docs/extensions/mv3/mv2-sunset/

[95] Chrome Developers. 2022. *Chrome Developers — chrome.declarativeNetRequest GUARANTEED_MINIMUM_STATIC_RULES*. https://developer.chrome.com/docs/extensions/reference/declarativeNetRequest/#property-GUARANTEED_MINIMUM_STATIC_RULES

[96] Yana Dimova, Gunes Acar, Lukasz Olejnik, Wouter Joosen, and Tom van Goethem. 2021. The CNAME of the Game: Large-scale Analysis of DNS-based Tracking Evasion. *Proc. Priv. Enhancing Technol.* 2021, 3 (2021), 394–412. https://doi.org/10.2478/popets-2021-0053

[97] Kenny Do and CAD Team. 2022. *GitHub — Cookie-AutoDelete*. https://github.com/Cookie-AutoDelete/Cookie-AutoDelete

[98] Donny and contributors. 2022. *swc_ecma_parser*. https://rustdoc.swc.rs/swc_ecma_parser/

[99] Antonin Durey, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. 2021. FP-Redemption: Studying Browser Fingerprinting Adoption for the Sake of Web Security. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 18th International Conference, DIMVA 2021, Virtual Event, July 14-16, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12756)*, Leyla Bilge, Lorenzo Cavallaro, Giancarlo Pellegrino, and Nuno Neves (Eds.). Springer, 237–257. https://doi.org/10.1007/978-3-030-80825-9_12

[100] Peter Eckersley. 2010. How Unique Is Your Web Browser?. In *Privacy Enhancing Technologies, 10th International Symposium, PETS 2010, Berlin, Germany, July 21-23, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6205)*, Mikhail J. Atallah and Nicholas J. Hopper (Eds.). Springer, 1–18. https://doi.org/10.1007/978-3-642-14527-8_1

[101] Steven Englehardt and Arthur Edelstein. 2021. *Mozilla Security Blog — Firefox 85 Cracks Down on Supercookies*. https://blog.mozilla.org/security/2021/01/26/supercookie-protections/

[102] Steven Englehardt and Arvind Narayanan. 2016. Online Tracking: A 1-million-site Measurement and Analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 1388–1401. https://doi.org/10.1145/2976749.2978313

[103] Steven Englehardt, Dillon Reisman, Christian Eubank, Peter Zimmerman, Jonathan R. Mayer, Arvind Narayanan, and Edward W. Felten. 2015. Cookies That Give You Away: The Surveillance Implications of Web Tracking. In *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, Aldo Gangemi, Stefano Leonardi, and Alessandro Panconesi (Eds.). ACM, 289–299. https://doi.org/10.1145/2736277.2741679

[104] ericlaw. 2019. *text/plain — Surprise: Undead Session Cookies*. https://textslashplain.com/2019/06/24/surprise-undead-session-cookies/

[105] fanboy, MonztA, Famlam, and Khrin. 2022. *EasyList*. https://easylist.to/easylist/easylist.txt

[106] Alexander Farkas. 2022. *lazysizes*. https://github.com/aFarkas/lazysizes

[107] Edward W. Felten and Michael A. Schneider. 2000. Timing attacks on Web privacy. In *CCS 2000, Proceedings of the 7th ACM Conference on Computer and Communications Security, Athens,*

Greece, November 1-4, 2000, Dimitris Gritzalis, Sushil Jajodia, and Pierangela Samarati (Eds.). ACM, 25–32. https://doi.org/10.1145/352600.352606

[108] Imane Fouad, Nataliia Bielova, Arnaud Legout, and Natasa Sarafijanovic-Djukic. 2020. Missed by Filter Lists: Detecting Unknown Third-Party Trackers with Invisible Pixels. *Proc. Priv. Enhancing Technol.* 2020, 2 (2020), 499–518. https://doi.org/10.2478/popets-2020-0038

[109] OpenJS Foundation and Node.js contributors. 2023. *Modules: CommonJS modules — Node.js v19.6.0 documentation.* https://nodejs.org/api/modules.html

[110] Romain Fouquet, Pierre Laperdrix, and Romain Rouvoy. 2022. JSRehab: Weaning Common Web Interface Components from JavaScript Addiction. In *Companion of The 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, 376–382. https://doi.org/10.1145/3487553.3524227

