Fingerprinting in Style: Detecting Browser Extensions via Injected Style Sheets

Pierre Laperdrix
Univ. Lille, CNRS, Inria

Oleksii Starov
Palo Alto Networks

Quan Chen
North Carolina State University

Alexandros Kapravelos
North Carolina State University

Nick Nikiforakis
Stony Brook University

Abstract

Browser extensions enhance the web experience and have seen great adoption from users in the past decade. At the same time, past research has shown that online trackers can use various techniques to infer the presence of installed extensions and abuse them to track users as well as uncover sensitive information about them.

In this work we present a novel extension-fingerprinting vector showing how style modifications from browser extensions can be abused to identify installed extensions. We propose a pipeline that analyzes extensions both statically and dynamically and pinpoints their injected style sheets. Based on these, we craft a set of triggers that uniquely identify browser extensions from the context of the visited page. We analyzed 116K extensions from Chrome's Web Store and report that 6,645 of them inject style sheets on any website that users visit. Our pipeline has created triggers that uniquely identify 4,446 of these extensions, 1,074 (24%) of which could not be fingerprinted with previous techniques. Given the power of this new extension-fingerprinting vector, we propose specific countermeasures against style fingerprinting that have minimal impact on the overall user experience.

1 Introduction

In the last decade, researchers have revealed that a user's online activity is invisibly tracked by a multitude of third parties. These third parties record the websites that users visit in an effort to better understand them (i.e. their socioeconomic characteristics and preferences), most commonly for the purpose of better ad targetting. This type of tracking happens through two broad sets of tracking techniques: *stateful tracking* and *stateless tracking*.

Stateful tracking makes use of browser cookies and other stateful identifiers that enable trackers to recognize returning users and expand their browsing profiles with newly visited websites [41]. Because of the limitations of stateful tracking (such as the existence of options to block third-party cookies and a browser's private mode) stateless tracking techniques arose that enable third parties to track users across sessions, without relying on previously set cookies or other stateful identifiers. These stateless techniques essentially "fingerprint" a user's browsing environment (such as the exact version of their browser, the resolution of their screen, and the way with which their graphics card renders complex 3D images) and associate browsing sessions with this fingerprint [15, 19, 30, 33, 37]. As long as a user's fingerprint remains relatively stable over time, this approach subsumes the need for cookies and works equally well both in and out of a browser's private mode.

The most recent addition to the arsenal of browser fingerprinting is the fingerprinting of browser extensions, such as, ad-blockers, video downloaders, productivity tools, and password managers. Prior work has shown that browser extensions can be fingerprinted by the resources they make available to websites [22, 24, 44], the way they modify a page's DOM [29,45,47], and the messages they send between origins with *postMessage* [29, 45]. Unlike traditional fingerprinting which could only be abused in the sense of offering bits of entropy for differentiating users from each other, the ability to detect browser extensions can also be abused to infer sensitive information about users. This is because users *choose* to install specific browser extensions and these choices can betray sensitive information about them. Recent work by Karami et al. [29] showed that browser extensions can reveal, among others, a user's age, religion, political affiliation, and ethnicity.

In this paper, we present a new method of fingerprinting browser extensions which, to the best of our knowledge, has never been presented before. Our fingerprinting method arises from the observation that, like regular web pages, browser extensions rely on Cascading Style Sheets (CSS) for the styling of their user interfaces (UIs). These UIs include not only the user-facing UIs that are invisible to pages (such as the UIs shown to users who click on an extension's icon), but also the ones that extensions inject in the pages where they are active (e.g. a new download menu under each YouTube video). This observation coupled with the ability of modern browsers to

check the styling of individual DOM elements, allow web pages to create "tripwire" DOM elements that have the same IDs and class names as the ones that an extension injects and styles. A webpage can therefore present thousands of invisible elements to a visiting user's browser and detect the ones whose styles are different than the default ones. In this way, a web page can detect the presence of specific extensions, without the need of any user interactions.

To quantify the vulnerability of browser extensions to this new attack, we design an analysis pipeline that detects both statically and dynamically whether an extension injects CSS rules into public webpages, extracts correspondent CSS selectors and builds a set of triggers that can be used for fingerprinting (e.g., DOM elements or hierarchies with particular class names and IDs), tests those triggers dynamically for actual style or dimension changes and whether those changes are stable from visit to visit, and finally, evaluates the uniqueness of the obtained fingerprints. By analyzing more than 116K extensions from the Chrome extension store, which include at least 6,645 extensions that add styles on any URL, we could fingerprint 4,446 extensions, which can be uniquely identified by any web page. Among them, 1,074 extensions could not be fingerprinted with existing methods. Finally, given the severity of the attack, we present a new countermeasure that hides styles from extension origins through a self-contained web component called Shadow DOM. When the browser checks for the style of an element, its call is rerouted to a mirrored DOM that is free of all extension styles, deceiving any fingerprinting attempts.

2 Background

This section provides the necessary background on browser extensions, focusing on how the CSS rules injected by extensions can be used to fingerprint them. We briefly discuss known privacy risks from browser extensions and also provide the necessary details of the `getComputedStyle` API, which enables the fingerprinting techniques we present in this work.

2.1 Browser extensions and Style Sheets

Figure 1 shows the high-level architecture of modern browser extensions. A browser extension is essentially a set of JavaScript, HTML, and CSS files that implements the functionalities of the extension, packaged into a single zip archive together with a mandatory manifest file describing the extension. Apart from providing metadata about the extension, such as, an extension's name and version number, the manifest plays a crucial role in that it allows the extension authors to specify background scripts that listen for specific page events, content scripts that are injected and executed in the page context, and CSS rules to be applied on the matching page elements. Altogether, background/content scripts and CSS rules allow extensions to achieve their essential functionality.

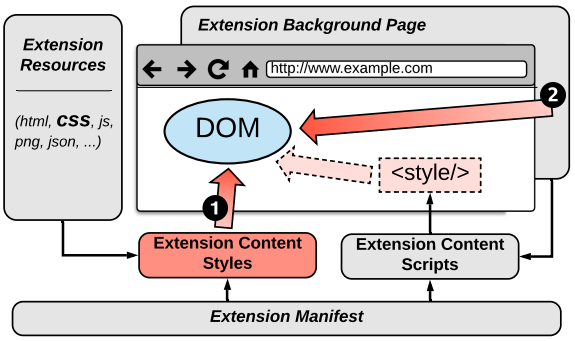

Figure 1: *Different ways that browser extensions use to inject styles.*

Content scripts and CSS rules can be injected either declaratively via the manifest using match patterns [11] (in which case they are injected automatically by the browser into pages with the matching URLs), or they can be injected programmatically at runtime. In the case of CSS rules, programmatic injection is done via the extension API `chrome.tabs.insertCSS`, which is only available to JavaScript code running in the extension context (and not the normal webpage JavaScript). Additionally, since the content scripts (regardless of whether injected declaratively or programmatically) run in the same context as the page they are injected in, they can also modify the style sheets of the page. Figure 1 shows the means in which extensions can affect the CSS rules of a page.

In current browser implementations, the effects of CSS rules injected by extensions are visible to *all* JavaScript code running on the affected page, regardless of their origin, and regardless of the fact that such injected style sheets are hidden from the `document.styleSheets` API. This presents a channel where information about the installed extensions can be leaked. For example, a malicious script can deliberately inject an element that matches the CSS rules injected by extensions, and then use the `getComputedStyle` API (discussed in Section 2.3) to read back the CSS properties after all CSS rules are applied by the browser. Given a database of which extensions style which elements and in what way, a script can create thousands of "tripwire" elements, check which elements' CSS properties are modified, and deduce the presence of specific installed extensions. This information leak forms the basis of our work.

2.2 Risks of using and detecting browser extensions

Browser extensions are known to expose their users to increased privacy risks, either in an *active* or in a *passive* way. Previous work (e.g., [18, 21, 31, 46, 50]) has shown that extensions can actively endanger user privacy by abusing their access to privileged APIs and exfiltrate sensitive user information over the network. Orthogonally to active abuse, fingerprinting installed extensions can reveal private and per-

sonal information about the user. As some extensions offer very specific functionality, their presence can reveal the user's age, interests, ethnicity, political affiliation or religion, which could then be abused to build a profile and serve targeted ads [29]. Moreover, having an exact list of installed extensions in the browser introduce additional entropy for fingerprinting a user's browsing environment. Previous works demonstrated that browser extensions can be fingerprinted via, for example, their Web Accessible Resources (WARs) [22, 24, 44], or the changes they introduce in the DOM [45, 47]. Section 7 provides a detailed description of previous extension-fingerprinting techniques.

