Unnecessarily Identifiable: Quantifying the fingerprintability of browser extensions due to bloat

Oleksii Starov[†], Pierre Laperdrix[†], Alexandros Kapravelos[‡], Nick Nikiforakis[†]
†Stony Brook University
‡North Carolina State University

ABSTRACT

In this paper, we investigate to what extent the page modifications that make browser extensions fingerprintable are necessary for their operation. We characterize page modifications that are completely unnecessary for the extension's functionality as *extension bloat*. By analyzing 58,034 extensions from the Google Chrome store, we discovered that 5.7% of them were unnecessarily identifiable because of extension bloat. To protect users against unnecessary extension fingerprinting due to bloat, we describe the design and implementation of an in-browser mechanism that provides coarse-grained access control for extensions on all websites. The proposed mechanism and its built-in policies, does not only protect users from fingerprinting, but also offers additional protection against malicious extensions exfiltrating user data from sensitive websites.

ACM Reference Format:
Oleksii Starov[†], Pierre Laperdrix[†], Alexandros Kapravelos[‡], Nick Nikiforakis[†]. 2019. Unnecessarily Identifiable: Quantifying the fingerprintability of browser extensions due to bloat. In *Proceedings of the 2019 World Wide Web Conference (WWW'19), May 13–17, 2019, San Francisco, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3308558.3313458

1 INTRODUCTION

The past two decades have witnessed the web expanding and evolving at a tremendous rate with a similar expansion of the devices that users utilize to browse the web. However, this booming diversity of devices brought with it a new tracking technique called browser fingerprinting [8, 10–13, 16, 17]. Browser extensions are a recent addition to the fingerprinting domain [14, 18, 20, 23]. In order to provide added functionality, many extensions require full access to the pages that users visit. Some extensions modify existing elements from the Document Object Model (DOM) of webpages while others create new ones. Interacting with the DOM has the unfortunate consequence of causing side-effects which can later be detected and attributed back to installed extensions. In previous work, we showed that the presence of many browser extensions can be inferred just by looking at the modifications that they perform to the DOM [23]. Any malicious script running on the same webpage can then try to fingerprint the list of extensions installed in the browser. This poses a serious privacy risk as extension fingerprintability can lead to the identification of a user.

WWW '19, May 13–17, 2019, San Francisco, CA, USA

ACM ISBN 978-1-4503-6674-8/19/05.
https://doi.org/10.1145/3308558.3313458

Fixing the problem of fingerprintable browser extensions is not straightforward. Browser extensions are intended to access and modify webpages that users are visiting and, in many cases, this is their main and desired functionality (e.g., hiding ads, highlighting search results, or enlarging photo thumbnails). At the same time, the question remains whether such functional side effects need to manifest on all pages unconditionally. For example, if a webpage does not contain ads, search results, photos, or other triggering content, it is completely unnecessary for an extension to disclose its presence since it will be hurting the privacy of the user while delivering no useful functionality.

In this paper, we investigate whether the page modifications that make extensions fingerprintable are necessary for their operation. We define the notion of *bloat* in the implementation logic of browser extensions as the unnecessary side effects that offer no functionality to users and can be abused for extension identification. By analyzing 58,034 extensions from the Google Chrome store, we identify the behaviors characteristic of bloat and quantify its prevalence in the wild. We study the origin of fingerprintable bloat and discover cases of shared libraries and common coding practices, which contribute to it. Moreover, we show that bloat is responsible for a large fraction of all fingerprintable on-page changes from extensions.

Orthogonally to our measurement and characterization of extension bloat, we argue that it is important for users to have client-side control over the reach of browser extensions and their interactions with the DOM of webpages. We provide the design and implementation details of a client-side access control mechanism that mitigates the aforementioned privacy risks that derive from bloat by empowering users to selectively enable/disable extensions on arbitrary websites. Moreover, we discuss how the same client-side access-control mechanism protects against general privacy risks of untrustworthy browser extensions, i.e., limiting or altogether stopping the leakage of browsing history and other sensitive on-page information by buggy and malicious extensions.