[111] Romain Fouquet, Pierre Laperdrix, and Romain Rouvoy. 2023. Breaking Bad: Quantifying the Addiction of Web Elements to JavaScript. *ACM Trans. Internet Technol.* 23, 1, Article 22 (feb 2023), 28 pages. https://doi.org/10.1145/3579846

[112] Gertjan Franken, Tom van Goethem, and Wouter Joosen. 2018. Who Left Open the Cookie Jar? A Comprehensive Evaluation of Third-Party Cookie Policies. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 151–168. https://www.usenix.org/conference/usenixsecurity18/presentation/franken

[113] Brent Fulgham. 2018. *WebKit — Protecting Against HSTS Abuse.* https://webkit.org/blog/8146/protecting-against-hsts-abuse/

[114] Barton Gellman, Andrea Peterson, and Ashkan Soltani. 2022. *The Washington Post — NSA uses Google cookies to pinpoint targets for hacking.* https://www.washingtonpost.com/news/the-switch/wp/2013/12/10/nsa-uses-google-cookies-to-pinpoint-targets-for-hacking/

[115] Barton Gellman and Laura Poitras. 2013. *The Washington Post — U.S., British intelligence mining data from nine U.S. Internet companies in broad secret program.* https://www.washingtonpost.com/investigations/us-intelligence-mining-data-from-nine-us-internet-companies-in-broad-secret-program/2013/06/06/3a0c0da8-cebf-11e2-8845-d970ccb04497_story.html

[116] Phillipa Gill, Vijay Erramilli, Augustin Chaintreau, Balachander Krishnamurthy, Konstantina Papagiannaki, and Pablo Rodriguez. 2013. Best paper - Follow the money: understanding economics of online aggregation and advertising. In *Proceedings of the 2013 Internet Measurement Conference, IMC 2013, Barcelona, Spain, October 23-25, 2013*, Konstantina Papagiannaki, P. Krishna Gummadi, and Craig Partridge (Eds.). ACM, 141–148. https://doi.org/10.1145/2504730.2504768

[117] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. 2018. Hiding in the Crowd: an Analysis of the Effectiveness of Browser Fingerprinting at Large Scale. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*, Pierre-Antoine Champin, Fabien Gandon, Mounia Lalmas, and Panagiotis G. Ipeirotis (Eds.). ACM, 309–318. https://doi.org/10.1145/3178876.3186097

[118] Google. 2018. *AngularJS Website*. Retrieved 2021-04-26 from https://angularjs.org/

[119] Google. 2019. *Making JavaScript and Google Search work together*. Retrieved 2021-05-06 from https://web.dev/javascript-and-google-search-io-2019/

[120] Google. 2019. *web.dev — Without JavaScript*. Retrieved 2021-04-26 from https://web.dev/without-javascript/

[121] Google. 2022. *Google Privacy Policy — Unique identifiers*. https://policies.google.com/privacy#footnote-unique-id

[122] Filament Group. 2020. *loadCSS Repository*. Retrieved 2021-04-20 from https://github.com/filamentgroup/loadCSS/

[123] W3C's Privacy Community Group. 2023. *Client-Side Storage Partitioning*. https://privacycg.github.io/storage-partitioning/

[124] 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

[125] 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

[126] HackerOne. 2020. *HackerOne — 4th Annual Hacker Powered Security Report*. https://www.hackerone.com/hacker-powered-security-report

[127] 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

[128] Raymond Hill. 2020. *uMatrix Repository*. Retrieved 2021-05-10 from https://github.com/gorhill/uMatrix

[129] Raymond Hill. 2021. *uBlock Origin Repository*. Retrieved 2021-05-10 from https://github.com/gorhill/uBlock/

[130] Raymond Hill and contributors. 2022. *GitHub gorhill/uBlock Wiki — Resources Library*. https://github.com/gorhill/uBlock/wiki/Resources-Library

[131] Raymond Hill and contributors. 2022. *GitHub gorhill/uBlock Wiki — Static filter syntax*. https://github.com/gorhill/uBlock/wiki/Static-filter-syntax

[132] Raymond Hill and contributors. 2022. *uBlock Origin Repository*. https://github.com/gorhill/uBlock/

