Everyone is Different: Client-side Diversification for Defending Against Extension Fingerprinting

Erik Trickel, *Arizona State University;* Oleksii Starov, *Stony Brook University;*
Alexandros Kapravelos, *North Carolina State University;* Nick Nikiforakis,
Stony Brook University; Adam Doupé, *Arizona State University*

https://www.usenix.org/conference/usenixsecurity19/presentation/trickel

This paper is included in the Proceedings of the 28th USENIX Security Symposium.

August 14–16, 2019 • Santa Clara, CA, USA

978-1-939133-06-9

Open access to the Proceedings of the 28th USENIX Security Symposium is sponsored by USENIX.

Everyone is Different: Client-side Diversification for Defending Against Extension Fingerprinting

Erik Trickel[⋆]*, Oleksii Starov*[†]*, Alexandros Kapravelos*[‡]*, Nick Nikiforakis*[†]*, and Adam Doupé*[⋆]

[⋆]*Arizona State University*
{etrickel, doupe}@*asu.edu*

[†]*Stony Brook University*
{ostarov, nick}@*cs.stonybrook.edu*

[‡]*North Carolina State University*
akaprav@*ncsu.edu*

Abstract

Browser fingerprinting refers to the extraction of attributes from a user's browser which can be combined into a near-unique fingerprint. These fingerprints can be used to re-identify users without requiring the use of cookies or other stateful identifiers. Browser extensions enhance the client-side browser experience; however, prior work has shown that their website modifications are fingerprintable and can be used to infer sensitive information about users.

In this paper we present CloakX, the first client-side anti-fingerprinting countermeasure that works without requiring browser modification or requiring extension developers to modify their code. CloakX uses client-side diversification to prevent extension detection using anchorprints (fingerprints comprised of artifacts directly accessible to any webpage) and to reduce the accuracy of extension detection using structureprints (fingerprints built from an extension's behavior). Despite the complexity of browser extensions, CloakX automatically incorporates client-side diversification into the extensions and maintains equivalent functionality through the use of static and dynamic program analysis. We evaluate the efficacy of CloakX on 18,937 extensions using large-scale automated analysis and in-depth manual testing. We conducted experiments to test the functionality equivalence, the detectability, and the performance of CloakX-enabled extensions. Beyond extension detection, we demonstrate that client-side modification of extensions is a viable method for the late-stage customization of browser extensions.

1 Introduction

As the web expands and continues being the platform of choice for delivering applications to users, the browser becomes a core component of a user's interactions with the web. Modern browsers advertise a wide range of features, from cloud-syncing and notifications to password management and peer-to-peer video and audio communications. An important feature of modern browsers is their ability to be extended by users, as they see fit, by installing *browser extensions*. Namely, Google Chrome and Mozilla Firefox, the browsers with the largest market share, offer dedicated browser extension stores that house tens of thousands of extensions. In turn, these extensions advertise a wide range of additional features, such as enabling the browser to store passwords with online password managers, blocking ads, and saving articles for later reading.

From a security perspective, the ability to load third-party code into the browser comes at a cost, even though extensions rely on web technologies such as HTML, JavaScript, and CSS. Browsers afford extensions significantly more privileges than they do to a webpage. For example, the same origin policy restricts webpages from accessing content, such as a cookie, that does not originate from the same domain. For a webpage to bypass this restriction, it must implement cross-origin resource sharing, whereas extensions may not only access resources of any domain but may also alter the content. Historically, malicious extensions abuse these privileges to perform advertising fraud and to steal private and financial user data [22, 28, 44, 47].

Next to security issues, using browser extensions can also lead to the loss of privacy. Given that users choose the extensions to install, it is possible to make inferences about a user's thoughts and beliefs *based solely on the extensions she keeps.* For example, the detection of a coupon-finding extension [1] reveals information about the user's income-level. Additionally, an extension that hides articles about certain political figures [20, 21] reveals the user's political leanings. Lastly, the use of browser extensions may provide a means for websites to persistently identify a user over the course of distinct browser sessions.

Although browser vendors do not offer any programmatic methods for a webpage's JavaScript to detect the extensions currently installed in a user's browser, researchers recently discovered side-channel techniques for fingerprinting many extensions. Sjösten et al. were the first to demonstrate a new method for detecting browser extensions that exploited the public nature of *web-accessible resources* (WARs) [38]. A

WAR is any resource (e.g., JavaScript or image) within an extension that the extension identifies as externally accessible. As a result, a webpage can determine whether a visitor uses an extension by requesting one of the exposed WARs. Sjösten et al. showed that more than 50% of the top 1,000 browser extensions use WARs, which any webpage might use to detect extensions. Later, Starov and Nikiforakis demonstrated another technique for fingerprinting extensions that uses an extension's modifications to the document-object-model (DOM) to detect their presence [42]. The authors developed XHOUND, a system that automatically discovers the DOM side-effects of extensions. Through their experiments, they showed that more than 10% of the top 50K extensions were fingerprintable.