2.3 The `getComputedStyle` API

The techniques we present in this paper primarily rely on the DOM API `window.getComputedStyle`, which takes a DOM element (e.g., a `div` element) and returns the resolved CSS properties of that element, after all active style sheets are applied [13]. The return value also takes into account element-specific properties (e.g., inline `style` attributes) along with the current JavaScript modifications. The Internet Explorer browser implements a proprietary version of this API, albeit as an element property `currentStyle` (accessed as `Element.currentStyle` on the target DOM element) [10]. Since this API returns the computed (i.e., actual showing) CSS properties, such as width/height and background color of an element, it provides web developers with an accurate view of the rendered UI elements [40].

In addition to static styling, with the CSS3 specification, all major browsers now support creating *transitions* and *animations* of HTML elements using CSS. Transitions specify that a CSS property change should be done gradually over a period of time, while animations are used to animate other CSS properties (e.g., color, width/height) by specifying key frames. The `getComputedStyle` API also plays an important role here by allowing developers fine grained control over the animation, or otherwise to trigger the starting or ending of a transition [9].

2.4 Known Risks of `getComputedStyle`

It is well-known in the web security and privacy community that a malicious website could deduce the user's browsing history by using a technique called link color differentiation [16, 26]. A malicious website could inject a list of hyperlinks of interest as DOM objects, and use the `getComputedStyle` API on each injected hyperlink and check their color: a previously visited link will have a different color than the non-visited ones. In response to this type of information leakage, major browsers modified the implementation of `getComputedStyle` so that it always reports the unvisited color for hyperlinks.

```
<div class="drwebThreatLink">(trigger)</div>
```

(a) *Example of a CSS trigger*

(b) *No extension* ***(c)*** *With extension*

Figure 2: *Appearance of the HTML trigger (a) when the Dr.Web Link Checker extension (239K users) is absent (b) and present (c).*

Previous research [25] has also used `getComputedStyle` in attacks aiming to steal confidential information from victim websites by utilizing so-called cross-origin CSS. Due to the permissive nature of CSS, attackers can inject CSS rule fragments into the target webpage that contains confidential information (e.g., by sending CSS rule fragments as email titles so they appear in the victim's inbox page), and then induce the victim to visit a website controlled by the attacker. The attacker website will then import the entire target page as a style sheet, and finally use `getComputedStyle` to retrieve confidential information from the target page.

3 Style-Fingerprinting Example and Threat Models

As we described in Section 2, browser extensions have multiple ways to style elements that they introduce in webpages. Unfortunately, web pages can take advantage of this behavior by presenting trigger elements, i.e., elements with the appropriate IDs and class names which exist for the sole purpose of matching the CSS rules of the present extensions and thereby inheriting the specified styles.

Figure 2 shows a class-based trigger that can be used to detect the presence of an extension called *Dr. Web* in the browser. The visual appearance of the trigger element with class "drwebThreatLink" radically changes when the extension is installed, since it inherits all the CSS properties that are injected by that extension (shown in Listing 1). A webpage can use all of the properties listed in Table 1 to detect style changes in that element, or check for the resulting dimensional changes with the listed methods, and thereby infer the presence of that extension. Note that all of the above happens without the need of user interaction and can therefore fingerprint extensions that inject CSS rules in a webpage but do not change a webpage in any other way. A video demo that demonstrates the power of our proposed technique by fingerprinting 20 extensions without any user interaction is available at this URL: https://vimeo.com/430428308

Listing 1: *Extension-injected CSS rules for the example trigger*

```
r.drwebThreatLink {
    background-repeat: no-repeat;
    width: 86px;
    height: 84px;
    background-position: 0 0;
    background-image: url(data:image/png;base64
    ,...);
}
```

Given that an extension must have the permission to inject CSS rules in a given webpage (we describe the permission system and manifest files in more detail in Section 4) we identify two separate classes of fingerprintable extensions, that match the ones of Starov and Nikiforakis [47]:

- **Fingerprintable on any domain** These extensions are the ones that have permissions to operate on all domains that users visit and thereby potentially inject CSS rules in all of these domains. Typical examples of these extensions would be ad-blockers, password managers, security- and privacy-related extensions, and screenshot extensions. In this case, any website that a user visits has the ability to deploy the appropriate CSS-based triggers and detect the presence of a given extension.
- **Fingerprintable on some domains** Many extensions are tailored to one or more specific domains, typically those of popular services, such as, GMail, Twitter, and YouTube. In this case, these extensions can only be fingerprinted on these domains. Note however that prior research has identified the large footprint of third parties on the popular web [35]. Any JavaScript-capable third party that is present on a domain on which an extension is active, can deploy arbitrary trigger elements and therefore fingerprint these specialized extensions.

4 Data collection and processing

In this section, we detail our initial dataset of browser extensions and how we process them to extract and verify their fingerprints. The presented pipeline is used to build our database of style fingerprints that we analyze in Section 5.

4.1 Initial dataset

For our experiments, we collected 116,485 extensions from the Chrome Store in April 2019, intentionally excluding irrelevant themes and apps. We cover all types of extensions from the most popular ones with millions of users to those with one or no user at all at the time of writing. Each collected extension was submitted to the pipeline detailed below in order to obtain a final "ready-to-use" fingerprinting script, which can be deployed on any domain and URL. This fingerprinting script consists of DOM triggers for particular style changes and logic to determine the cause of each change. In addition, we also collected 501,349 extensions and their versions dating back from as early as 2014 to perform a longitudinal analysis (see Section 5.7 for more details).

Table 1: *Changed visible properties of the example trigger*

window.getComputedStyle	***Position & Dimensions***
background	getBoundingClientRect.bottom
backgroundImage	getBoundingClientRect.height
backgroundPosition	getBoundingClientRect.right
backgroundPositionX	getBoundingClientRect.width
backgroundPositionY	offsetHeight
backgroundRepeat	offsetWidth
blockSize	
height	
inlineSize	
perspectiveOrigin	
transformOrigin	
webkitLogicalHeight	
webkitLogicalWidth	
webkitPerspectiveOrigin	
webkitTransformOrigin	
width	

We gathered these extensions by crawling daily the Chrome Store website with a custom script written in Python that makes HTTP requests using the *requests* library. It stores all metadata and extensions encountered in a MongoDB database. Though the appropriate setting of the HTTP User Agent, the script pretends to be a recent Chrome browser version (updated occasionally over the years) and fetches the information page of all publicly listed extensions available at `https://chrome.google.com/webstore/sitemap`. It then proceeds to download all extensions that have a new version that does not exist in our database. The script is ~100 lines of Python code and executes daily via a cronjob since 2014.

4.2 Processing pipeline

Figure 3 provides an overview of our processing pipeline to generate style fingerprints. At the very end of our pipeline, each remaining trigger links back to a single browser extension from our dataset. It should be noted that this pipeline can be executed as often as necessary to obtain new fingerprinting scripts for updated browser extensions. Our implementation is currently limited to the WebExtension format supported by Chrome, Firefox, Opera, Edge, and Brave. Note, however, that our attack uses standardized JavaScript APIs and can therefore be extended to other extension systems.

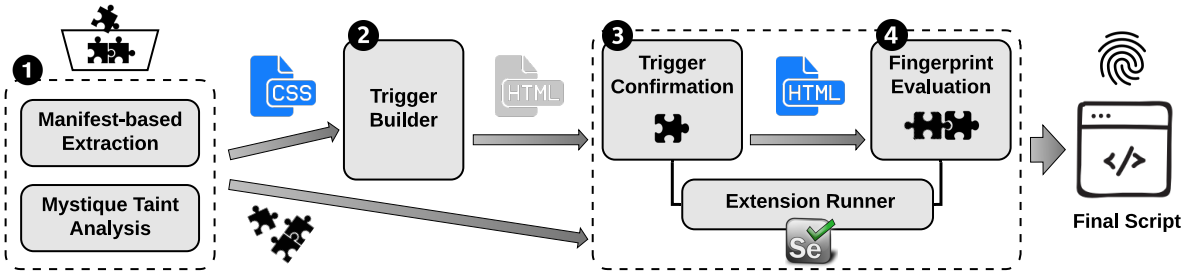

Figure 3: *Extension analysis pipeline for collecting style-fingerprints: ① extract injected CSS; ② generate candidate triggers; and perform dynamic tests for ③ trigger confirmation and ④ final fingerprint evaluation.*

Listing 2: *Extract from the manifest.json file of the Wikiwand: Wikipedia Modernized extension*

```
1  "content_scripts": [
2      {
3        "matches": [
4          "http://*/*",
5          "https://*/*"
6        ],
7        "css": [
8          "css/autowand.css",
9          "css/cards.css"
10       ],
11       "js": [
12         (...)
13       ],
14       "run_at": "document_start"
15     }
16 ]
```

4.2.1 Extracting injected CSS

The first step is to extract styles that can be injected in a web page by an extension.

Detecting declarative injection With the *manifest.json* file, a developer can declare what CSS style sheets should be applied to the DOM. Listing 2 presents a snippet of the manifest from the *Wikiwand: Wikipedia Modernized* extension. Here, through *content scripts*, the extension injects two different CSS files (lines 7 to 9) on all HTTP and HTTPS URLs (lines 4 and 5).