2 BLOAT-RELATED FINGERPRINTABILITY OF BROWSER EXTENSIONS

Software bloat refers to software including unnecessary code and libraries while the process of *debloating* denotes the removal of the excess components. For this paper, we focus on software bloat in terms of unnecessary implementation logic inside browser extensions, which results in them being unnecessarily fingerprintable. We use the following definition:

Definition. *Bloat-related side effects of browser extensions are artifacts on a web page which do not deliver functionality that is desired by users, yet reveal the extension's presence to trackers and fingerprinting scripts.*

2.1 Types of bloat

When the user visits a page with an installed extension, both the extension and the page have access to the same Document Object Model (DOM), which provides a programmatic interface of the page to JavaScript code. The extension can add or remove elements from the DOM of the visited page by injecting JavaScript. The core observation of our definition with regard to extension bloat, is that an extension should not modify the DOM of the visited page, unless it is absolutely necessary for its functionality. Unnecessary DOM modifications by the extension disclose its presence to visited pages and therefore to trackers situated on those pages.

Some extensions will perform changes on the visited page that are part of their intended functionality. For example, static panels filled with information (e.g., showing website statistics), toolbars with URL-independent tools (e.g., CSS inspectors), and modified text (e.g., extensions to correct the spelling of words) are all intentional. One may notice that the common feature of all of these examples is the presence of useful functionality, even if that functionality manifests on all pages.

To identify cases of true extension bloat, i.e., modifications that are unnecessary since they do not provide any useful function, we manually analyzed the types of changes that extensions perform on an empty page and their purpose. Our intuition was that an empty webpage has no existing content to trigger the logic of browser extensions. As such, any DOM modifications are potentially the result of bloat and should be investigated. Through this manual experiment we identify the following extension-independent categories of unnecessary DOM modifications:

•**Injected empty placeholders**. We discovered that it is a common practice among many extensions to include empty or invisible DOM elements first, which will be filled with data later based on a trigger (e.g., when specific content appears on a page, or when a user clicks the extension's icon) (Figure 1). The direct consequence of this preemptive DOM modification is that these placeholders unnecessarily increase the fingerprintability of extensions especially in the cases where the extension remains otherwise inactive.

•**Injected script tags**. Some extensions add script tags to the page's DOM, which are visible to any other script executing on the page. However, extensions have the ability to add and hide script tags, before any original code is executed on the page, by properly registering content scripts (i.e. scripts running together with webpages with access to a shared DOM).

•**Injected style tags**. Many extensions inject CSS tags into the page's DOM, which are visible to any script on the page. This behavior is unnecessary because there exist special APIs for extensions to inject CSS stylesheets in a page (e.g. `chrome.tabs.insertCSS`). When an extension uses this API, the added styles are not visible via standard CSS-querying APIs, such as, `document.styleSheets`. Using these methods, the injection does not modify the DOM tree.

•**Attributes for body/head/html tags**. Bloat also occurs when an extension sets attributes to parent DOM nodes, such as the `body`, `head` or `html` tag. We discovered many cases of custom attributes with words "installed" or "injected", which do nothing except disclose the extension's presence. Even if these attributes are used in CSS selectors, given that they target well-defined tags, they are unnecessary and can be removed from the corresponding CSS rules.

```
// Unnecessary and invisible placeholders
<div class="plugin-body" style="display:none;">
</div>
<div id="addManualImageDiv" style="display: none;">
</div>
<div id="mystickies"></div>
// Unnecessary top-level attributes
<html style="visibility: visible;">
<body screen_capture_injected ="true">
<body style="">
```

Figure 1: Examples of bloat-related DOM modifications of browser extensions.

Similarly to empty placeholders, attributes with empty values are signs of bloat in the extension's logic. The bottom half of Figure 1 illustrates several real examples of such behavior.

•**Window messages**. The browser provides a programmatic way for different JavaScript execution contexts to communicate via messages. Extensions can send a message to the visited page using `window.postMessage`, but that message can be received by any script running on the page that has registered the appropriate callback function. Since these messages reveal the presence of the extension to the visited page, we consider the presence of such messages on an empty webpage as extension bloat.

2.2 Quantification of extension bloat

To measure the fingerprintability due to browser-extension bloat, we collected 58,034 extensions from the Chrome Store in October 2017. These include 25,779 extensions with permissions to modify web pages on any URL. We evaluated each extension according to the necessary condition of our bloat definition. Namely, we retrieved all the fingerprintable DOM modifications that an extension introduces on any domain and filtered out ubiquitous side effects which were part of an extension's desirable functionality.