[133] 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/

[134] Johann Hofmann and Tim Huang. 2021. *Mozilla Security Blog — Firefox 93 features an improved SmartBlock and new Referrer Tracking Protections*. https://blog.mozilla.org/security/2021/10/05/firefox-93-features-an-improved-smartblock-and-new-referrer-tracking-protections/

[135] IETF. 2011. *HTTP State Management Mechanism — Limits Model*. https://httpwg.org/specs/rfc6265.html#implementation-limits

[136] IETF. 2011. *HTTP State Management Mechanism — The Cookie Header*. https://httpwg.org/specs/rfc6265.html#cookie

[137] IETF. 2011. *HTTP State Management Mechanism — The Set-Cookie Header*. https://httpwg.org/specs/rfc6265.html#set-cookie

[138] Facebook Inc. 2021. *React Website*. Retrieved 2021-04-26 from https://reactjs.org/

[139] 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

[140] Umar Iqbal, Zubair Shafiq, and Zhiyun Qian. 2017. The ad wars: retrospective measurement and analysis of anti-adblock filter lists. In *Proceedings of the 2017 Internet Measurement Conference, IMC 2017, London, United Kingdom, November 1-3, 2017*, Steve Uhlig and Olaf Maennel (Eds.). ACM, 171–183. https://doi.org/10.1145/3131365.3131387

[141] Collin Jackson, Andrew Bortz, Dan Boneh, and John C. Mitchell. 2006. Protecting browser state from web privacy attacks. In *Proceedings of the 15th international conference on World Wide Web, WWW 2006, Edinburgh, Scotland, UK, May 23-26, 2006*, Les Carr, David De Roure, Arun Iyengar, Carole A. Goble, and Michael Dahlin (Eds.). ACM, 737–744. https://doi.org/10.1145/1135777.1135884

[142] Artur Janc, Krzysztof Kotowicz, Lukas Weichselbaum, and Roberto Clapis. 2020. Information Leaks via Safari's Intelligent Tracking Prevention. *CoRR* abs/2001.07421 (2020). arXiv:2001.07421 https://arxiv.org/abs/2001.07421

[143] Artur Janc and Lukasz Olejnik. 2010. Web Browser History Detection as a Real-World Privacy Threat. In *Computer Security - ESORICS 2010, 15th European Symposium on Research in*

Computer Security, Athens, Greece, September 20-22, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6345), Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou (Eds.). Springer, 215–231. https://doi.org/10.1007/978-3-642-15497-3_14

[144] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2010. An empirical study of privacy-violating information flows in JavaScript web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov (Eds.). ACM, 270–283. https://doi.org/10.1145/1866307.1866339

[145] 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

[146] Timofey Kachalov. 2022. *JavaScript Obfuscator Tool.* https://obfuscator.io/

[147] Rahul Kashyap. 2014. *Wired — Why Malvertising Is Cybercriminals' Latest Sweet Spot.* https://www.wired.com/insights/2014/11/malvertising-is-cybercriminals-latest-sweet-spot/

[148] 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

[149] kkapsner and contributors. 2022. *GitHub — CanvasBlocker.* https://github.com/kkapsner/CanvasBlocker/

[150] Takashi Kokubun. 2021. *GitHub Ranking — Repositories Ranking.* Retrieved 2021-08-11 from https://gitstar-ranking.com/repositories

[151] Balachander Krishnamurthy and Craig E. Wills. 2009. Privacy diffusion on the web: a longitudinal perspective. In *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*, Juan Quemada, Gonzalo León, Yoëlle S. Maarek, and Wolfgang Nejdl (Eds.). ACM, 541–550. https://doi.org/10.1145/1526709.1526782

[152] David M. Kristol. 2001. HTTP Cookies: Standards, privacy, and politics. *ACM Trans. Internet Techn.* 1, 2 (2001), 151–198. https://doi.org/10.1145/502152.502153

[153] Malwarebytes Labs. 2016. *Large Angler Malvertising Campaign Hits Top Publishers.* https://www.malwarebytes.com/blog/news/2016/03/large-angler-malvertising-campaign-hits-top-publishers

[154] Tomer Laor, Naif Mehanna, Antonin Durey, Vitaly Dyadyuk, Pierre Laperdrix, Clémentine Maurice, Yossi Oren, Romain Rouvoy, Walter Rudametkin, and Yuval Yarom. 2022. DRAWN