One approach to reducing fingerprintable extensions is through education and developer training. However, historically, developers — and web developers in particular — ignore even well-known security concerns. Even after nearly 20 years, the most common website vulnerabilities are still SQL injection vulnerabilities [43]. Therefore, it is unlikely that asking extension developers to make their extensions less fingerprintable will have the desired effect on the ecosystem.

To empower users to protect their own privacy, in this paper we propose CloakX, a client-side countermeasure against extension detection using fingerprints. Instead of trying to remove the fingerprintable attributes of extensions, our approach is to automatically alter, randomize, and add to these attributes without requiring web browser modifications or any involvement from the extension's developer. Through these modifications, CloakX diversifies the extension's anchorprints, which are fingerprints consisting of items that can be accessed directly from a webpage, and structureprints, which are fingerprints that embody the structural changes an extension makes to a webpage (for more details refer to Section 2.2). On the surface, client-side diversification of the fingerprintable attributes seems straightforward; however, the dynamic nature of JavaScript and the complexity of the browser extension's architecture necessitated a complex approach that relies on both static and dynamic program analysis.

CloakX uses static and dynamic analysis techniques to automatically diversify the extension's fingerprint without modifying the browser, without requiring any changes by the extension's author, and without altering the extension's functionality. To diversify the extension's anchorprint, CloakX automatically renames WARs, IDs, and class names and corrects any references to them in the extension's code, which severs the link between the published extension and the currently installed version. In addition to static changes, the diversification is also performed by our dynamic DOM proxy (Droxy), which intercepts DOM modifications from the extension's code and makes the changes on-the-fly. To diversify the extension's structureprint, Droxy also injects random tags, attributes, and custom attributes into each webpage, which obfuscates the extension's structureprint. As a result, an extension cloaked by CloakX is undetectable by a webpage using anchorprints and is obfuscated from a webpage using structureprints; however, from the user's point of view, the extension operates the same.

In summary, we make the following contributions:

- We present the design of a novel system that automatically identifies and randomizes browser extension fingerprints to defend against existing extension fingerprinting techniques without requiring any browser changes or any involvement from the extension's developer.
- We describe the implementation of our design into a prototype, CloakX, that uses a combination of: (1) static rewriting of extension JavaScript code and (2) a dynamic DOM proxy, Droxy, that intercepts and rewrites extension requests on-the-fly.
- We use a combination of high-fidelity testing (extensive manual testing) and low-fidelity testing (broad automated testing) on the extensions rewritten by CloakX to quantify the breakage caused by our system, demonstrating that client-side modification of extensions introduces minimal defects.
- We also evaluate the detectability of cloaked extensions and show that some cloaked extensions are undetectable while others are more difficult to detect.

2 Background

In this section, we provide insights into the complexity of modern browser extension frameworks that must be taken into account when designing a client-side countermeasure against extension fingerprinting. We start by describing the architecture of browser extensions, focusing on the details that pertain to their fingerprintability. Next, we discuss fingerprinting and detecting extensions using anchorprints (fingerprints that are comprised of items directly accessible from a tracking webpage's JavaScript) and structureprints (fingerprints built from the extension's behavior). Last, we finish this section by presenting the threat model that CloakX can defend against.

2.1 Browser Extensions Explained

While modern web browsers provide an ever-increasing range of functionality to users and webpages, an off-the-shelf browser cannot possibly provide a sufficiently large set of features to satisfy every user's browsing needs. To improve the user's browsing experience, browsers enable users to enhance their functionality through extensions. Users add extensions to their browsers to change the browser's look, to add helpful toolbars, to block ads, and to enhance popular webpages [5].

Although extensions utilize web technologies such as HTML, CSS, and JavaScript, they also have access to powerful extension-only APIs that enable them to, among others, access and modify cross-origin content and a browser's