In order to automate the testing of extensions, we modified and used the source code of XHound [23]. Given the more narrow definition of the bloat-related DOM modifications (compared to *all* modifications that XHound can uncover), we simplified XHound by taking each extension to a custom domain only once, simulating the visiting of an arbitrary website. Moreover, we served an empty HTML page to extensions in order to eliminate the possibility of any specific content triggers, and we disabled the usage of the XHound's OnTheFlyDOM library as we are not interested in DOM modifications in response to specific content or user actions. As such, we collected only those fingerprintable side-effects that occur on an arbitrary webpage without any triggering content and without user actions. To further minimize the probability of false positives, we filtered out any cases of discovered "bloat" which do not conform to our definition, therefore calculating a lower bound of bloat-related fingerprintability.

Overall, we discovered 3,320 extensions, or 5.7% of all the extensions in Chrome's webstore, which have at least one fingerprintable side effect because of bloat. Out of those, 2,189 inject unnecessary nodes into the DOM (e.g., div, span, script, style), and 1,526 set unnecessary attributes to the body, head or HTML nodes. Moreover, 65% out of 3,320 extensions had *all* their changes categorized as bloat. In other words, these 2,145 extensions should have been completely invisible on our test pages. Figure 2 shows that both popular and less popular extensions are affected by this type of

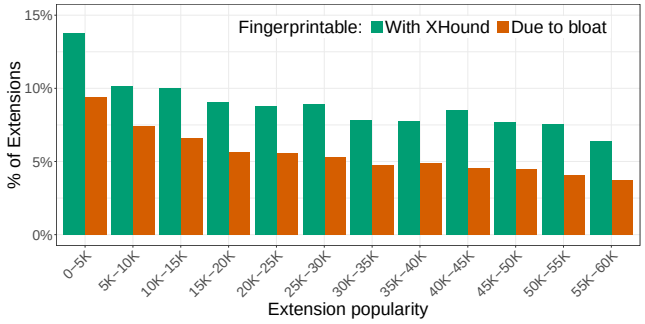

Figure 2: Percentage of fingerprintable extensions, and extensions with fingerprintable bloat, shown by extension popularity. High-ranked extensions tend to be more fingerprintable, and bloat-related side effects contribute to a stable large fraction of fingerprints.

bloat with more than 9% of the top 5K extensions containing bloat, and approximately 4% of the less popular extensions.

Apart from bloat-related DOM modifications, we discovered that 213 extensions reveal their presence by posting disclosing messages to web pages upon installation. We were surprised to find such a large number of extensions since Chrome APIs provide the ability to set up communication channels with only selected web pages with the `externally_connectable` permission. Moreover, we discovered 637 extensions listening to messages from any web page. This means that a tracker can potentially craft and send expected messages and wait for visible side effects from extensions in response. For example, as shown on Figure 3, Crypto-Plugin, a banking extension with 304,730 active users, responds to a specially-crafted message if there is an input element with id `CryptoPluginInstance` and a value containing a JSON-formatted command. Note that this behavior of taking arbitrary content and treating it as command can, in addition to make extensions unnecessarily identifiable, lead to severe security issues.

2.3 Impact on extension fingerprintability

In this section, we compare our results to the unmodified XHound prototype, which deploys additional techniques for triggering the functionality of browser extensions and can thus discover more fingerprintable side-effects. We find that a large fraction of fingerprintable extensions are actually fingerprintable due to bloat and could thus become invisible to trackers, if their bloat-related DOM modifications are removed.

Overall, using XHound, we could identify 5,323 fingerprintable extensions. Given our earlier finding of 3,320 extensions introducing bloat-related fingerprintable side effects, approximately 62% of all the fingerprintable extensions have at least one bloat-related side effect, which adds to their fingerprint. Moreover, even when using XHound's on-the-fly triggering content, 1,073 extensions remain with bloat only, suggesting that this bloat can be safely removed to make extensions invisible from prying webpages.

The bloat-related side effects of 3,320 Chrome extensions resulted in 2,032 extensions sharing their bloat-related fingerprint with no other extension (i.e. their bloat-related DOM modifications are unique to them and could be used to uniquely identify them). In terms of anonymity set sizes: 61.2% had totally distinct fingerprints, 8.8% shared their fingerprint with up to 10 other extensions, 7.2%

```
window.addEventListener("message", function(event) {
  if (event.data.sender == "crypto.plugin.native") {
    console.log(JSON.stringify(event.data));
  }
}, false);

var dummy_input = document.createElement("input");
dummy_input.setAttribute("value", "{}");
dummy_input.id = "CryptoPluginInstance";
document.body.append(dummy_input);

window.postMessage({sender: "crypto.plugin.js"}, "*")

// CONSOLE OUTPUT:
{"answer":{"errorCode":1,"errorText":"Wrong function"},
"type":"error", "sender":"crypto.plugin.native"}
```

Figure 3: Detecting the Crypto-Plugin extension by sending an appropriately-crafted message, simulating a triggering DOM content, and listening to the response.

shared with up to 100 extensions, while 22.8% shared their fingerprint with more than 100 extensions because of similar bloating changes related to jQuery (we discuss this in the next section). As a result, unnecessary bloat-related fingerprintability is, in and of itself, a clearly dangerous extension-fingerprinting vector.