APART: A Device Identification Technique based on Remote GPU Fingerprinting. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/auto-draft-242/

[155] Pierre Laperdrix, Gildas Avoine, Benoit Baudry, and Nick Nikiforakis. 2019. Morellian Analysis for Browsers: Making Web Authentication Stronger with Canvas Fingerprinting. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11543)*, Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren (Eds.). Springer, 43–66. https://doi.org/10.1007/978-3-030-22038-9_3

[156] Pierre Laperdrix, Benoit Baudry, and Vikas Mishra. 2017. FPRandom: Randomizing Core Browser Objects to Break Advanced Device Fingerprinting Techniques. In *Engineering Secure Software and Systems - 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10379)*, Eric Bodden, Mathias Payer, and Elias Athanasopoulos (Eds.). Springer, 97–114. https://doi.org/10.1007/978-3-319-62105-0_7

[157] 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

[158] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. 2016. Beauty and the Beast: Diverting Modern Web Browsers to Build Unique Browser Fingerprints. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 878–894. https://doi.org/10.1109/SP.2016.57

[159] 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, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2507–2524. https://www.usenix.org/conference/usenixsecurity21/presentation/laperdrix

[160] Hieu Le, Salma Elmalaki, Athina Markopoulou, and Zubair Shafiq. 2022. AutoFR: Automated Filter Rule Generation for Adblocking. *CoRR* abs/2202.12872 (2022). arXiv:2202.12872 https://arxiv.org/abs/2202.12872

[161] Hieu Le, Athina Markopoulou, and Zubair Shafiq. 2021. CV-Inspector: Towards Automating Detection of Adblock Circumvention. 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/cv-inspector-towards-automating-detection-of-adblock-circumvention/

[162] Dimi Lee and Christoph Kerschbaumer. 2021. *Mozilla Security Blog — Firefox 87 trims HTTP Referrers by default to protect user privacy.* https://blog.mozilla.org/security/2021/03/22/firefox-87-trims-http-referrers-by-default-to-protect-user-privacy/

[163] Kevin K. Lee. 2022. *Chrome Developers — Storage Partitioning*. https://developer.chrome.com/docs/privacy-sandbox/storage-partitioning/

[164] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. 2016. Internet Jones and the Raiders of the Lost Trackers: An Archaeological Study of Web Tracking from 1996 to 2016. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lerner

[165] 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

[166] Bin Liang, Wei You, Liangkun Liu, Wenchang Shi, and Mario Heiderich. 2014. Scriptless Timing Attacks on Web Browser Privacy. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*. IEEE Computer Society, 112–123. https://doi.org/10.1109/DSN.2014.93

[167] Su-Chin Lin, Kai-Hsiang Chou, Yen Chen, Hsu-Chun Hsiao, Darion Cassel, Lujo Bauer, and Limin Jia. 2022. Investigating Advertisers' Domain-changing Behaviors and Their Impacts on Ad-blocker Filter Lists. 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, 576–587. https://doi.org/10.1145/3485447.3512218

[168] Snyk Ltd. 2022. *bootstrap vulnerabilities | Snyk*. Retrieved 2022-03-17 from https://snyk.io/vuln/npm%3Abootstrap

[169] Justin Ma, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. 2009. Beyond blacklists: learning to detect malicious web sites from suspicious URLs. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Paris, France, June 28 - July 1, 2009*, John F. Elder IV, Françoise Fogelman-Soulié, Peter A. Flach, and Mohammed Javeed Zaki (Eds.). ACM, 1245–1254. https://doi.org/10.1145/1557019.1557153

[170] Mack. 2005. *Macriot — Apple Safari, Private Browsing*. https://macriot.com/mcrt/2005/10/09/apple-safari-private-browsing/

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

[172] Giorgio Maone Maone and contributors. 2022. *NoScript Repository*. https://github.com/hackademix/noscript

[173] Alex Marthews and Catherine E. Tucker. 2017. Government Surveillance and Internet Search Behavior. *Berkeley Tech. L.J.. Berkeley Technology Law Journal* (2017).