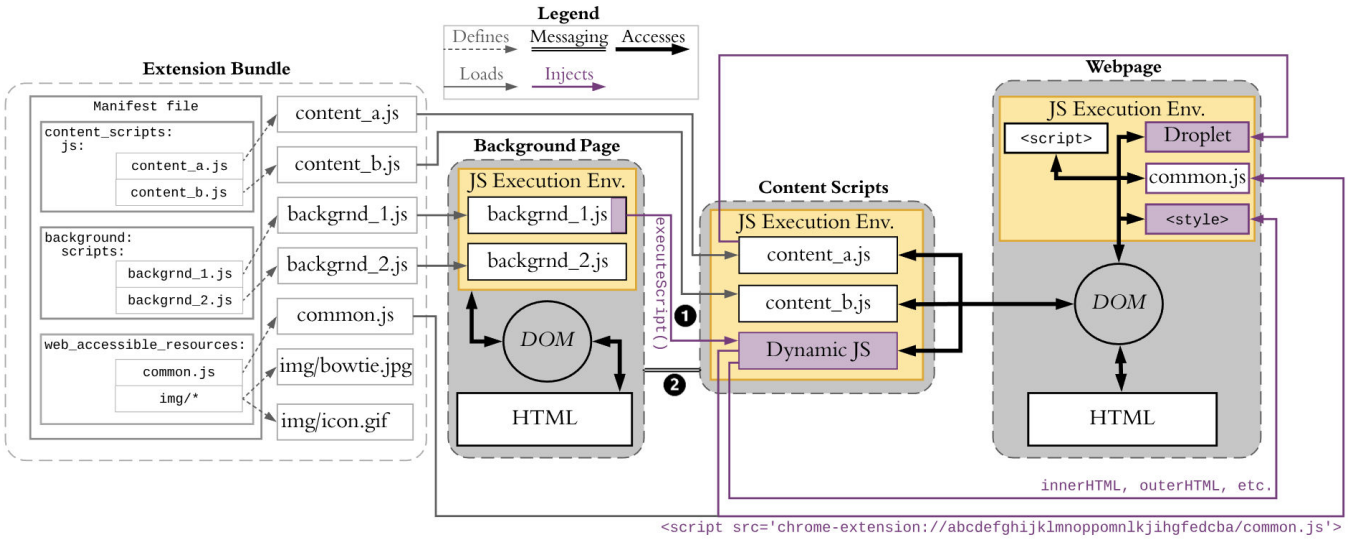

Figure 1: Extension architecture. A high-level overview of Chrome's extension architecture with the static content of the extension on the left side and the multiple execution environments on the right. Background pages can 1) inject content scripts dynamically using the executeScript() *method in Chrome's extension API and 2) send and receive messages from the content scripts.*

client-side storage. However, before an extension can access broader privileges or interact with a webpage, it must request this access from the browser. As Figure 1 depicts, the modern extension architecture implements a layered security approach within the browser that creates multiple execution environments with varying levels of persistence and privileges for each extension and webpage.

The left-hand side of Figure 1 depicts the static parts of an extension, including items such as the manifest, JavaScript, HTML, and image files. For the browser to parse and install an extension, it must have a *manifest* file which defines the extension's properties. Similar to the manifest shown in Figure 1, extensions commonly rely on three properties, which describe background pages, content scripts, and web accessible resources [5].

When the *background* property is included in the extension's manifest, the browser automatically constructs a hidden background page for the extension. The background page contains HTML, a DOM, and a separate JavaScript execution environment (labeled as "Background Page" in Figure 1). The JavaScript executed in the background page often contains the main logic of the extension, maintains long-term state, and operates independently from the life-cycle of the webpages [28].

Content scripts bridge the gap between the background page and the current webpage. An extension uses content scripts to modify the current webpage and communicate with the background page. These content scripts are either statically declared by an extension in the manifest file or programmatically injected into the current webpage. For example, on the left-hand side of Figure 1, the manifest declares the two content scripts content_a.js and content_b.js. To programmatically inject a content script, an extension must call executeScript() from a background page (see ❶ in Figure 1).

To modify a webpage, a content script uses the webpage's DOM [10]. DOM APIs provide a systematic way for interacting with a webpage. In this paper, we call the content script's interaction with the DOM APIs *DOM requests*.

Notice in Figure 1 that the background page, content scripts, and webpage each run their own JavaScript execution environment. The separate execution environments prevent the JavaScript variables and functions from directly interacting. Google Chrome's documentation states that content scripts "live in an isolated world, allowing a content script to make changes to its JavaScript environment without conflicting with the page or additional *content scripts*" [3] (emphasis added). This statement, however, is misleading because we experimentally discovered that content scripts loaded from the same extension share variables and can call functions from other content scripts. Thus, an extension's content scripts share a single execution environment; however, they do not share an environment with the background page, webpage, or other extensions (depicted in Figure 1).

Using DOM requests, a content script has significant control over the rendered webpage. Content scripts can inject HTML into the webpage (using DOM element properties such as innerHTML or DOM methods such as appendChild()). We call this injected HTML *droplets* (the extension drops them onto the webpage). Among other elements, droplets may contain <script> tags where the extension includes either inline or remote JavaScript. By injecting JavaScript, the content script purposefully bypasses the isolation between the content scripts and the webpage's execution environments.

The Chrome Extension API provides privileged functionality available only to extensions. Chrome grants back-

ground scripts broad access to the API's capabilities. However, Chrome grants content scripts limited access to the API while making the API inaccessible to webpages. For example, only an extension's background page can access network resources, view platform information, and communicate with native applications. However, both content scripts and background scripts may use the API to initiate and listen for communications from one another via the appropriate Chrome APIs (as shown by the double lines towards the bottom of Figure 1). Background scripts cannot directly interact with a webpage, however they can *indirectly* send messages to it via the extension API using the method chrome.runtime.sendMessage() [2]. Part of the reason for this layered security model, including the separate execution environments, is to isolate the components and prevent webpages from unauthorized access to the extension API's more sensitive functions.

Another important property in the manifest is the *web-accessible-resources* property [7]. Prior to January 2014, Chrome permitted external access to all of an extension's resources, i.e., a webpage could reference resources belonging to installed extensions. In more modern versions of Google Chrome, an extension must explicitly whitelist a resource before a webpage may retrieve it [8]. An extension whitelists its resources by adding them to the *web-accessible-resources* property in the manifest. Once added, a resource becomes accessible to any webpage or any installed extension.