Since all extensions have a manifest file, it is straightforward to automate the detection by iterating through all of them and parsing the *content_scripts* field.

Detecting programmatic injection CSS can also be injected dynamically by calling the appropriate browser APIs. Statically detecting these injections is challenging since the code may be obfuscated and the injected code may be assembled at runtime (e.g. through the concatenation of multiple variables).

Quan et al. developed a tool called Mystique that uses taint analysis to detect leaks of privacy-sensitive information in browser extensions [18]. Mystique builds upon the HoneyPages mechanism by Kapravelos et al. [28] where specific elements are populated in the browser's DOM as extensions are requesting them. For our purposes, this means that we do not need to know beforehand the requirements for a style to be injected as Mystique will resolve the calls to missing elements on the fly. In our experiment, we used Mystique's web interface [34] to monitor calls to the `tabs.insertCSS` API and save the styles injected in the DOM. This approach will capture injections of both raw CSS code as well as paths to CSS files.

4.2.2 Generating style triggers

After identifying what styles are injected by each extension, the second step converts all the collected CSS rules into decoy triggers. The goal is that each trigger will receive the corresponding style changes when the right extension is present. Note that this is not a straightforward engineering task given the wide range of possible CSS selector constructions and complexity of required DOM hierarchies. As such, we devised a pragmatic and effective approach for the translation of CSS rules to triggers, focusing on IDs and class names to recreate the trigger hierarchy. As detailed in Section 5, we did not need to consider additional CSS constructs like pseudo-classes or pseudo-elements when building triggers as extensions were already fingerprintable by only focusing on IDs and class names.

Listings 3 and 4 present an example of a CSS rule that is converted into a decoy trigger. To make the transformation, we divide the selector into its different parts and build the corresponding hierarchy. Here, the first element we generate is a *div* with the *ww_hovercard* ID (if the type of an element is not specified, we used a *div* by default). Then we add another *div* with the *ww_image* class and we finish with an *img* element. When running the test page, the style of the structure we generated will match the rule of the injected CSS and the style will be applied.

Listing 3: *CSS rule from the "Wikiwand: Wikipedia Modernized" (WikiWand) extension*

```
#ww_hovercard .ww_image img {
  display: block;
  float: right;
  max-height: 150px;
  max-width: 180px;
  width: auto;
  height: auto;
  margin: 10px;
  border-radius: 2px;
}
```

Listing 4: *Decoy trigger for the WikiWand extension*

```
<div id="ww_hovercard">
  <div class="ww_image">
    <img trigger="yes"></img>
  </div>
</div>
```

It should be noted that we limited ourselves to 50 triggers per extension as some of them included full libraries with hundreds of rules. Generating triggers for each of them would have been redundant as only a few of them are needed to identify them. At the same time, the fact that there are hundreds of ways that these extensions can be fingerprinted shows the difficulty of defending against this type of fingerprinting.

4.2.3 Confirming trigger fingerprints

The third step consists in verifying that all generated triggers are correct and can be exploited to perform extension fingerprinting. Indeed, even if triggers were built directly from CSS rules, it can be hard to predict the exact runtime behavior of an extension. Other styles could counter its effect and dynamic code could remove an element or change its class on the fly. For these reasons, we need to perform a thorough verification as there is no guarantee that a decoy trigger will be effective in identifying an extension. As part of this verification, we perform the following checks:

- We need to ensure that the observed changes are consistent over multiple runs. We collect style changes from the test page of each extension three times and check that they are identical. This check helps us to discard non-deterministic changes that are the result of unreliable extension behavior.
- We also need to verify that our baseline calculation is effective. In our test pages, we use a baseline element to decide if a style was applied to an element or not. This baseline element is located in a hierarchy that mimics the decoy one, but with one important difference: it does not have any IDs or class names. This way, if we detect differences between the baseline element and the decoy trigger, we can build the extension fingerprint from their differences.

Listing 5: *Decoy trigger with the baseline elements.*

```
<div class="trigger" id="26622">

  <!-- Baseline Elements -->
  <div orig_id="ww_hovercard">
    <div orig_class="ww_image">
      <img trigger="no"></img>
    </div>
  </div>

  <!-- Trigger Elements -->
  <div id="ww_hovercard">
    <div class="ww_image">
      <img trigger="yes"></img>
    </div>
  </div>

</div>
```

Listing 5 shows the final code our system generated for the trigger that we presented in Listing 4. The first structure is the baseline one while the second one is the one where the extension (if present) will apply the corresponding style. The style differences between the two will form the style fingerprint of the extension.

4.2.4 Verifying collisions between extensions

While the analysis of a single extension can obviously reveal injected CSS styles, this is not sufficient to extract and craft unique fingerprints. If a change of style is triggered by an extension, there is no guarantee that no other extension produces the exact same style change. Some extensions could share the same IDs and class names while others could inject very generic rules. To characterize possible collisions, we exposed each extension capable of injecting CSS against the triggers of all extensions and recorded all the style changes.

5 Analysis

This section provides a detailed reporting of how extensions are fingerprintable through the styles they inject. We look at what makes them identifiable and, for the ones that are not identifiable, we explore the reasons why. We focus on studying extensions that inject style rules universally on all web pages (and are therefore fingerprintable on all page). Finally, we also look at older versions of the extensions present in our dataset to understand whether extensions are becoming fingerprintable over time.

5.1 Pipeline statistics

Table 2 reports on the impact of our pipeline on our complete dataset of 116,485 Chrome extensions.

Table 2: *Number of extensions and triggers kept after each step of the pipeline shown in Figure 3 (Ma=Manifest, My=Mystique)*

		Steps			
	Initial dataset	*1*	*2*	*3*	*4*
Extensions	116,485	6,543 (Ma) 137 (My) 6,645 (Combined)	5,885	4,806	4,446
Triggers	-	-	102,997	54,788	40,722

Step 1. After parsing the *manifest.json* file of all extensions, 17,712 extensions (15.2%) inject at least one CSS file through the *Content script* directive and 6,543 of them are doing so on any domain. By using Mystique, we detected 137 extensions that rely on `tabs.insertCSS` to inject styles dynamically into a page. Since 35 them were already injecting styles declaratively, we ended up with **6,645** potential fingerprintable extensions. Note that this number represents the ceiling of our fingerprinting technique. An extension that does not inject CSS rules cannot be fingerprinted through them.

Step 2. To generate the corresponding triggers, we use the rules present in CSS files listed in the manifests and the ones recorded by Mystique. In total, we generated 102,997 decoy triggers distributed across 5,885 test pages, one page for each extension. For the extensions where we could not generate triggers, it was mainly due to the presence of pseudo-classes in the rules. Pseudo-classes are keywords in CSS that reflects the state of an element like `hover`, `focus` or `active` and they require specific user interaction to be activated. Even though we could craft pages for these specific scenarios, our goal is to study style fingerprinting that can happen in the background *without user interaction*, so we discarded them. Other extensions that had empty CSS files or with all rules commented out were also removed at this stage.

Step 3. The goal of this step is to confirm that differences in styles are indeed detectable. We ran all the extensions on their own test pages with Selenium to collect the style fingerprints. For some extensions, we observed no difference between the trigger element and the baseline. This happened when some of the rules were very generic and did not rely on a specific classes or IDs. For other extensions, Selenium crashed or did not return any data. At the end of this step, we had 54,788 confirmed triggers for 4,806 potentially fingerprintable extensions.

Step 4. The final step is to make sure that no two extensions share the exact same style fingerprint. We tested each of the 6,645 extensions on all the triggers from the 4,806 potentially fingerprintable extensions to identify possible collisions between fingerprints. We describe the results of this particular step in more detail in Section 5.4. After verification, we removed 14,066 decoy triggers that produced the exact same change between two or more extensions. 4,446 (3.8%) extensions out of our initial set of 116,485 extensions can be uniquely identified on any webpage because of the styles they inject.

5.2 Evaluating different fingerprinting strategies

An advantage of style fingerprinting compared to more traditional browser fingerprinting, is that the quantity of collected data can be adapted depending on the desired speed and precision of the fingerprinting process. This difference translates into three different collection strategies:

1. **Triggers:** If an extension has a unique trigger that is not shared with any other extension, it is sufficient to test if the style of the trigger is different from the one of the baseline. The identification is fast as there is no need for additional data processing.

2. **Triggers and properties:** If several extensions share the same trigger, it can be enough to collect the list of modified properties to identify each of them. For example, for extensions modifying a link element, one extension may increase the size of the font while another may change the background color. By identifying which properties of the styled element were modified, one can differentiate between the two extensions.

3. **Trigger, properties, and values:** This last strategy is the one that produces the most data but it can lead to more precise results as you one can attribute a specific change directly to the right extension.