[174] Antonio García Martinez. 2018. *How Trump Conquered Facebook—Without Russian Ads*. https://www.wired.com/story/how-trump-conquered-facebookwithout-russian-ads/

[175] Matomo. 2022. *Matomo Analytics Platform — Import Google Search keywords in Matomo*. https://matomo.org/faq/reports/import-google-search-keywords-in-matomo/

[176] Jonathan R. Mayer and John C. Mitchell. 2012. Third-Party Web Tracking: Policy and Technology. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*. IEEE Computer Society, 413–427. https://doi.org/10.1109/SP.2012.47

[177] Georg Merzdovnik, Markus Huber, Damjan Buhov, Nick Nikiforakis, Sebastian Neuner, Martin Schmiedecker, and Edgar R. Weippl. 2017. Block Me If You Can: A Large-Scale Study of Tracker-Blocking Tools. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*. IEEE, 319–333. https://doi.org/10.1109/EuroSP.2017.26

[178] Vikas Mishra, Pierre Laperdrix, Antoine Vastel, Walter Rudametkin, Romain Rouvoy, and Martin Lopatka. 2020. Don't Count Me Out: On the Relevance of IP Address in the Tracking Ecosystem. In *Proceedings of The Web Conference 2020* (Taipei, Taiwan) *(WWW '20)*. Association for Computing Machinery, New York, NY, USA, 808–815. https://doi.org/10.1145/3366423.3380161

[179] Marvin Moog, Markus Demmel, Michael Backes, and Aurore Fass. 2021. Statically Detecting JavaScript Obfuscation and Minification Techniques in the Wild. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021*. IEEE, 569–580. https://doi.org/10.1109/DSN48987.2021.00065

[180] Keaton Mowery and Hovav Shacham. 2012. Pixel Perfect: Fingerprinting Canvas in HTML5. In *Proceedings of W2SP 2012*, Matt Fredrikson (Ed.). IEEE Computer Society.

[181] Mozilla. 2006. *Mozilla Firefox 2 Release Notes*. https://website-archive.mozilla.org/www.mozilla.org/firefox_releasenotes/en-us/firefox/2.0/releasenotes/

[182] Mozilla. 2009. *Mozilla Firefox 3.5 Release Notes*. https://website-archive.mozilla.org/www.mozilla.org/firefox_releasenotes/en-us/firefox/3.5/releasenotes/

[183] Mozilla. 2022. *Firefox rolls out Total Cookie Protection by default to all users worldwide*. https://blog.mozilla.org/en/mozilla/firefox-rolls-out-total-cookie-protection-by-default-to-all-users-worldwide/

[184] Mozilla. 2022. *mozsearch — Firefox ETP shims*. https://searchfox.org/mozilla-central/source/browser/extensions/webcompat/shims

[185] 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

[186] 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

[187] 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

[188] mozilla.org contributors. 2022. *Enhanced Tracking Protection in Firefox for desktop*. https://support.mozilla.org/en-US/kb/enhanced-tracking-protection-firefox-desktop

[189] mozilla.org contributors. 2022. *Firefox Help — Multi-Account Containers*. https://support.mozilla.org/en-US/kb/containers

[190] mozilla.org contributors. 2022. *webRequest.onBeforeRequest — urlClassification*. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest/onBeforeRequest#urlclassification

[191] MyBrowserAddon. 2022. *Canvas Fingerprint Defender :: MyBrowserAddon*. https://mybrowseraddon.com/canvas-defender.html

[192] Netcraft. 2022. *Netcraft News — December 2022 Web Server Survey*. https://news.netcraft.com/archives/2022/12/20/december-2022-web-server-survey.html

[193] 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/

[194] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. 2015. PriVaricator: Deceiving Fingerprinters with Little White Lies. In *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, Aldo Gangemi, Stefano Leonardi, and Alessandro Panconesi (Eds.). ACM, 820–830. https://doi.org/10.1145/2736277.2741090

[195] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2013. Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, 541–555. https://doi.org/10.1109/SP.2013.43

[196] npm, Inc. 2023. *npm*. https://www.npmjs.com/

[197] Lukasz Olejnik, Claude Castelluccia, and Artur Janc. 2012. Why Johnny Can't Browse in Peace: On the Uniqueness of Web Browsing History Patterns. In *5th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2012)*. Vigo, Spain. https://hal.inria.fr/hal-00747841