2.4 Origin of bloat in browser extensions

According to our manual analysis of extension bloat (Section 2.1), poorly-designed internal logic is the most common reason for extension bloat. First, we discovered 1,070 extensions with the design flaw of adding placeholders before filling them with useful content. Second, we witnessed several cases of implementation bugs when a DOM modification appears because of poor coding practices. For example, the code in Figure 4 adds an empty class attribute to the body node even if `eod_disabled` was not set initially.

Next to extensions having their custom logic leading to unique cases of bloat, we discovered several libraries that perform unnecessary modifications in the DOM. For example, the jQuery library, when included as a content script, injects an empty style to the body of web pages (discovered on more than 980 extensions). Moreover, particular versions of jQuery have other hardcoded side effects like an injected div tag with id `pg_hgfkj4kj32mda` (version 1.7.2), or the added style `zoom: 1;` to the body of web pages (version 1.9.0). We also discovered the FancyBox JavaScript library on 65 extensions, which adds custom styles to web pages. Additionally, the CrossriderAPI, a cloud-based extension development tool, adds its own attribute to a page's body, namely `crossrider_data_store_temp`, on at least 45 extensions. In addition to the aforementioned cases of bloat, we discovered other cases of the same bloat modifications manifesting across tens of extensions but we could not attribute them to a particular library. These types of modifications could be due to reused code across related extensions or extensions developed by the same developers.

In summary, we discovered that a large number of browser extensions introduce fingerprintable modifications that could have been avoided and which are not necessary for the proper operation of the extension. On the one hand, this is a positive development since this finding suggests that the more careful design and implementation of browser extensions would allow users to be less fingerprintable, without the need to avoid using these extensions. On the other hand, solely relying on extension developers to voluntarily make their

```
document.documentElement.className =
  document.documentElement.className.replace(
    /eod_disabled/g, "")
```

Figure 4: An example of implementation bug in browser extension, which results in unnecessarily fingerprintable DOM modification

extensions more robust against browser fingerprinting, is likely not going to lead to drastic improvements with regard to user privacy. As such, we argue that there is a need to protect users, even when extension authors are not committed to change the logic of their extensions. We discuss such a solution in Section 3.

By comparing the set of fingerprintable/undetectable extensions throughout the years, we discovered that most extensions retain their original status. That is, if the first version of an analyzed extension was fingerprintable, the extension remains fingerprintable through subsequent updates. Overall, only 258 extensions from the initial 1,000 fingerprintable set were not fingerprintable in their first version. Similarly, most extensions that were originally undetectable, remain undetectable. This finding suggests that the fingerprintable bloat of extensions remains stable over time and thus requires an explicit attempt to remove it (i.e. will not disappear in future versions of extensions).

In the following, we present a few case studies of extensions which migrated from undetectable to fingerprintable and vice-versa. One example of an extension which became fingerprintable is Extended JS Console. It was undetectable for two years until the developer used the Crossrider framework to make development across different browsers easier. This framework injects a `crossrider_data_store_temp` attribute to the list of body tags that makes the framework easily detectable. Another example is the Dayboard extension which, after 18 undetectable versions, started to inject a `db-visibility-check` empty div on all visited page. Fortunately, there are also 72 extensions which were once fingerprintable but later became undetectable, e.g., the Web PKI, which, since February 2018, has stopped injecting empty div placeholders.

3 COUNTERMEASURES

Since extension authors cannot be trusted to appropriately limit the access of their own extensions (either due to the inability to specify a finite whitelist that works for all users, a lack of motive, or outright maliciousness), the limiting of those extensions must happen at the client-side.

This means that, in addition to extension authors specifying on which websites a given extension should run, there should be a second layer of access-control which empowers users to limit the access of extensions according to their needs. For example, users may want to disable ad-blockers on specific websites which they want to support with ad-revenue, even if the ad-blockers do not support per-website whitelisting. Similarly, users may want to disable all extensions on their banking website, to limit their exposure, in case of accidental information leakage [22] or malicious data exfiltration [7, 9, 15].