To access a web accessible resource (WAR) from the context of a web page, a webpage developer uses a URL of the format: chrome-extension://[extId]/[path-to-resource]. The extId in the URL is a unique identifier generated by the Google Web Store upon publication of an extension which does not change when extensions are updated.

2.2 Extension Fingerprinting and Detection

In 2017, Sjösten et al. demonstrated that, with WAR fingerprinting, any extension using WARs is trivially detectable by a webpage [38] by creating a database of which WARs are utilized by each extension available in the Google Store. Given that an extension's ID is globally unique and permanent, a tracker can detect an extension by requesting any one of its previously identified WARs. If the request is successful, then the corresponding extension is installed on the user's browser. Next to its simplicity and the 16,479 (28%) of extensions that utilize WARs (and are thus fingerprintable), WAR fingerprinting works in the browser's private mode.

Orthogonally to WAR fingerprinting, Starov et al.'s Extension Hound (XHOUND) [42] creates a DOM fingerprint based on the extension's DOM modifications. XHOUND uses dynamic analysis to exercise extensions and detect changes introduced to the DOM through the extension's operation. By loading a set of webpages with and without a given extension, XHOUND can compare the two resulting DOMs and isolate the DOM changes that were performed by the given extension. These changes can straightforwardly be converted into fingerprints which trackers can use to detect the presence of any DOM-modifying extension.

When using WAR and DOM fingerprints for detection of extensions, we reclassify all such fingerprints into *anchorprints* and *structureprints* to describe the method and accuracy of the detection techniques. Anchorprints rely on an *anchor* between the webpage's JavaScript and the extension. An anchor is a unique identifier formed to facilitate access and communication between webpages and extensions. An anchor provides a way to directly access elements and resources available to the webpage. Some examples of anchors include WARs, IDs, class names, and custom attributes. For example, the Chrome extension Grammarly adds a unique class to the root <html> element on each webpage. Thus, if a webpage uses document.getElementsByClassName() and receives the <html> element, it is likely the user has Grammarly installed.

An anchorprint is comprised of all the WARs, IDs, class names, and custom attributes made available by an extension. With the items in an anchorprint, a webpage need only to query the DOM or send an XMLHttpRequest to detect an extension. WARs are the most powerful of the anchorprint elements because, due to the unique extension identifier, an anchorprint with even one WAR is always 100% accurate. Although IDs, class names, and custom attributes might be 100% accurate, they often have a much lower per element accuracy than WARs because webpages and extensions alike often use some of the same names. Despite this limitation, the accuracy of the anchorprint improves dramatically with each additional element included in it.

Structureprints are less precise (in terms of fingerprinting) but are formed based on the *structure* of the changes the extension makes to the underlying webpage. Structureprints effectively create a DOM fingerprint that uses the extension's unique and intended behavior to identify the extension. The idea of a structureprint is that it can be used to detect a specific extension because the extension always behaves in a predictable manner and alters a webpage consistently, thus creating a *structure* that is unique among extensions. For instance, consider a popular Google Calendar extension that is the *only* extension with a structureprint that contains the tags a and img with the following attribute names href, location, target, blank, width, height, src, alt and style Surprisingly, we found during our experiments that a tracking webpage can reliably detect 28.93% (1,511) of extensions using only the tagName of the DOM elements added or deleted from a webpage by an extension. Adding attribute names, attribute values, and the text of the DOM elements to the structureprint increases the number of detectable extensions to 73.65% (3,847).

An important *subset* of structureprints that target an extension's behavior are called *behaviorprints*. For example,

Grammarly creates a green button inside a `text area`. With manual analysis, it is possible to identify whether the green button has been added to the webpage without relying on the IDs or class names injected by Grammarly. Another example of using behaviorprints are in the detection of ad-blocking extensions, such as Detect AdBlock [4]. However, no recent research has shown how to create a behaviorprint in an automated way at scale. As a result, current behaviorprints are limited to targeted attacks against specific extensions or narrowly constrained categories of extensions (e.g., ad-blocking extensions).

Beyond the obvious implementation differences between anchorprints and structureprints, the fingerprint classes differ in their accuracy and their destructibility. For most anchorprints, matching the WAR, ID, class name, and custom attributes of a published extension often provides a (unique) one-to-one match. However, for structureprints, finding a match is often less certain because many extensions have similar behavior, which results in the same structureprint. Another key difference between anchorprints and structureprints is the permanence of their link between the published extension and the user's installed version. For anchorprints using WARs, IDs, and class names, CloakX completely renames the values. By renaming the values, CloakX completely destroys the link between the published extension and the user's installed version. Without that link, it is impossible for a tracking webpage to use the anchorprint to identify the installed extension because the anchorprint no longer matches the published extension. Whereas with structureprints, the destruction of the link between the published extension and the user's installed version is difficult. This difficulty occurs because of the requirement that a cloaked extension retain the same behavior (i.e., user experience). By maintaining the same behavior, the structureprint of a cloaked extension is only being obfuscated, which means that with enough effort a tracker can eventually deobfuscate the cloaked structureprint and, thus, detect the cloaked extension.