Table 3 shows the number of fingerprintable extensions depending on the chosen strategy. Strategies 2 and 3 offer an improvement of 6% and 15% respectively from Strategy 1 but albeit at a slightly higher performance cost as more data is collected and processed. When comparing the use of computed styles and dimensions, the numbers are comparable between the two with no major differences. Dimension changes, however, could be sensitive to differences between devices particularly when the database of fingerprints was generated with a device that had a much larger screen, compared to the one that is being fingerprinted. One possible solution is to have multiple databases of dimension-related style fingerprints so that the fingerprinting algorithm can match the ones that are the closest to the user's own screen size. We view this as an implementation detail to make the fingerprinting

Table 3: *Numbers of extensions found to be fingerprintable via two CSS-originating leakages (i.e., computed styles and changed dimensions) separately and together. Three implementation strategies give different number of uniquely attributed extensions.*

Fingerprinting Strategy	***Change of Computed Styles***	***Change of Dimensions***	***Union***
Strategy 1: Unique $(trigger)$	3,865	3,866	3,866
Strategy 2: Unique $(trigger, parameters)$	4,088	3,927	4,090
Strategy 3: Unique $(trigger, parameters, values)$	4,412	4,162	4,446

Table 4: *Distribution of the number of users across fingerprintable and non-fingerprintable extensions*

	Percentile			
	.25	*.50*	*.75*	*.99*
Fingerprintable	10.0	71.0	754.0	219,420.5
Non-fingerprintable	6.0	41.0	681.0	637,104.5

process more robust and hence we consider it as out of scope for this paper.

5.3 Statistics on fingerprintable extensions

Mix between unique and shared triggers Out of the 4,446 uniquely identifiable extensions, 3,475 of them have at least one trigger that is not shared with any other extension. This means that the fingerprinting process for them is fast and straightforward as a script only has to check a single trigger for any difference in style compared to a baseline element. For 846 extensions, they share all their triggers with other extensions but the changed properties and values are still unique to them. Finally, for 125 extensions, they are detectable because of the unique combination of non-unique triggers they change.

Distribution of popularity Looking at the number of users in Table 4, there is no significant difference between fingerprintable and non-fingerprintable extensions. Both categories have extensions with few users as well as extensions with more than 10 million users. If we look closely at extensions with more than 100,000 users, 68 of them are vulnerable to style fingerprinting while 28 of them are not. Overall, we do not observe a correlation between the popularity of an extension and its fingerprintability as it is mainly tied to its functionality and how it was coded.

Modified properties Injected styles can modify a wide range of properties in HTML elements. Table 6 in Appendix B list the top 50 properties that are the most modified by fingerprintable extensions. We want to highlight here some of our findings.

At the top of the list are `perspectiveOrigin`, `transformOrigin`, `webkitPerspectiveOrigin` and `webkitTransformOrigin`. Even though few extensions in our dataset explicitly set values for these properties,

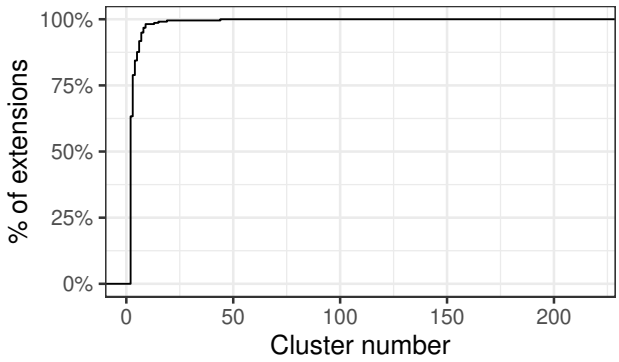

Figure 4: *CDF graph of the distribution of collisions between non-unique extensions that inject CSS.*

96.8% of the tested extensions presented changes in them. Many of these properties expose high-precision values (e.g. six floating-point digits, such as "951.5px 0.046875px") which unfortunately lead to extensions being uniquely fingerprintable because of them. In terms of dimensions, the `width` and `height` of an element are high on our list with 96.0% and 84.2% of extensions affecting these properties, respectively. Interestingly, color-related style changes are not as common as we originally expected with the first color-related property (`backgroundColor`) being on the 24^{th} position of Table 6.

5.4 Understanding non-uniquely fingerprintable extensions

Here, we investigate the reasons why some extensions that inject CSS rules are not uniquely fingerprintable.

Distribution of collisions Figure 4 presents the distribution of the clusters of collisions we have in our dataset. Most of them are between a very small number of extensions as confirmed by the long tail in our graph. Out of 218 different clusters, 138 (63.3%) are between two extensions and 34 (15.6%) are between three. The 5 largest collision clusters we identified are of size 44, 19, 15, 13 and 9.

Reasons for collisions We manually analyzed 50 different extensions to understand why two extensions would share the

same style fingerprint. Our findings reveal that the majority of collisions are due to very specific development behaviors:

- **Same name with different IDs:** Several extensions can have different IDs but they share the exact same name. One example is the "Antalyx Desktop Sharing" Chrome extension which has 4 different IDs in our dataset. They are linked to the same developer but every new version was uploaded as a brand new extension instead of an update of an existing one.

- **Same developer with different variants:** Extensions can have different IDs but they are simply variants coming from the same developer. One example is the "Bonusway{.se,.ro,.cz...}" extension that is available in 13 different variants. The code across all extensions is identical but each of them embeds its own locale file for the interface. Another example comes from a series of "Safe Site" extensions we identified that only presented a difference in the branding. At first, we thought they belonged to different companies as each of them linked to different websites: *Ultra VPN*, *Total AV*, *Safe VPN*, *ScanGuard*, *PC Protect* and *Privacy Web*. Yet, looking at their terms of use revealed that all of them belong to the same group called *Protected.net*.

- **Copies:** Extensions can simply be a copy of another extension that was uploaded to the store. One example of such case is with "Privasee" that is a copy of an older version of the "DuckDuckGo Privacy Essentials" extension.

- **Same libraries:** Extensions can share fingerprints if they use the exact same list of libraries. Several extensions in our dataset are only injecting styles based on *jQuery*: "jquery-ui.css" and "jquery.qtip.css". Moreover, if an extension builds on top of other well-known libraries, this can lead to additional collisions. For example, the "uPerform® In-application Help" extension that is installed by more than 90,000 users relies on "jquery-ui" to build its UI foundation. One of the triggers generated by our pipeline is the following:

```
<div id="ancile-csh" class="ancile-csh">
  <div trigger="yes" class="ui-front"></div>
</div>
```

  The inner *div* will be triggered by all extensions with "jquery-ui" while the outer one will only be triggered by "uPerform".

- **Coincidence:** Sometimes, two extensions share the same fingerprint for no reason other than pure coincidence. We detected one case in our whole dataset where two extensions have completely different goals but they share one identical CSS rule. The "ePubby" and "Link Shortcuts" extensions share the same rule on elements of class `css-isolation-popup`.

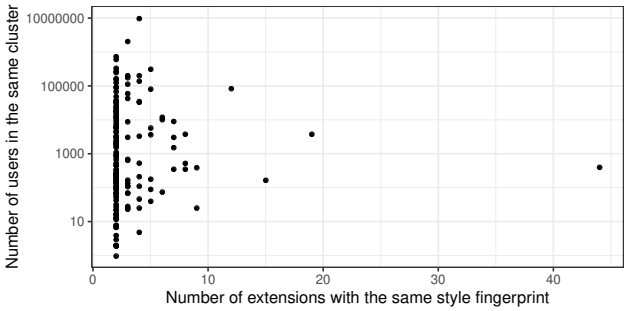

Figure 5: *Total number of users in clusters of extensions sharing an identical style fingerprint*

Impact of collisions on identifiability Being able to discover the exact list of extensions installed in a browser can contribute to the overall device fingerprint and render its user identifiable. Yet, there is a large difference between detecting an extension shared by millions of users with one shared by a few tens of users. To understand whether the extensions that share style fingerprints have similar populations of users, we investigate the impact of collisions on the identifiability of their users.

In Figure 5, we clustered together the extensions with the same fingerprint and combined their userbase to understand how many users are present in each cluster. It should be noted that we did not get the number of users for all extensions as some of them were not available in the Chrome store at the time of writing. We can see that there is no direct correlation between the number of extensions in a cluster and the total number of users. For example, there are 2,106,549 users in a cluster containing 3 extensions while there are 411 users in the one containing 44. Then, some clusters have as many as 10 million users while others can have as few as two users. In the end, if the goal is to uniquely identify users, detecting a group of several extensions can provide a lot more discriminating information than detecting a single extension that is shared by many users. Note that this discussion focuses entirely on the discriminatory power of browser extensions, in terms of differentiating users from each other. Orthogonally to this issue, even extensions that are shared by millions of users can reveal sensitive socioeconomic characteristics of their users.