Instead of relying on users to whitelist/blacklist each of their installed extensions on every domain that they visit, we argue that we can build higher-level access-control primitives which can be intuitively utilized by users with limited technical expertise. Specifically, we propose primitives that take the form of the following access-control modes:

```
{   "ExtensionSettings":{
      #UBlock Origin
      "cjpalhdlnbpafiamejdnhcphjbkeiagm":{
         "runtime_blocked_hosts":[
            "*://*.bankofamerica.com"
         ]
      },
      #Dark Mode
      "dmghijelimhndkbmpgbldicpogfkceaj":{
         "runtime_blocked_hosts":[
            "*://*.google.com",
            "*://*.bankofamerica.com"
         ]
}}}
```

Figure 5: Example of an ExtensionSettings policy for two separate extensions.

•**Blocking extensions by default**. The goal of this mode is to stop "drive-by fingerprinting", i.e., the complete fingerprinting of a user's extensions just because a user visited an untrusted website.

•**Automatic disabling of extensions on sensitive domains**. We argue that there is often no need for extensions to be able to run in any and all websites that the users visit. As such, given a list of sensitive URLs (such as banking websites, social networks, dating websites, and healthcare-related websites), the browser can automatically disable all extensions on these domains and allow users to re-enable only the ones that they absolutely need.

•**Content-based automated blocking**. Next to maintaining lists of sensitive websites, we argue that a user's exposure can be further reduced if a browser disables extensions when a webpage allows users to input content. For example, disabling extensions on login pages could help reduce the likelihood of an extension exfiltrating user credentials.

Given the HTTPS issues of client-side proxies and the lack of APIs that a browser extension can use to block other browser extensions, a client-side access control mechanism for browser extensions *has* to include some level of browser modifications. By inspecting the source code of Chromium (our browser of choice given the availability of its source code and its market share) and discussing with developers of Chromium, we discovered the presence of an *Enterprise* mechanism built in the browser. Enterprise mechanisms are typically utilized in corporate environments where system administrators wish to limit the features that are available to individual deployments of powerful software, such as, a web browser. Even though these features are completely invisible to regular users, we discovered that the Enterprise features of Chromium were introduced in 2012 and are still part of the code base of the regular browser that users download.

Chromium's enterprise mechanisms are driven by an enterprise policy written in a platform-dependent format (e.g. JSON on Linux) which allows administrators to manage the browser's configuration and behavior. These policies cover a wide range of browser settings, from power management and proxy configurations, to printers, startup pages, and the default search engine. For our purposes, the `ExtensionSetting` policy allows administrators to disable specific extensions, and stop users from installing any extension that requests specific permissions [2]. The following two directives that control which websites can be accessed by extensions, are of particular interest for our client-side access control mechanism:

•`runtime_blocked_hosts`: List of strings representing hosts whose webpages the extension will be blocked from modifying.

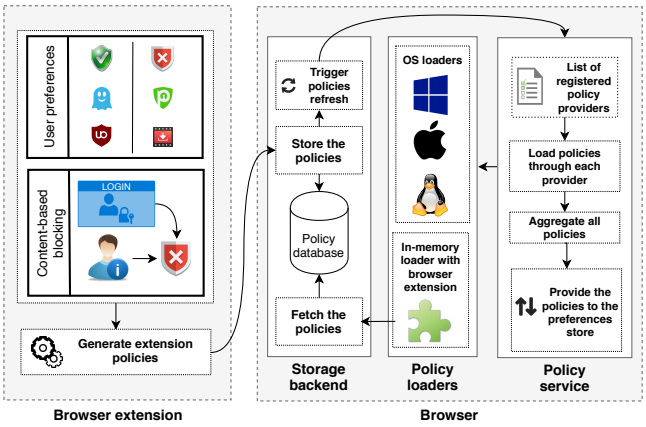

Figure 6: Implementation overview of the client-side access control mechanism for browser extensions.

•`runtime_allowed_hosts`: List of strings representing hosts that an extension can interact with, regardless of whether they are listed in `runtime_blocked_hosts`.

Figure 5 shows an example of an *ExtensionSettings* policy. The first ID refers to the ad blocker "uBlock Origin" and the second to the "Dark Mode" extension (an extension that changes the background color of a website and which we will use in a later demonstration). This policy instructs the browser to block both extensions from accessing any webpages from Bank of America but only the ad blocker is allowed to operate on `google.com`.