2.3 Threat Model

In our threat model, attackers use a database of fingerprints to detect the extensions installed by a visitor to the site. However, we limit the attackers to the information and privileges afforded to the webpage's JavaScript execution environment. In essence, we assume that there are no zero-day vulnerabilities that would allow webpages to bypass the layered-security architecture depicted in Figure 1. Therefore, the attackers cannot access the content of an extension installed on a visitor's device.

In this paper, we explore two different types of attackers. The automated attacker uses automated extension detection techniques. Specifically, we limit the automated attacker to anchorprints and structureprints. To detect an extension, the automated attacker must find either an exact or fuzzy match to an entry in their fingerprint database. The targeted attacker is permitted to manually generate targeted structureprints using portions of the structureprint (i.e., behaviorprints) for extension detection. While we focus on defending against the automated attacker because automated large-scale detection is a feasible attack, we also include the targeted attacker to explore how CloakX can defend against the targeted attacks.

3 CloakX

The core idea behind CloakX is to diversify each extension's fingerprint from the client-side while maintaining equivalent functionality *without making any changes to the browser and without requiring the developers to alter their extensions*. Client-side diversification of the anchorprints (fingerprints comprised of items directly accessible from a tracking webpage's JavaScript) and structureprints (fingerprints built from the extension's behavior) reduces the extension's detectability by breaking a webpage's ability to link together a published extension and the one installed on the user's machine. CloakX defeats detection using an anchorprint by randomizing the names of the WARs, IDs, and classes. However, CloakX does not completely defeat anchorprint detection using custom attributes. CloakX's approaches combat custom attribute-based detection by randomly injecting more unique custom attributes into each webpage. CloakX reduces the efficacy of structureprints by introducing random attributes and tags into the webpage. Although CloakX does not completely prevent detection using custom attributes or structureprints, it is a step beyond current solutions and CloakX achieves these protections without any changes to the browser and without requiring the intervention of extension developers.

Figure 2 shows the overall process of CloakX, a multiphase tool that leverages static- and dynamic-analysis techniques to achieve extension diversification while maintaining functional equivalence. In the first phase, CloakX analyzes the extension for the DOM fingerprints and CloakX identifies the droplets that must be statically analyzed. In the second phase, CloakX renames each WAR within the extension to a unique random value, finds all the references to the original name, and replaces them with their randomized counterpart. In the third phase, CloakX adds a dynamic proxy (Droxy) to the extension's content and background scripts. Droxy dynamically intercepts DOM and WAR requests and substitutes the original ID, class names, and WAR names with their random counterparts. In the last phase, CloakX statically analyzes and rewrites the DOM IDs and class names inside droplets that cannot be dynamically intercepted by Droxy.

3.1 XHOUND Analysis

CloakX uses XHOUND (we obtained a copy of the XHOUND prototype by contacting the paper's authors [42]) to generate a DOM fingerprint for the extension and to identify the droplets

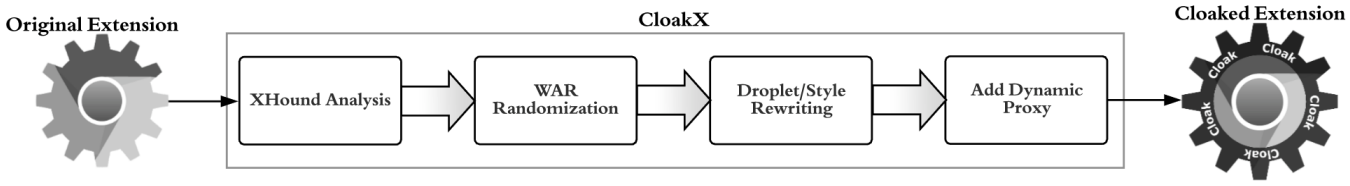

Figure 2: Overview of the CloakX process.

injected into the webpage. Each DOM fingerprint consists of four types of artifacts: (1) adding a new DOM element, (2) deleting a DOM element, (3) setting or altering an element's attribute, and (4) changing text on the page.

Of the four types, DOM additions are the most common type of detectable artifacts according to XHOUND [42]. This is because DOM additions are generic operations that often rely on loose coupling with a webpage for them to be triggered. Whereas most DOM modifications or deletions require a tighter coupling between the extension and the webpage, which limits their applicability to the problems often solved by developers. For instance, consider a password manager extension that injects a stylized element into every password form field (so that the user can invoke the password manager interface). The extension adds the element to the webpage and gives it a unique ID and a custom class name. It requires the ID to communicate with the element once it's placed on the webpage. Using the added ID and class name (i.e., the extension's anchorprint), a webpage can detect the extension by checking for the presence of either the unique ID or class name on the webpage.

Next, CloakX uses XHOUND to identify any droplets the extension injects into the webpage's execution environment so that CloakX can preprocess the droplets to identify the ID and class names within them. As discussed in Section 2.1, droplets (purple-colored boxes in Figure 1) are JavaScript strings that an extension injects directly into a rendered webpage. Droplets can include any text literal such as HTML, JavaScript, or base64-encoded images; however, the preprocessing is only performed on droplets containing inline JavaScript and those <script> elements that reference WARs.

Finally, during this phase, CloakX creates a map from the original ID and class names used to fingerprint the extension to the new randomized values.