5.5 Performance Benchmarks

In this section, we quantify the real-world performance of our proposed extension fingerprinting method. Our evaluation is based on our proof-of-concept fingerprinting script, which we used for our video demo (`https://vimeo.com/430428308`) that we mentioned in Section 3. Specifically, we measure the time our script takes to detect random subsets of the 20 extensions that we used for the video demo, with subset sizes varying from 1 to 20. We set up our script so that the detection logic runs inside the `window.onload` event listener (i.e., the

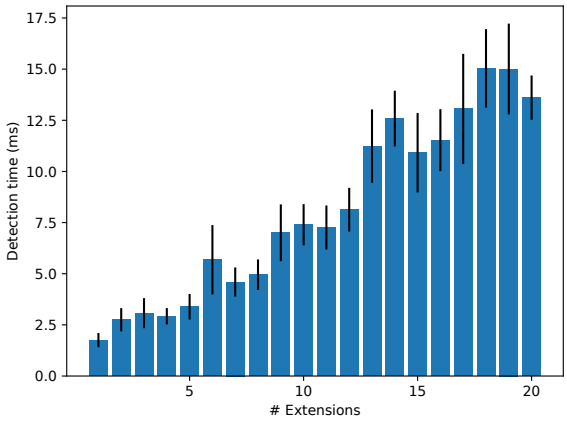

Figure 6: *Average detection time for different numbers of installed extensions, with whiskers representing 95% confidence intervals.*

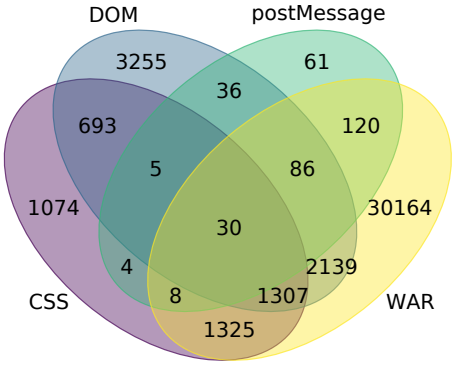

Figure 7: *Venn diagram showing the number of extensions detectable by four fingerprinting techniques. Our newly-proposed method can detect 1,074 extensions which are "invisible" to all other methods.*

detection script is triggered after the page has loaded), and we use the `performance.now` API for timing the execution of our detection script (a start timestamp is taken on entrance to the `window.onload` listener, and an end timestamp is taken when all extensions in the subset are detected, and the detection time is the difference of these two timestamps). The tests were run on a laptop with Intel Core i7-6600U CPU and 12GB RAM. For each subset size from 1 to 20, we measure the detection time 10 times and take the average.

Figure 6 shows the results of our benchmarks. One can see an overall upward trend as the number of extensions being fingerprinted increases, while the variations are likely attributed to changing system load during the measurement. The increase in detection time as the number of fingerprinted extensions grows is due to the fact that more trigger elements need to be compared against their baseline (recall that for each trigger we compare both the style rules returned by `getComputedStyle`, as well as its position and dimensions). However, note that even for 20 extensions, our detection script still finishes in around 15 milliseconds. Therefore, the real-world performance overhead of this fingerprinting vector is clearly not going to be a hindrance against trackers using it to fingerprint thousands of popular extensions.

5.6 Comparison with related work

Prior work has explored different ways to detect browser extensions: probing for Web Accessible Resources (WAR) [24], detecting DOM modifications [47], and capturing messages sent by *postMessage* [29, 45]. Figure 7 reports on the fingerprintability of our complete dataset by each of these techniques, including our newly proposed, CSS-based extension fingerprinting.

In total, CSS fingerprinting can uniquely detect 4,446 extensions. Only 30 extensions are covered by all methods and 1,074 extensions are now detectable through our CSS-based fingerprinting that were previously "invisible" to all other fingerprinting techniques. If WAR fingerprinting were to disappear, akin to the randomization of UUIDs present in Firefox [8], CSS fingerprinting would be the only one to cover an additional 1,325 extensions. Overall, Figure 7 shows that there is no ultimate method to detect all browser extensions as different techniques are able to fingerprint different sets of extensions. Our findings are inline with the ones reported by Karami et al. on a dataset of 102,482 extensions [29].

5.7 Longitudinal analysis

Lastly, to understand whether the injection of CSS rules by extensions is a new phenomenon, we analyze a Chrome extension dataset that spans five years (mid 2014 to mid 2019). It comprises 501,349 extensions which is reduced to 426,807 after excluding themes and apps. For an average month, our dataset includes 4,384 new/updated extensions, with the store size increasing from around 22K in 2014 to more than 116K in 2019. Figure 8 presents the percentage of extensions injecting CSS out of all collected extensions for this five-year period. One can observe that the percentages of extensions injecting CSS rules on *all* and *some* domains are largely stable over time.

As a separate experiment, for the 4,446 universally fingerprintable extensions discovered in this paper, we tested their corresponding detection triggers after about a year. Overall, we discovered that, as of June 2020, only 940 extensions were updated, i.e., 79% of extensions have the same style fingerprints as they had a year ago. Out for the 940 extensions that updated at least once, after re-running our testing pipeline, 776 triggered at least one of their previously discovered triggers. In other words, 82.5% still remain fingerprintable despite their updates.

Overall, when we consider these two experiments together, we can conclude that i) extensions that are currently fingerprintable are likely to remain fingerprintable to CSS-based fingerprinting, and ii) the trigger database that a tracker would

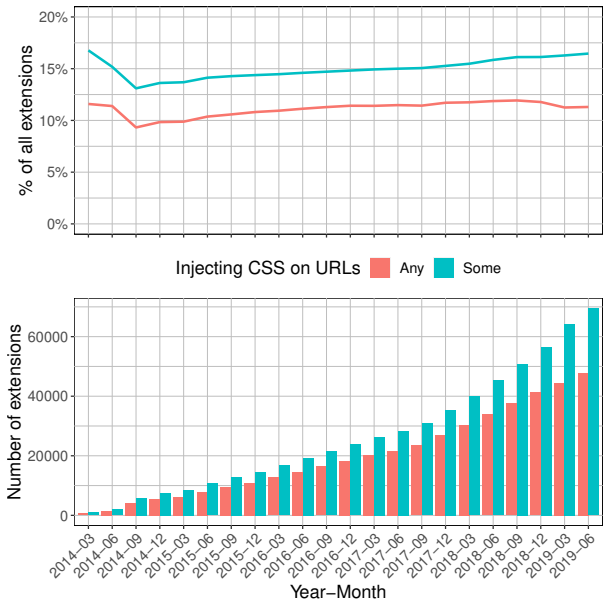

Figure 8: *Extensions injecting CSS styles (into any visited web page, or only on some specific URLs), shown over all collected extensions in the Chrome Web Store from 2014 to 2019 at three-month intervals.*

need to compile for the fingerprintable extensions can remain effective for more than a year, before it would need to be updated.

6 Countermeasures

Given the power of CSS-based extension fingerprinting, in this section, we discuss possible countermeasures against it. First, we examine how the `getComputedStyle` API that this new fingerprinting technique relies on, is currently used in the wild, and whether it is possible to simply remove support for this API. Second, we present the design and evaluation of an in-browser countermeasure that defends against this type of attack, by hiding the effects of extension-originating styles from the pages on which they are active.

6.1 Can `getComputedStyle` be removed?

To measure the prevalence of `getComputedStyle` usage and, more importantly, understand its uses cases in the current web, we crawl the Alexa top 100K websites using VISIBLEV8, an open-source tool which adds instrumentation to Chromium so that all JavaScript API accesses during runtime are logged [27]. In total, we found that 1) there are 61,414 unique scripts (as distinguished by their SHA-256 hashes) that use the `getComputedStyle` API (hereafter for convenience we refer to these as `getComputedStyle` scripts), 2) these `getComputedStyle` scripts are served from 60,375 distinct TLD+1 domains, and 3) 76,638 out of the top 100K websites

Table 5: *Top 10 TLD+1 domains that serve scripts that use the `getComputedStyle` API, by the number of script inclusions. In our crawl, we observe a total of 283,516 such inclusions.*

TLD+1 Domain	*# Inclusions*	*% All*	*# SHA256 (% All)*
googlesyndication.com	54,966	19.39%	62 (0.10%)
facebook.com	14,950	5.27%	9,727 (15.84%)
ajax.googleapis.com	14,027	4.95%	135 (0.22%)
doubleclick.net	11,930	4.21%	28 (0.05%)
twitter.com	7,201	2.54%	4 (0.01%)
adsafeprotected.com	6,077	2.14%	2,588 (4.21%)
youtube.com	5,182	1.83%	29 (0.05%)
vidible.tv	4,497	1.59%	24 (0.04%)
2mdn.net	4,006	1.41%	198 (0.32%)
cloudflare.com	3,059	1.08%	411 (0.67%)
Total	145,979	51.49%	13,952 (22.72%)

in our crawl contain at least one `getComputedStyle` script (i.e., in 76.64% of the crawled websites). By *inclusion counts*, Table 5 shows the top 10 TLD+1 domains that served the most `getComputedStyle` scripts. These domains alone account for 51.49% of all such script inclusions. For reference, we also list in Table 5 the number of unique `getComputedStyle` scripts (i.e., by SHA-256 hash) served from each domain.