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

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

[200] Jonathon W. Penney. 2016. Chilling Effects: Online Surveillance and Wikipedia Use. *Berkeley Tech. L.J.. Berkeley Technology Law Journal* 31, IR (2016), 117. http://lawcat.berkeley.edu/record/1127413

[201] Mike Perry, Erinn Clark, Steven Murdoch, and Georg Koppen. 2019. *The Design and Implementation of the Tor Browser [DRAFT] — 4.6. Cross-Origin Fingerprinting Unlinkability.* https://2019.www.torproject.org/projects/torbrowser/design/#fingerprinting-linkability

[202] Adblock Plus. 2022. *Adblock Plus — How to write filters.* https://help.adblockplus.org/hc/en-us/articles/360062733293

[203] 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/

[204] Victor Le Pochat, Tom van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019.* The Internet Society. https://www.ndss-symposium.org/ndss-paper/tranco-a-research-oriented-top-sites-ranking-hardened-against-manipulation/

[205] PortSwigger. 2022. *Web Security Academy Cross-site request forgery (CSRF) — Referer-based defenses against CSRF.* https://portswigger.net/web-security/csrf#referer-based-defenses-against-csrf

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

[207] 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

[208] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. 2012. Detecting and Defending Against Third-Party Tracking on the Web. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, Steven D. Gribble and Dina Katabi (Eds.). USENIX Association, 155–168. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/roesner

[209] 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

[210] Eleanor Roosevelt. 1948. *Universal Declaration of Human Rights.* https://www.un.org/en/about-us/universal-declaration-of-human-rights

[211] 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/

[212] Iskander Sánchez-Rola and Igor Santos. 2018. Knockin' on Trackers' Door: Large-Scale Automatic Analysis of Web Tracking. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10885)*, Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc (Eds.). Springer, 281–302. https://doi.org/10.1007/978-3-319-93411-2_13

[213] 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

[214] Justin Schuh. 2020. *Chromium Blog: Building a more private web: A path towards making third party cookies obsolete.* https://blog.chromium.org/2020/01/building-more-private-web-path-towards.html

[215] Ryan Seddon. 2010. *Custom radio and checkbox inputs using CSS.* Retrieved 2022-03-04 from https://ryanseddon.com/css/custom-inputs-using-css/

[216] Amazon Web Services. 2022. *Amazon CloudFront.* https://aws.amazon.com/cloudfront/features/

[217] Suphannee Sivakorn. 2016. I'm Not a Human: Breaking the Google reCAPTCHA.

[218] 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

[219] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. 2019. Anything to Hide? Studying Minified and Obfuscated Code in the Web. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia (Eds.). ACM, 1735–1746. https://doi.org/10.1145/3308558.3313752

[220] Michael Smith, Peter Snyder, Benjamin Livshits, and Deian Stefan. 2021. SugarCoat: Programmatically Generating Privacy-Preserving, Web-Compatible Resource Replacements for Content Blocking. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 2844–2857. https://doi.org/10.1145/3460120.3484578

[221] Peter Snyder, Cynthia Bagier Taylor, and Chris Kanich. 2017. Most Websites Don't Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security. In *Proceedings of the 2017*

ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 179–194. https://doi.org/10.1145/3133956.3133966

[222] 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

[223] Konstantinos Solomos, Panagiotis Ilia, Soroush Karami, Nick Nikiforakis, and Jason Polakis. 2022. The Dangers of Human Touch: Fingerprinting Browser Extensions through User Actions. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 717–733. https://www.usenix.org/conference/usenixsecurity22/presentation/solomos

[224] Konstantinos Solomos, Panagiotis Ilia, Nick Nikiforakis, and Jason Polakis. 2022. Escaping the Confines of Time: Continuous Browser Extension Fingerprinting Through Ephemeral Modifications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 2675–2688. https://doi.org/10.1145/3548606.3560576

[225] 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/

[226] Oleksii Starov, Pierre Laperdrix, Alexandros Kapravelos, and Nick Nikiforakis. 2019. Unnecessarily Identifiable: Quantifying the fingerprintability of browser extensions due to bloat. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia (Eds.). ACM, 3244–3250. https://doi.org/10.1145/3308558.3313458

[227] StatCounter. 2022. *statcounter — Mobile Browser Market Share Worldwide*. Retrieved 2022-01-24 from https://gs.statcounter.com/browser-market-share/mobile/worldwide