3.2 Diversification of Web-Accessible Resources (WARs)

The principle behind the diversification of Web-Accessible Resources (WARs) is straightforward: if each installation of an extension has different filenames for the same WARs, then a tracker can no longer create a global database of WARs and, therefore, can no longer detect the presence or absence of any given extension based on its WAR anchorprint.

In the first stage of the WAR diversification process, CloakX identifies all the resources declared as WARs in the manifest file of each extension. Although many extensions explicitly list the resources they wish to make accessible, it is also possible to use a * wildcard [34]. With wildcards, an entire folder, its contents, and all its subfolders can be designated as web-accessible — this includes using a single *, which designates every file in the extension as web-accessible. Even though making every file in the extension web-accessible is likely an implementation error, we discovered 419 extensions that made all of their resources web-accessible, out of 59K analyzed extensions. In the second stage, CloakX computes the shortest unique file path to facilitate the search-and-replace in the final stage. Specifically, CloakX reduces the full path of each WAR to the minimum length necessary to uniquely identify the resource (compared to all the other resources in the extension). This operation reduces the number of resource references missed (i.e., false negatives) associated with dynamic string concatenation (often a directory path).

In the final stage of the WAR diversification process, CloakX uses the shortest unique path to find every use of the WAR within the extension's files and to replace that with the appropriate random value, maintaining the correctness of WAR references for each extension.

In addition to the static alterations described above, CloakX relies on Droxy, discussed in the Section 3.3, to dynamically translates any WAR requests missed by the static replacement method.

3.3 Droxy

The next step in the CloakX process adds Droxy to the extension. Droxy is a content script that injects random attributes and tags into the DOM to further obfuscate the extension's DOM fingerprint while also translating any uncloaked WAR requests and the IDs and class names used in DOM requests into their cloaked versions. CloakX patches Droxy into the extension and configures Droxy to execute before any of the extension's content scripts.

Droxy adds random attributes and tags to the DOM to reduce the accuracy of detection using structureprints. As each webpage is loaded, Droxy adds a random number of randomly generated tags to the DOM to make extension detection less accurate. To further frustrate detection using structureprint matching, Droxy adds random attributes to the DOM elements added by each extension.

Droxy also uses cross injection of custom attributes to frustrate anchorprint detection. For trackers using custom attributes to detect extensions, cross injection allows the user to impersonate other extensions, which increases a tracker's false positives when using anchorprint detection. This is done by adding custom attributes that are randomly selected from a list of the 244 unique custom attributes used by other extensions with a DOM fingerprint.

Droxy also dynamically catches any WAR requests made using the resource's original filename, which serves as a backup for the static replacement method described in Section 3.2. Droxy achieves this by watching for changes to the DOM using a MutationObserver() that checks for uncloaked WAR requests inside the DOM elements altered by the extension. In addition, Droxy overrides the XMLHttpRequest.open() method and adds functionality to translate any WAR requests for the original filename to the new, randomized filename.

Droxy translates the ID and class names used to create a DOM anchorprint. As the first content script to load, Droxy overrides DOM accessor and mutator methods before the extension uses them to interact with the DOM, which effectively wraps all DOM requests in a translation layer (blue area in Figure 3). Each of the overridden methods are augmented to intercept and translate ID and class names used to create the DOM fingerprint. Droxy determines which ID and class names to translate by checking the ID and class names against the cloaking map, created in Section 3.1. The cloaking map contains name–value pairs where each XHOUND-discovered ID and class name is paired with a randomized version. If it finds a match in the cloaking map, it translates the original value on-the-fly into the randomized version. By intercepting and translating the fingerprintable ID and class names to randomized values, Droxy alters the extension's DOM fingerprint from the perspective of a tracker's execution context breaking the link between the user's installed extension and the publicly available version.

To prevent the use of anchorprint detection, Droxy translates IDs and class names into random values according to the map created in Section 3.1. For ID and class name translation, Droxy also tracks DOM queries and DOM mutations. Droxy intercepts and inspects the extension's queries that use IDs, element names, class names, and query selectors, which include the methods getElementById(), getElementsByName(), getElementsByTagName(), getElementsByClassName(), querySelector(), and querySelectorAll(). To handle more complex query selectors, Droxy parses the selectors using the open-source Sizzle engine to accurately identify the ID and class names [9].

For DOM mutations performed via JavaScript, Droxy intercepts all the ways in which an ID or class name can be introduced to the DOM. This dynamic interception of ID and class names is done by overriding setAttribute() and getAttribute() methods and redefining id and className properties to use the overridden setAttribute() and getAttribute(). In addition, Droxy overrides the classList property. Because classList is an object, Droxy overrides the add(), contains(), and remove() methods of the classList. As a result, Droxy translates the extension's use of IDs and class names whether it is done when a DOM element is created or modified.

For DOM mutations performed via the injection of raw HTML, Droxy uses static and dynamic analysis to make the translation of ID and class names straight-forward and precise. Droxy overrides the methods used to inject raw HTML, such as the innerHTML property and insertAdjacentHTML() method. Droxy uses the browser to parse the HTML by creating a mock container and adding the HTML to it without attaching the mocked container to the DOM. Droxy queries the mock container to identify and transform the ID and class names into their randomized versions. Droxy then exports the string representation from the mock container's DOM and then calls the original method to apply the modified string to the webpage's DOM.