Next, to shed light on the current usage scenarios of `getComputedStyle` and whether it is already being used for browser fingerprinting in the way that we describe in this paper, we conducted a manual analysis of representative sets of scripts that used `getComputedStyle`. These sets of scripts include: 1) the top scripts (by inclusion counts, as identified by SHA-256 hash of the script) from the top 10 TLD+1 domains that served the most `getComputedStyle` scripts; 2) similar to the first set, but here we focus on the top 10 scripts served from URLs blacklisted by EasyList/EasyPrivacy (EL/EP); and lastly, 3) a random sample of 20 unique scripts (again distinguished by their SHA-256 hashes) out of all of the `getComputedStyle` scripts in our crawl. We categorize their use cases in the rest of this section.

The use cases we present in the following paragraphs are not intended to be an exhaustive list of all `getComputedStyle` use cases from our sample scripts, but rather our best-effort manual analysis of these scripts, given that many of them are obfuscated and/or minified. The primary purpose of this section is to establish that 1) the fingerprinting technique that we present in this paper cannot be mitigated by simply removing the `getComputedStyle` API given that API's widespread usage, and 2) to demonstrate that we did not find evidence of this fingerprinting technique already being used in the wild.

Wrapper for Getting Element Styles The first category of `getComputedStyle` usage we describe is a class of wrapper functions that encapsulate `getComputedStyle`, along with `Element.style` and `Element.currentStyle`. Listing 6 shows one such wrapper from our manually examined samples that encapsulates `getComputedStyle`. The primary

Listing 6: *JS snippet showing the use of getComputedStyle as part of a cross-browser compatibility layer*

```
function get_element_style_property(elem, property
    ) {
  var value;
  if (elem.currentStyle)
    value = elem.currentStyle[property];
  else if (window.getComputedStyle)
    value = window.getComputedStyle(elem).
        getPropertyValue(property);
  else
    value = elem.style[property];
  return value;
}
```

roles of these wrapper functions are two-fold: 1) they serve as a cross-browser compatibility layer for reading the style sheets of an HTML element (e.g., `Element.currentStyle` is a proprietary version of `getComputedStyle` and available only on old versions of Internet Explorer, which do not support `getComputedStyle`), and 2) they provide a way to read the element's *inline* style as fallback when the `getComputedStyle` method is removed by scripts (e.g., by invoking `delete window.getComputedStyle`).

Note that as shown in Listing 6, besides their primary roles mentioned above, these wrapper functions often offer the added convenience of returning the value of a particular CSS property specified as one of the wrapper's arguments.

Compatibility Tests The `getComputedStyle` API is also used for compatibility testing. In such cases, CSS rules are set for an element injected by the script on-the-fly, and the script then immediately reads back the CSS properties of the element using `getComputedStyle`. One example of this is found in the popular jQuery, where the code sets the CSS property `top` to be `1%` and then checks whether the read-back value is in pixels. The reason for this test is that for certain CSS properties (e.g., `top`), some browsers will return their percentage values rather than absolute pixel values (see [7]), while the rest of the script is expecting pixel values.

Visibility Testing Another category of use cases for `getComputedStyle` is to test the visibility of an element on the page by checking, for example, if the value of the CSS property `display` is set to `none` (which means the element is not rendered on the page). Besides `display`, the properties `visibility` and `opacity` are often also included in these types of checks, as well as element dimensions, e.g., checking if the value of the `width` property is zero.

Adblocker Detection We have observed a few cases from our sample where the script is detecting whether the user has installed an adblocking extension. Specifically, the script accomplishes this by injecting an element with an ID or class name targeted by the filter rules of the adblocker, and checks whether the adblocker prevents the injected element from being displayed on the page (e.g., by using visibility testing methods that we described). In total, we observed this behavior in three out of the 40 sample scripts that we manually examined (all three scripts are identified by EL/EP as trackers). Although this method of adblocker detection is conceptually similar to what we describe in this paper, an important difference is that ad-blockers are expected to hide content and therefore checking for the absence of ad-like elements is a straightforward technique, variations of which were known as early as 2011 [32]. Contrastingly, our technique generalizes over all types of extensions (not just ad-blockers) and allows for the precise identification of an extension, as opposed to merely knowing whether an ad-blocker is present or absent.

Toggling Style Properties Lastly, there is also a category of `getComputedStyle` usage that probes for and toggles the displayed visual properties of elements on the page (e.g., toggles the visibility of an element by first checking whether the `visibility` property is set to `hidden`, and if so set it to `visible`).

6.2 Hiding Extension Effects

Given that we cannot just retire the `getComputedStyle` API, an alternative method for protecting users is to break the link between the injected content styles and the values returned by the `getComputedStyle` function. This would effectively hide the presence of extensions from webpages and therefore protect the users of browser extensions from being fingerprinted. This hiding can be done at different layers in the browser, each with its advantages and disadvantages.

In this section, we explain how a browser extension can replace the default `getComputedStyle` function with one that ignores the styles injected by extensions. In Appendix A, we provide the details of an alternative solution that modifies the browser in order to achieve the same results. Our hope is that, once browser vendors confirm that this is an issue worth tackling, that these details can provide a roadmap for the changes that need to happen.

Browser extension The biggest advantage of a browser extension is that it is lightweight and easy to distribute but it is limited to a finite set of browser APIs. Yet, making direct modifications to the DOM can provide a robust protection against CSS-based, extension fingerprinting, thanks to the existence of Shadow DOMs. Figure 9 provides a high-level overview of our approach.

A Shadow DOM is a hidden tree in the DOM that can be attached to elements in the regular DOM tree. Its purpose is to isolate all of its content from the regular DOM tree: IDs, names and styles do not "leak out" from Shadow DOMs and elements from the regular DOM tree also do not "bleed in." This feature was primarily introduced for developers to avoid naming conflicts when designing Web Components and we

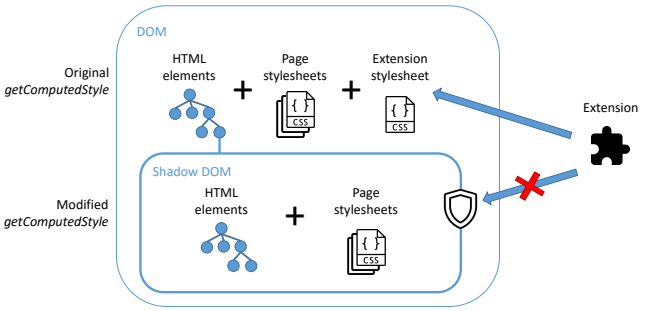

Figure 9: *Difference between the original and the modified getComputedStyle function.*

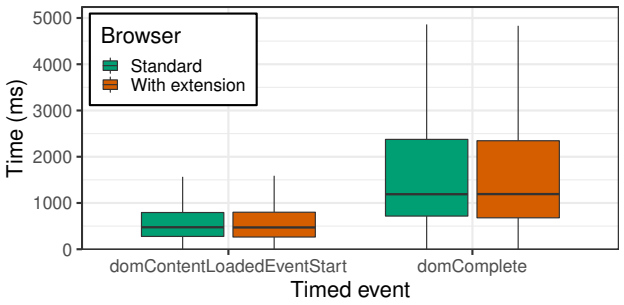

Figure 10: *Impact of the countermeasure on the loading times of webpages.*

can leverage it to modify the behavior of `getComputedStyle`. When injected as a content script on page loads, our extension performs the following actions:

1. Attach a new Shadow DOM to the document body.
2. Copy the complete regular DOM tree into the Shadow DOM. This creates a mirrored version of the regular DOM with all inline styles and all page style sheets. Content styles from extensions are not present as they do not have a physical presence in the regular DOM. They are applied seamlessly by the browser and, as such, cannot be copied into the Shadow DOM.
3. Modify the code of `getComputedStyle` to use the Shadow DOM. When the function is called on a element in the regular DOM, the modified function will look for the copy of this element in the Shadow DOM and execute the original `getComputedStyle` function on it. For optimization purposes, we only reroute calls on elements that have an ID or a class from one of the installed extensions.

In the end, the computed style will be the exact same as the one from the regular DOM element but without any modifications from content styles. A video showing our extension in action is available here: `https://vimeo.com/430428277`

Evaluation and performance In order to evaluate the performance of our browser extension and identify any potential breakage, we crawled the homepage of the Tranco top 200 websites [39] with and without our countermeasure. We used Puppeteer [12] to pilot a Chrome web browser on a laptop with an Intel i7 processor running on Ubuntu 19.10 and we collected the following information:

- **Loading times:** We used the `PerformanceNavigationTiming` API to collect the `responseEnd`, `domContentLoadedEventStart` and `domComplete` properties. These three metrics help us calculate the overhead imposed by our solution as they focus on the processing of documents and scripts after all major HTTP requests have been performed. They are independent of network speed, congestion, and other issues that could impact our measurements.