[228] StrongLoop, IBM, and other expressjs.com contributors. 2021. *Express - Node.js web application framework*. https://expressjs.com/

[229] Danny Sullivan. 2013. *Search Engine Land — Post-PRISM, Google Confirms Quietly Moving To Make All Searches Secure, Except For Ad Clicks*. https://searchengineland.com/post-prism-google-secure-searches-172487

[230] Erik Sy, Christian Burkert, Hannes Federrath, and Mathias Fischer. 2018. Tracking Users across the Web via TLS Session Resumption. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 289–299. https://doi.org/10.1145/3274694.3274708

[231] Bootstrap team. 2021. *Bootstrap 4 — Collapse, Accordion example.* Retrieved 2021-10-20 from https://getbootstrap.com/docs/4.6/components/collapse/#accordion-example

[232] Bootstrap team. 2021. *Bootstrap 5 — Components.* Retrieved 2021-10-19 from https://getbootstrap.com/docs/5.1/components/

[233] Bootstrap team and contributors. 2021. *Bootstrap Documentation — Accordion.* Retrieved 2021-10-25 from https://getbootstrap.com/docs/5.1/components/accordion/

[234] Bootstrap team and contributors. 2021. *Bootstrap Documentation — Components.* Retrieved 2021-10-25 from https://getbootstrap.com/docs/5.1/getting-started/introduction/#components

[235] Bootstrap team and contributors. 2021. *Bootstrap Documentation — Dropdowns.* Retrieved 2021-10-25 from https://getbootstrap.com/docs/5.1/components/dropdowns/

[236] Bootstrap team and contributors. 2021. *Bootstrap Homepage.* Retrieved 2021-04-25 from https://getbootstrap.com/

[237] Bootstrap team and contributors. 2021. *Bootstrap JavaScript.* Retrieved 2021-04-26 from https://getbootstrap.com/docs/5.1/getting-started/javascript/

[238] Brave Privacy Team. 2021. *Brave Browser — Ephemeral third-party site storage.* https://brave.com/privacy-updates/7-ephemeral-storage/

[239] Brave Privacy Team. 2021. *Brave Browser — Partitioning network-state for privacy.* https://brave.com/privacy-updates/14-partitioning-network-state/

[240] The Contributors to the User-Agent Client Hints Specification. 2022. *User-Agent Client Hints.* https://wicg.github.io/ua-client-hints/

[241] Minh Tran, Xinshu Dong, Zhenkai Liang, and Xuxian Jiang. 2012. Tracking the Trackers: Fast and Scalable Dynamic Analysis of Web Content for Privacy Violations. In *Applied Cryptography and Network Security - 10th International Conference, ACNS 2012, Singapore, June 26-29, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7341)*, Feng Bao, Pierangela Samarati, and Jianying Zhou (Eds.). Springer, 418–435. https://doi.org/10.1007/978-3-642-31284-7_25

[242] 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

[243] Can I use.... 2022. *Can I use... IndexedDB.* https://caniuse.com/indexeddb

[244] Can I use.... 2022. *Can I use... Window API: sessionStorage.* https://caniuse.com/mdn-api_window_sessionstorage

[245] Can I use.... 2023. *Can I use... 'SameSite' cookie attribute.* https://caniuse.com/same-site-cookie-attribute

[246] 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

[247] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. 2018. FP-STALKER: Tracking Browser Fingerprint Evolutions. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 728–741. https://doi.org/10.1109/SP.2018.00008

[248] Vercel. 2022. *Next.js Compiler.* https://nextjs.org/docs/advanced-features/compiler

[249] Vercel and contributors. 2023. *Turbopack — The successor to webpack.* https://turbo.build/pack

[250] Tanvi Vyas. 2017. *Tanvi Vyas — An Update on Firefox Containers.* https://blog.mozilla.org/tanvi/2017/10/03/update-firefox-containers/

[251] 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

[252] W3C. 2019. *W3C Tracking Preference Expression (DNT) — Activity.* https://www.w3.org/TR/tracking-dnt/#terminology.activity

[253] W3C. 2021. *Accessible Rich Internet Applications (WAI-ARIA) 1.3 — WAI-ARIA States and Properties.* https://w3c.github.io/aria/#introstates