In addition to DOM queries and mutations, Droxy intercepts styles and translates on-the-fly. An extension can include styles via text content inside <style> or CSSStyleSheet's methods such as addRule() or insertRule(). Once intercepted, Droxy uses CSS parsing to locate the IDs and class names. If found, Droxy replaces the ID or class name with its randomized counterpart.

Droxy replaces an extension's droplets with the statically rewritten version (content_a.js and Dynamic JS in Figure 3). As a part of the droplet rewriting process described in Section 3.4, Droxy receives a hash value of the original droplet and modified version of the code for each droplet used by the extension. Droxy then matches the current droplet's hash to the ones provided and replaces it with its cloaked counterpart. Droxy performs the matching and replacement by customizing the properties textContent, innerText, and HTMLScriptElement's and the methods append() and appendChild() This process is depicted by the dashed arrow near ❶ in Figure 3. Droxy relies on the preprocessed JavaScript because rewriting the code on-the-fly in the browser efficiently is currently infeasible.

3.4 Static Droplet Rewriting

As discussed previously and shown in Figure 1, Droxy cannot intercept a droplet with dynamically inserted JavaScript because when the inserted code is executed in the webpage's JavaScript execution environment. Unfortunately, Droxy is also unable to cloak the droplet before inserting it because the heavy-weight static analysis necessary would significantly degrade the extension's performance. Therefore, CloakX statically analyzes the droplets offline, identifies where the extension adds the fingerprintable ID and class names to the

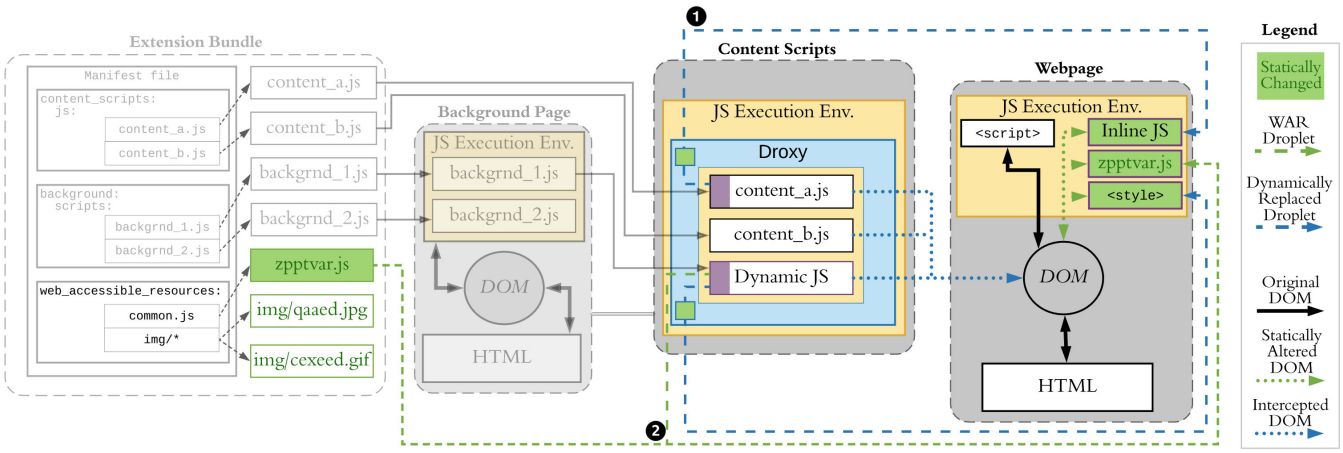

Figure 3: Diversified CloakX rewritten extension. CloakX hides fingerprints by rewriting the droplets, content styles, and renaming of web-accessible resources (WARs) and through Droxy's on-the-fly substitution. As a result, a tracking webpage cannot access the original identifiers; however, the internal logic of the extension still can because Droxy translates those requests.

DOM, rewrites the JavaScript code, and Droxy dynamically substitutes the original code with its rewritten counterpart.

Extensions commonly use *generic* values for IDs and class names, which often overlap with JavaScript keywords or JavaScript code constructs that refer to the class names and IDs dynamically. In addition, the expressiveness of JavaScript means that the ID and class names usage are context-sensitive. For example, if the fingerprintable class name is content, CloakX should only replace the instance of #content and ignore element.content, content.maximizer, and content-shaper, as each have a different semantic meaning. Developers often construct ID and class names dynamically in the code, which necessitates a more sophisticated form of static analysis. For example, an extension might attempt to access an element with the ID content by using getElementById("con" + "tent"), which would be missed by a regular expression searching for the full word.

CloakX statically rewrites droplets offline (i.e., before an extension is installed) using static analysis to identify the appropriate locations in the JavaScript. CloakX limits its rewrites to the ID and class names that occur in the JavaScript and are added to the webpage via the DOM. By identifying and only altering these DOM altering instances, CloakX limits the possibility of breaking the extension with the alterations. In essence, the static rewriting requires a tool that performs taint analysis where it labels DOM interactions as sinks and then analyzes the backward slices of the control flow graph (CFG) until it finds the fingerprintable IDs and class names as sources.