- **JavaScript errors:** To identify if the injected code disrupts the natural flow of JavaScript code execution, we collected JavaScript errors directly from the browser. By checking the number of errors with and without the extension, we can see if the countermeasure causes any new breakage issues that were not there before.

- **Screenshots:** As an extra verification step, we took screenshots of all visited pages with and without our browser extension, to check that our extension does not introduce any potential side effects with visible artefacts. Since the visited webpages include news websites with ever-changing, featured stories as well as dynamic ads, we opted to perform this verification manually.

We repeated our measurements five times with and without our extension to average the loading times and smooth out any unusual discrepancies. The results are presented in Figure 10.

Looking at the loading times, both boxplots are almost identical with a difference between mean values of less than 0.5%. In terms of JavaScript errors, only *reuters.com* presented additional errors when our extension was present (6 with and 0 without). By analysing the script that crashed, we found that `getComputedStyle` was called on a $< g >$ container in a SVG element that lacked an essential property that was used in our extension's logic. After adding one additional check, we revisited the same website and discovered no errors. Finally, looking at screenshots with and without the extension, we observed no noticeable differences between the two crawls apart from changes in the dynamic content.

Given the near-zero performance overhead, the lack of new JavaScript errors, and the visual confirmation that pages were not affected by our extension, we argue that our countermeasure protects against style fingerprinting with minimal impact on the overall user experience.

7 Related work

Browser fingerprinting has received signification attention from the research community over the last decade. Eckersley [19], Laperdrix et al. [30] and Gómez-Boix et al. [23] showed that it can be used to identify users on the Internet even though this may prove difficult at a very large scale. Moreover, later studies quantified the use of fingerprinting on the public web and showed its growing adoption by popular sites [14, 15, 20, 38]

Extension fingerprinting attacks Prior work has also investigated the specific problem of fingerprinting browser extensions. Sjosten et al. [44] demonstrated how Web Accessible Resources (WARs) could be abused to enumerate the presence of specific browser extensions. Gulyás et al. [24] built on their findings and performed a study on 16,393 users to understand how WAR fingerprinting contributes to users' uniqueness. They found that 54.86% of users with at least one detectable extension could be uniquely identified. Orthogonal to the use of WARs, Starov and Nikiforakis [47] looked at the fingerprintability of extensions through DOM modifications. With a tool named XHound, they tested the 10,000 most popular Chrome extensions and found that 9% of them introduce modifications that are detectable on any domain. Sanchez-Rola et al. [42] used a timing side-channel to infer the presence of any browser extension installed in the browser, even if they are disabled in incognito mode. Van Goethem and Joosen [49] presented in the same year a variation of this attack to link a user's isolated browsing sessions. These side channels have been fixed by the Chromium team [3, 4] and can therefore no longer be used for extension fingerprinting. Finally, Karami et al. [29] recently introduced a tool called Carnus to automate the creation and detection of extension fingerprints. They combine both WAR and behavioural fingerprints but also add inter and intra-communication based enumeration. Out of 102,482 extensions, they can detect 29,428 of them.

To the best of our knowledge, we are the first to show that injected style sheets can be used for detecting installed browser extensions, and to measure the vulnerability of extensions in the wild. As we showed in Section 5.6, this technique allowed us to fingerprint more than 1,000 extensions which were "invisible" to all other current methods of extension fingerprinting.

Extension fingerprinting defences Three studies have presented extensive designs to mitigate extension fingerprinting. Sjosten et al. [43] propose a defence system called Latex Gloves to prevent WAR fingerprinting. Extensions are repackaged to modify the whitelist of websites on which they can run and a special extension blocks unauthorized probing through the `webRequest` API. Starov et al. [45] also uses a whitelist to enforce strict access to browser extensions resources. Both of these approaches can mitigate our presented attack by basically turning off an extension on an undesired website. However, it remains unclear whether users are capable of configuring these whitelists and what is the real protection that these mechanisms offer, in the presence of multiple JavaScript third parties in popular sites who can take advantage of the trust associated with the first-party website.

CloakX by Trickel et al. follows a different approach for protecting extensions against fingerprinting [48]. It randomizes what makes an extension identifiable while maintaining equivalent functionality, i.e., it randomizes the path of web accessible resources to prevent WAR probing attacks, it changes the behavioural fingerprint by changing ID and class names that are injected, and it adds a proxy to handle dynamic references to randomized elements. CloakX does not account for styles and therefore cannot stop our new CSS-based, extension-fingerprinting attack.

8 Conclusion

Stateless tracking significantly affects the privacy of web users and has recently received increased attention by researchers and browser vendors. In this paper we focus on the CSS rules that browser extensions inject in visited web pages as part of their logic and show how these rules can be abused to identify a user's installed extensions. To understand the magnitude of this problem, we developed a pipeline that leverages both static and dynamic analysis of browser extensions in order to identify a set of triggers that can be used for CSS-based, extension fingerprinting. Our analysis of 116,485 extensions revealed that 4,446 (3.8%) of them can be uniquely identified on any webpage based on the styles they inject. We investigate how the involved browser APIs are used in the wild, propose concrete countermeasures that browser vendors can adopt to mitigate this problem, and provide a countermeasure solution via a browser extension that demonstrates our defense mechanism.

Availability

The artifact accompanying this paper can be found at https://github.com/plaperdr/fingerprinting-in-style. Our defense prototype can be installed and tested on a demo page in a Chromium-based browser. We also provide the complete set of 4,446 extensions detectable through style fingerprinting along with the generated trigger pages.

Acknowledgements

We thank the anonymous reviewers for their helpful feedback. This project is partially funded by the Hauts-de-France region in the context of the ASCOT project of the STaRS framework, by the National Science Foundation (under awards CNS-1941617, CNS-1703375 and CNS-1813974), and by the Office of Naval Research under grant N00014-20-1-2720.

References

[1] :visited support allows queries into global history - Mozilla Bug Tracker. https://bugzilla.mozilla.org/show_bug.cgi?id=147777, 2002.

[2] Keep visited links private so that history info isn't leaked. - Webkit Bug Tracker. https://bugs.webkit.org/show_bug.cgi?id=24300, 2009.

[3] Issue 611420: WebAccessibleResources take too long to make a decision about loading if the extension is installed. https://bugs.chromium.org/p/chromium/issues/detail?id=611420, 2017.

[4] Issue 709464: Detecting the presence of extensions through timing attacks (including Incognito) - Chromium bug tracker. https://bugs.chromium.org/p/chromium/issues/detail?id=709464, 2017.

[5] CSS Cascading and Inheritance Level 3 - W3C Candidate Recommendation. https://www.w3.org/TR/css3-cascade/#cascading-origins, 2018.

[6] Stylish - Custom themes for any website - Chrome Web Store. https://chrome.google.com/webstore/detail/stylish-custom-themes-for/fjnbnpbmkenffdnngjfgmeleoegfcffe, 2019.

[7] Bug 29084 - getComputedStyle returns percentage values for left / right / top / bottom . https://bugs.webkit.org/show_bug.cgi?id=29084, 2020.

[8] Chrome incompatibilities – Mozilla | MDN. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Chrome_incompatibilities#web_accessible_resources, 2020.

[9] Controlling CSS Animations and Transitions with JavaScript. https://css-tricks.com/controlling-css-animations-transitions-javascript/, 2020.

[10] Element.currentStyle. https://developer.mozilla.org/en-US/docs/Web/API/Element/currentStyle, 2020.

[11] Match Patterns. https://developer.chrome.com/extensions/match_patterns, 2020.

[12] Puppeteer: Headless Chrome Node.js API - GitHub. https://github.com/puppeteer/puppeteer, 2020.

[13] Window.getComputedStyle(). https://developer.mozilla.org/en-US/docs/Web/API/Window/getComputedStyle, 2020.

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

[15] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. FPDetective: Dusting the Web for fingerprinters. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.

[16] Gaurav Aggarwal, Elie Bursztein, Collin Jackson, and Dan Boneh. An analysis of private browsing modes in modern browsers. In *Proceedings of the 19th USENIX conference on Security*, pages 6–6. USENIX Association, 2010.

[17] Andrew Clover. CSS visited pages disclosure - BUGTRAQ mailing listposting. https://seclists.org/bugtraq/2002/Feb/271, 2002.

[18] Quan Chen and Alexandros Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[19] Peter Eckersley. How Unique Is Your Browser? In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, pages 1–17, 2010.

[20] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2016.

[21] Cristiano Giuffrida, Stefano Ortolani, and Bruno Crispo. Memoirs of a browser: A cross-browser detection model for privacy-breaching extensions. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 10–11. ACM, 2012.

[22] Nicolas Golubovic. Attacking browser extensions. Ruhr-Universitat Bochum, Volume 3, 2016. https://golubovic.net/thesis/master.pdf.

[23] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. Hiding in the Crowd: an Analysis of the Effectiveness of Browser Fingerprinting at Large Scale. In *WWW 2018: The 2018 Web Conference*, Lyon, France, April 2018.