The files containing the Enterprise policies, are expected to be present in predefined filesystem paths, according to the operating system of the user. If such a file is discovered during the startup of the browser, it is loaded and enforced by Chromium.

By retrofitting the already available enterprise mechanism and exposing it to regular users through a dedicated UI that converts user choices to JSON policies, we can enable client-side access control of extensions with minimal changes to the browser's code. In addition to modifying as little as possible of the browser's existing code (therefore increasing our chances of adoption), we also rely on a proven mechanism that provides comprehensive extension blocking in a way that is not possible by combining blocking directives available to extension authors (such as the `exclude_matches` directive which allows extension authors to opt-out of running on specific domains and URLs).

The source code of our modified Chromium browser along with a fully compiled version can be found on GitHub: `https://github.com/plaperdr/extension-access-control`.

3.1 Interfacing with the user

Since Enterprise policies are fully controlled by JSON-formatted files available in obscure file paths, we cannot reasonably expect users to directly interact with them. To bridge the Enterprise policy mechanism with the interfaces that users are already familiar with, we developed a browser extension that creates policies based on user choices. Next to allowing users to individually disable extensions on websites through our extension's UI, we provide support for the following three blocking strategies, which map to the access-control primitives described in Section 3:

•**Flexible mode (default mode)**: All extensions are allowed on all webpages and the user can disable (or re-enable) at any time an extension on a specific website. Note that this mode does not interfere with the selective exclusion that extension authors may request using the `exclude_matches` directive in their manifest files. As such, extensions that were previously active only on selected URLs (e.g. a video-downloading extension being active only on `youtube.com`) remain active only on the same URLs.

•**Sensitive mode**: This mode is similar to the *Flexible mode* but comes preloaded with a list of sensitive websites where all extensions are automatically blocked. For our proof-of-concept implementation, this list consists of 50 popular banking websites. This mode also supports content-based blocking where extensions can be disabled if our mechanism detects the presence of sensitive content on a website that is not part of our predefined lists. Specifically, our current implementation blocks extensions whenever a password input field is detected in a webpage's HTML code.

•**Strict mode**: In this mode, all extensions are by default blocked on all websites. Users can selectively enable extensions wherever they require them.

Given a desired policy of blocking and enabling browser extensions on specific websites, this policy needs to be communicated to the enterprise policy-parsing mechanisms of the browser. Chromium has a relatively complex policy-loading system that spans dozens of files and thousands of lines of code. Figure 6 provides an overview of our modifications. First, our user-facing extension transforms all user choices and all rules derived from content-based blocking into an *ExtensionSettings* policy readable by the browser, indicating for each installed extension which hosts are blocked and which are allowed. This policy is then stored in a database that is available to our extension.

As mentioned earlier, the Enterprise mechanism expects policies to be available on different filesystem paths depending on the user's operating system. Specifically, Chrome's policy loader uses GPO (Group Policy Object) on Windows, XML files on MacOS, and JSON files on Linux. All of these locations are encoded as different "providers" in one combined mechanism called a *Policy Service*. For our implementation, we modified the Policy Service and added a new policy provider which pulls new policies directly from the database of our user-facing extension. Whenever a policy changes (e.g. because the user whitelisted/blacklisted an extension, or switched modes), our service triggers a "refresh" which causes Chromium's Policy Service to pull a new copy of the policy. The policies from all policy providers are then combined into a unique policy and transmitted to Chromium's *Preferences Store* that will parse it and apply it.

3.2 Evaluation

Performance Overhead. To test the overhead of our client-side access control mechanism, we visited the Alexa Top 50 websites with the original (i.e. unmodified) version of the Google Chromium browser (v.71) and our modified copy of the same version of the same browser. For each visited page, we recorded the time it takes for the browser to completely render every element. Specifically, we collected the timestamps corresponding to the following three events from the *window.performance.timing* object:

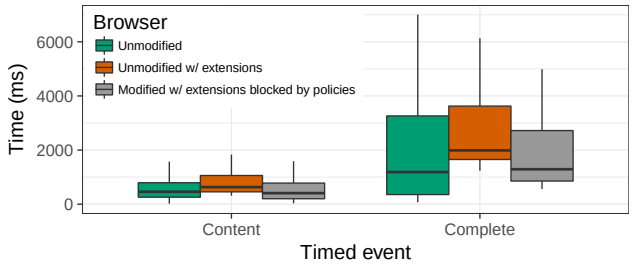

Figure 7: Performance measurements of unmodified Chromium version 71 with and without installed extensions, against our modified Chromium browser.