[254] W3C. 2021. *ARIA Specification — Design Patterns and Widgets.* Retrieved 2021-03-17 from https://w3c.github.io/aria-practices/#aria_ex

[255] W3C. 2021. *Selectors Level 4.* https://drafts.csswg.org/selectors-4/#overview

[256] W3C. 2021. *Selectors Level 4.* https://drafts.csswg.org/selectors/#relational

[257] W3C. 2021. *Web Accessibility Laws & Policies.* https://www.w3.org/WAI/policies/

[258] W3C. 2021. *Web Content Accessibility Guidelines (WCAG) 2.1 — Success Criterion 2.1.1 Keyboard.* https://w3c.github.io/wcag21/guidelines/#keyboard

[259] Jeremy Wagner. 2022. *How much JavaScript do we load? — Web Almanac.* https://almanac.httparchive.org/en/2022/javascript#how-much-javascript-do-we-load

[260] Jeremy Wagner. 2022. *JavaScript Bundlers — Web Almanac.* https://almanac.httparchive.org/en/2022/javascript#bundlers

[261] Evan Wallace and contributors. 2023. *esbuild — An extremely fast bundler for the web.* https://esbuild.github.io/

[262] WHATWG. 2021. *DOM Standard.* https://dom.spec.whatwg.org/#concept-tree-order

[263] WHATWG. 2021. *HTML Specification — Custom data attribute.* Retrieved 2021-03-17 from https://html.spec.whatwg.org/#custom-data-attribute

[264] WHATWG. 2021. *HTML Specification — Implicit submission.* Retrieved 2021-03-18 from https://html.spec.whatwg.org/multipage/form-control-infrastructure.html#implicit-submission

[265] 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

[266] WHATWG. 2022. *HTML Living Standard — Lazy loading attributes.* https://html.spec.whatwg.org/#lazy-loading-attributes

[267] WHATWG. 2022. *HTML Living Standard — The noscript element.* https://html.spec.whatwg.org/#the-noscript-element

[268] John Wilander. 2017. *WebKit — Intelligent Tracking Prevention.* https://webkit.org/blog/7675/intelligent-tracking-prevention/

[269] John Wilander. 2018. *WebKit — Intelligent Tracking Prevention 1.1.* https://webkit.org/blog/8142/intelligent-tracking-prevention-1-1/

[270] John Wilander. 2018. *WebKit — Intelligent Tracking Prevention 2.0.* https://webkit.org/blog/8311/intelligent-tracking-prevention-2-0/

[271] John Wilander. 2019. *WebKit — Intelligent Tracking Prevention 2.1.* https://webkit.org/blog/8613/intelligent-tracking-prevention-2-1/

[272] John Wilander. 2020. *WebKit — Full Third-Party Cookie Blocking and More.* https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more/

[273] Craig E. Wills and Can Tatar. 2012. Understanding what they do with what they know. In *Proceedings of the 11th annual ACM Workshop on Privacy in the Electronic Society, WPES 2012, Raleigh, NC, USA, October 15, 2012*, Ting Yu and Nikita Borisov (Eds.). ACM, 13–18. https://doi.org/10.1145/2381966.2381969

[274] Gilbert Wondracek, Thorsten Holz, Engin Kirda, and Christopher Kruegel. 2010. A Practical Attack to De-anonymize Social Network Users. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 223–238. https://doi.org/10.1109/SP.2010.21

[275] 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

[276] Foundation Yetinauts. 2021. *Foundation — The most advanced responsive front-end framework in the world.* https://foundation.zurb.com/

[277] Foundation Yetinauts. 2021. *Foundation Documentation*. Retrieved 2021-03-17 from https://get.foundation/sites/docs/

[278] Evan You and contributors. 2021. *Vue.js Website*. Retrieved 2021-04-26 from https://vuejs.org/

[279] Eric Zeng, Rachel McAmis, Tadayoshi Kohno, and Franziska Roesner. 2022. What factors affect targeting and bids in online advertising?: a field measurement study. In *Proceedings of the 22nd ACM Internet Measurement Conference, IMC 2022, Nice, France, October 25-27, 2022*, Chadi Barakat, Cristel Pelsser, Theophilus A. Benson, and David Choffnes (Eds.). ACM, 210–229. https://doi.org/10.1145/3517745.3561460