[24] Gabor Gyorgy Gulyas, Doliere Francis Some, Nataliia Bielova, and Claude Castelluccia. 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, pages 14–27, 2018.

[25] Lin-Shung Huang, Zack Weinberg, Chris Evans, and Collin Jackson. Protecting browsers from cross-origin CSS attacks. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 619–629, 2010.

[26] Artur Janc and Lukasz Olejnik. Feasibility and real-world implications of web browser history detection. *Proceedings of W2SP*, 2010.

[27] Jordan Jueckstock and Alexandros Kapravelos. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2019.

[28] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 641–654, San Diego, CA, August 2014. USENIX Association.

[29] Soroush Karami, Panagiotis Ilia, Konstantinos Solomos, and Jason Polakis. Carnus: Exploring the privacy threats of browser extension fingerprinting. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, 2020.

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

[31] Zhuowei Li, XiaoFeng Wang, and Jong Choi. SpyShield: Preserving privacy from spy add-ons. In *Recent Advances in Intrusion Detection*, pages 296–316. Springer, 2007.

[32] Keaton Mowery, Dillon Bogenreif, Scott Yilek, and Hovav Shacham. Fingerprinting information in JavaScript implementations. In Helen Wang, editor, *Proceedings of W2SP 2011*. IEEE Computer Society, May 2011.

[33] Keaton Mowery and Hovav Shacham. Pixel perfect: Fingerprinting canvas in HTML5. In *Proceedings of the Web 2.0 Security & Privacy Workshop*, 2012.

[34] Mystique Analyzer. https://mystique.csc.ncsu.edu/about.

[35] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: Large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 736–747, 2012.

[36] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. PriVaricator: Deceiving Fingerprinters with Little White Lies. *Research.Microsoft.Com*, 2014.

[37] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Proceedings of the IEEE Symposium on Security and Privacy*, SP '13, pages 541–555, 2013.

[38] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 541–555, 2013.

[39] Victor Le Pochat, Tom van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. 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, 2019.

[40] John Resig. Pro JavaScript Techniques, 2006.

[41] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and defending against third-party tracking on the web. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 12–12, Berkeley, CA, USA, 2012. USENIX Association.

[42] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. Extension breakdown: Security analysis of browsers extension resources control policies. In *26th USENIX Security Symposium*, pages 679–694, 2017.

[43] Alexander Sjösten, Steven Van Acker, Pablo Picazo-Sanchez, and Andrei Sabelfeld. Latex Gloves: Protecting Browser Extensions from Probing and Revelation Attacks. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

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

[45] Oleksii Starov, Pierre Laperdrix, Alexandros Kapravelos, and Nick Nikiforakis. Unnecessarily Identifiable: Quantifying the Fingerprintability of Browser Extensions Due to Bloat. In *The World Wide Web Conference*, WWW, 2019.

[46] Oleksii Starov and Nick Nikiforakis. Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1481–1490. International World Wide Web Conferences Steering Committee, 2017.

[47] Oleksii Starov and Nick Nikiforakis. XHOUND: quantifying the fingerprintability of browser extensions. In *2017 IEEE Symposium on Security and Privacy, SP 2017*, pages 941–956, 2017.

[48] Erik Trickel, Oleksii Starov, Alexandros Kapravelos, Nick Nikiforakis, and Adam Doupé. Everyone is Different: Client-side Diversification for Defending Against Extension Fingerprinting. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[49] Tom Van Goethem and Wouter Joosen. One side-channel to bring them all and in the darkness bind them: Associating isolated browsing sessions. In *WOOT*, 8 2017.

[50] Michael Weissbacher, Enrico Mariconti, Guillermo Suarez-Tangil, Gianluca Stringhini, William Robertson, and Engin Kirda. Ex-Ray: Detection of history-leaking browser extensions. In *Annual Computer Security Applications Conference (ACSAC)*, 2017.

A Countering style fingerprinting at the browser level

While browser extensions are lightweight and can easily be installed, their scope of actions is limited to the available WebExtension APIs. A built-in protection can go beyond in terms of flexibility and performance by having its logic directly integrated with native code. We also argue that this problem should be fixed directly by browser vendors to protect all their users from style leakage. To that end, we provide here a blueprint of the modifications that could be made to prevent style leakage through extensions.

Overview Figure 11 provides information on how the browser can be modified to provide protection. The approach is similar in essence to the one applied to fix the *visited* history leakage [1,2,17] but extended in many ways to fulfill our goal. Throughout the entire page rendering pipeline, the only stage that needs to be changed is the *Style* one. It is responsible for collecting all style sheets and computing the style for each individual element. In a nutshell, to prevent style leakage,

Step	Action	Modifications needed	C++ classes
1	Gather all style rules and index them	Add support for "ExtensionAuthor" and "ExtensionUser" origins for a CSS rule	Document, WebDocument, StyleEngine
2	Visit each element and see what styles apply to them	-	Document, Element, StyleResolver
3	Combine rules to get the final computed style	Compute two styles: one with and one without content styles	StyleResolver, StyleBuilder, CSSProperty

Style computation

Step	Action	Modifications needed	C++ classes and methods
1	getComputedStyle call in JavaScript	-	V8Window
2	getComputedStyle call in the window context	Add support for an "allowExtensionStyles" boolean	LocalDomWindow
3	Get the right style value depending on the context	Add a switch that selects the right computed style	CSSComputedStyleDeclaration CSSProperty

Style retrieval

Figure 11: *Overview of the built-in browser modifications.*

the browser needs to maintain two computed styles for each element: one with the style sheets from installed extensions and one without.

Maintaining two computed styles Each style rule applied on a webpage has one of three different CSS cascade origins [5]:

- *Author Origin*: this origin belongs to rules contained in the source document or in external style sheets.
- *User Origin*: it comes from rules that the user has specified for a specific document (set through a special interface or with an extension like Stylish [6]).
- *User Agent Origin*: this is the default style provided by the browser. This style can be modified if the user changes the default fonts or accessibility options.

These origins are important as they determine which rule has priority over another one. Introducing additional origins with new priorities is not appropriate as it will make the overall design of a webpage even more complicated for developers. Instead, we propose to extend the first two cascade origins with two additional ones: *Extension Author Origin* and *Extension User Origin*. They will have the exact same priority as their non-extension counterpart but will carry the additional information that they originate from a browser extension. This way, thanks to a custom Style resolver, the `StyleFromElement` function can properly compute two separate styles and maintain them throughout the lifetime of the HTML element.

Modifying getComputedStyle Now that two distinct computed styles exist, we need to modify the `getComputedStyle` function to direct it to the right style depending on the execution context. We propose to add a boolean called "allowExtensionStyles" that can be propagated up to each CSS property to select the proper value to return. For example, if `getComputedStyle` is executed in a standard webpage, a "false" value will be propagated to prevent style leakage. In the context of using Chrome DevTools for debugging an application, a "true" value will be sent, allowing the user to see the true computed value with extension styles.

Protection at the *Layout* stage Some extensions may introduce custom style sheets that have a direct impact on the size of an element. For example, by changing the relative width of an element from 20 to 30%, its actual size will change at the *Layout* stage and could be detected by a malicious script. To counter this problem, we can go even further by combining our approach with the one proposed by Nikiforakis et al. in [36]. In it, they introduce randomization policies that can be used to modify specific attributes of HTML elements. In our case, we can use a policy to randomize an element's dimensions to prevent such leakage.

B Top modified properties by fingerprintable extensions

Table 6: *List of the top 50 properties ranked by the number of extensions modifying them with injected styles*

Property	Count
perspectiveOrigin	4302
transformOrigin	4302
webkitPerspectiveOrigin	4302
webkitTransformOrigin	4302
inlineSize	4268
webkitLogicalWidth	4268
width	4268
position	3749
blockSize	3743
height	3743
webkitLogicalHeight	3743
top	3662
left	3631

Property	Count
background	3610
right	3583
bottom	3538
border	3439
borderBlockEnd	3403
borderBottom	3403
webkitBorderAfter	3403
borderInlineStart	3390
borderLeft	3390
webkitBorderStart	3390
backgroundColor	3387
borderInlineEnd	3361
borderRight	3361

Property	Count
webkitBorderEnd	3361
borderBlockStart	3355
borderTop	3355
webkitBorderBefore	3355
borderColor	3348
borderBlockEndColor	3304
borderBottomColor	3304
webkitBorderAfterColor	3304
borderInlineStartColor	3303
borderLeftColor	3303
webkitBorderStartColor	3303
borderInlineEndColor	3269
borderRightColor	3269

Property	Count
webkitBorderEndColor	3269
borderBlockStartColor	3265
borderTopColor	3265
webkitBorderBeforeColor	3265
padding	3166
zIndex	3166
font	3152
paddingInlineStart	3052
webkitPaddingStart	3052
paddingLeft	3051
paddingInlineEnd	2983