• the *DOMLoading* event is fired when the parser starts parsing the received webpage.

• the *DOMContentLoaded* event marks the point when both the DOM is ready and there are no stylesheets that are blocking JavaScript execution.This means that the browser can start combining the DOM and CSSOM into a render tree.

• the *DOMComplete* event is fired when the page and all of its subresources are ready.

We tested three different configurations: unmodified Chromium with no running extensions, unmodified Chromium with extensions, and our custom Chromium in "Strict" mode with extensions (i.e. all extensions are present but blocked by our custom policy). For our tests, we utilized five popular extensions: AdBlock [1], Google Keep [3], Grammarly [4], Honey [5], and LastPass [6]. The number of extensions was chosen to match the average number of extensions that regular users install, according to Starov et al. [23].

We repeated the measurements ten times and averaged the results, which are shown in Figure 7. The "Content" category corresponds to the time between the *DOMLoading* event and the *DOMContentLoaded* one. The "Complete" one is the time between *DOMLoading* and *DOMComplete*.

For both categories, we observe that the timings of our modified browser with all extensions disabled by the applied "Strict" policy closely match the timings of the unmodified browser without any extensions installed. Given that our mechanism makes use of existing Chromium code which we retrofit to give users the ability to make client-side access-control decisions, the obtained performance results make intuitive sense. Overall, our findings demonstrate that client-side access control of extensions not only will not slow down a user's browser but will, in fact, increase its performance by disabling extensions when they are not necessary.

Blocking Coverage. To test whether the extensions disabled through our access-control mechanism are completely disabled, we performed the following experiment. We selected 1,000 extensions which were introducing bloat-related fingerprintable DOM side effects and installed them, one by one, on our modified Chromium browser. We then used that browser together with the XHound tool [23] and attempted to extract fingerprints from each extension.

For the 985 extensions that we could install in our modified browser (15 out of the 1K sample would not properly execute on Chromium version 71), XHound was unable to extract fingerprintable DOM modifications, when these extensions were blocked through the "Strict" policy of our client-side access control mechanism. Given that XHound searches for any and all DOM modifications that can be attributed to an extension, this gives us confidence that our modified browser successfully disabled all extensions.

4 RELATED WORK

In the last few years, the privacy implications of browsers extensions have received significant attention [9, 14, 18, 19, 21–23, 25]. To the best of our knowledge, this is the first paper that i) quantifies to what extent the fingerprintable DOM modifications performed by extensions are necessary, ii) points out the resulting issue of *bloat* in the context of browser extensions, and iii) proposes a client-side access control mechanism for reducing the footprint of extensions in a user's browser which can counter many of the attacks described in the aforementioned papers. In concurrent work, Sjösten et al. described the privacy issues of unique extension identifiers and also proposed a client-side access control system for extensions [19]. While the two systems are conceptually similar, our work utilizes existing code within the Chromium browser thereby avoiding the disabling of security checks of browser extensions and the need for extension rewriting.

On October 1, 2018 and in parallel with our work, Google announced that future versions of their Chrome browser will allow users to selectively enable and disable extensions on different websites [24]. We find this a welcoming development and a confirmation of the need of client-side access control for browser extensions. The current instantiation of that mechanism allows users to whitelist extensions on a site-by-site basis, or only enable them when the user interacts with them through the browser's UI. We argue that our proposed policies strike a better balance between security and usability (e.g. extensions enabled everywhere *except* on sensitive sites) and we therefore hope that Google, as well as other browser vendors, will follow our proposed design.

5 CONCLUSION

In this paper, we investigated the fingerprintability of browser extensions due to bloat, i.e., the unnecessary side-effects caused by faulty application logic that reveal an extension's presence without providing any useful functionality. Bloated extensions represent a risk to online privacy as they facilitate the fingerprinting of installed extensions. We analyzed 58,034 extensions from the Google Chrome store and found that 5.7% of them contained fingerprintable bloat. For 61% of these extensions, their bloat was unique which can be abused for their direct identification. Finally, we presented the design and implementation of a client-side access control mechanism for Google Chromium which enables users to control the reach of extensions, either on a one-by-one basis, or via access modes (e.g. automatically blocking all extensions on sensitive websites). Overall, our paper highlights the problems associated with bloat for both users and developers. We hope that this work will motivate the more careful implementation of extensions by their developers and the adoption of client-side access control by all modern browsers.

Acknowledgements: This work was supported by the Office of Naval Research (ONR) under grant N00014-17-1-2541 and by the National Science Foundation (NSF) under grants CNS-1813974, CMMI-1842020, CNS-1617593, CNS-1703375, and CNS-1527086.

REFERENCES

[1] 2018. AdBlock - Chrome Web Store. https://chrome.google.com/webstore/detail/adblock/gighmmpiobklfepjocnamgkkbiglidom.

[2] 2018. Extension Settings Full Description | The Chromium Projects. https://www.chromium.org/administrators/policy-list-3/extension-settings-full.

[3] 2018. Google Keep Chrome Extension - Chrome Web Store. https://chrome.google.com/webstore/detail/google-keep-chrome-extens/lpcaedmchfhocbbapmcbpinfpgnhiddi.

[4] 2018. Grammarly for Chrome - Chrome Web Store. https://chrome.google.com/webstore/detail/grammarly-for-chrome/kbfnbcaeplbcioakkpcpgfkobkghlhen.

[5] 2018. Honey - Chrome Web Store. https://chrome.google.com/webstore/detail/honey/bmnlcjabgnpnenekpadlanbbkooimhnj.

[6] 2018. LastPass: Free Password Manager - Chrome Web Store. https://chrome.google.com/webstore/detail/lastpass-free-password-ma/hdokiejnpimakedhajhdlcegeplioahd.

[7] Lawrence Abrams. 2018. MEGA Chrome Extension Hacked To Steal Login Credentials and CryptoCurrency. https://www.bleepingcomputer.com/news/security/mega-chrome-extension-hacked-to-steal-login-credentials-and-cryptocurrency/.

[8] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. 2014. The Web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*.

[9] Quan Chen and Alexandros Kapravelos. 2018. Mystique: Uncovering Information Leakage from Browser Extensions. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[10] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. 2018. The Web's Sixth Sense: A Study of Scripts Accessing Smartphone Sensors. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[11] Peter Eckersley. 2010. How Unique Is Your Browser?. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*.

[12] David Fifield and Serge Egelman. 2015. Fingerprinting web users through font metrics. In *Financial Cryptography and Data Security*. Springer, 107–124.

[13] 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 World Wide Web Conference (WWW)*.

[14] Gabor Gyorgy Gulyas, Doliere Francis Some, Nataliia Bielova, and Claude Castelluccia. 2018. To Extend or Not to Extend: On the Uniqueness of Browser Extensions and Web Logins. In *Proceedings of the 2018 Workshop on Privacy in the Electronic Society (WPES'18)*.

[15] Alexandros Kapravelos, Chris Grier, Neha Chachra, Chris Kruegel, Giovanni Vigna, and Vern Paxson. 2014. Hulk: Eliciting Malicious Behavior in Browser Extensions. In *Proceedings of USENIX Security Symposium*.

[16] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. 2016. Beauty and the Beast: Diverting modern web browsers to build unique browser fingerprints. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[17] 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 *Proceedings of the IEEE Symposium on Security and Privacy*.

[18] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. 2017. Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies. In *Proceedings of USENIX Security Symposium*.

[19] Alexander Sjösten, Steven Van Acker, Pablo Picazo-Sanchez, and Andrei Sabelfeld. 2019. LATEX GLOVES: Protecting Browser Extensions from Probing and Revelation Attacks. In *Network and Distributed System Security Symposium (NDSS)*.

[20] Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld. 2017. Discovering Browser Extensions via Web Accessible Resources. In *Proceedings of the ACM on Conference on Data and Application Security and Privacy (CODASPY)*.

[21] Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld. 2017. Discovering browser extensions via web accessible resources. In *Proceedings of the ACM on Conference on Data and Application Security and Privacy (CODASPY)*.

[22] Oleksii Starov and Nick Nikiforakis. 2017. Extended Tracking Powers: Measuring the Privacy Diffusion Enabled by Browser Extensions. In *Proceedings of the International Conference on World Wide Web (WWW)*.

[23] Oleksii Starov and Nick Nikiforakis. 2017. XHOUND: Quantifying the Fingerprintability of Browser Extensions. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[24] James Wagner. 2018. Trustworthy Chrome Extensions, by Default. https://security.googleblog.com/2018/10/trustworthy-chrome-extensions-by-default.html.

[25] Michael Weissbacher, Enrico Mariconti, Guillermo Suarez-Tangil, Gianluca Stringhini, William Robertson, and Engin Kirda. 2017. Ex-Ray: Detection of History-Leaking Browser Extensions. In *Proceedings of the ACM Annual Computer Security Applications Conference (ACSAC)*.