We decided to use TAJS — a state-of-the-art and feature rich JavaScript analyzer — as the program analysis core of the CloakX static rewriting. We chose TAJS because it (1) performs type analysis on JavaScript, (2) supports most of the ECMAScript 5 standard and DOM functionality, (3) is under active development, (4) is open source [6], and (5) is the product of recent research [13, 14, 15, 23, 24, 25, 27].

TAJS performs dataflow analysis by using techniques that examine the flow of data along program execution paths. As TAJS iterates over the CFG, it creates a semilattice of program states that are unique for each basic block in the CFG [26]. For each variable represented in the lattice at a given basic block, TAJS assigns a set of possible values. The dataflow analysis completes when the values inside the lattice reach a fixed point and no longer change with each iteration. Using these values, it is possible to follow data both forwards and backwards through the CFG [26].

3.4.1 TAJS for Extensions

We enhanced TAJS to support static rewriting of the droplets by adding support for Chrome extensions, adding DOM taint analysis, and maximizing its exploration of the CFG. In addition, we plan to make our changes to TAJS publicly available because there are currently no other program analysis tools for browser extensions.

We added extension support to TAJS by creating stubs for Chrome's extension API and implementing support for necessary methods such as sendMessage(), getURL(), executeScript(), onMessage.addListener() .

We implemented taint analysis within TAJS that tracks data through an application until it reaches a sink, where a sink is a location of interest within the CFG [16]. For the purposes of this analysis, TAJS tracks string literals matching the fingerprintable IDs and class names through the CFG until they are used to interact with the DOM. As a part of the taint tracking, we added functionality that maintains an *audit trail* of the changes to each variable while traversing the CFG so that upon reaching a sink CloakX can trace the values of interest to their origins.

We increased TAJS's code coverage by adding edges to the end of the CFG that force a call to every named and anonymous function defined within the code. For the purposes

of extension rewriting, it is necessary that TAJS analyzes all the JavaScript within a droplet because some functions appear unreachable without complete semantic understanding of Chrome's extension execution environment. However, the dynamic aspects used by TAJS itself to strike a balance between soundness and precision came at the cost of code coverage [15]. For example, TAJS does not analyze functions unless they are called by the JavaScript and the call is also reachable from the beginning of the CFG. Because extension rewriting requires TAJS to analyze all of the JavaScript within a droplet, we added edges to the end of the CFG that simulates a call to every named and anonymous function in the droplet. The potential downside to adding the edges is the decreased precision of our analysis (i.e., we are adding behavior to the application that does not exist at run-time), however for the purposes of identifying DOM fingerprints the trade-off is acceptable.

3.4.2 Static Analysis Results

Automated analysis of real-world JavaScript code is a difficult problem and despite all the advances made by TAJS, it, as well as similar tools, cannot analyze some JavaScript programs. As a JavaScript program increases in complexity and size, it becomes increasingly less likely TAJS will complete the analysis due to the explosion of dataflows (i.e., the classic state space explosion problem). As acknowledged by the authors, TAJS initially targeted hand-written JavaScript applications of a "few thousand lines of code" [26]. Plus, the addition of the fake edges dramatically increased the complexity of the CFG and the number of states, which decreased the code TAJS could successfully analyze to about 1,000 lines of code.

Fortunately, CloakX only needs TAJS analysis for the 197 extensions using droplets, which is only 3.2% of the extensions identifiable by XHOUND, because Droxy handles the rest of the extensions. Out of those 197 extensions, TAJS analyzed 212 total scripts of which 94 were JavaScript files that were designated as a WAR (and thus accessed via a src attribute, see Figure 3) and 118 were inline JavaScript. TAJS successfully completed analysis of 134 scripts (63.2%) finding 19,380 basic blocks and analyzing 18,497 (95.44%). However, TAJS was unable to analyze 78 (36.8%) of the inline JavaScript and WARs because the analysis for 34 scripts timed out, 6 scripts failed with an analysis exception, 6 scripts failed due to syntax errors in the JavaScript, and 32 scripts failed when TAJS crashed.

After manually analyzing the results we found the following reasons for why TAJS failed.

Exceeded timeout threshold. Most of the JavaScript code that caused TAJS to timeout were large JavaScript files that varied in size from 75 kilobytes to over a megabyte. In other cases, TAJS failed to finish analyzing smaller JavaScript code because of a bug in the forced path exploration code.

Analysis exceptions. TAJS failed to complete the analysis because it was missing support for the ECMAScript standard.

Syntax errors. TAJS was unable to analyze scripts with error in the JavaScript syntax.

Crashed. Some of the scripts triggered a bug in TAJS, causing it to crash with null pointer, stack overflow, or other miscellaneous exceptions.

3.5 Cloaked Extension

Once CloakX completes its modifications to the extension, the extension is cloaked and it appears to a webpage using anchorprint or structureprint detection techniques as though the user no longer has that particular extension installed. Architecturally, the resulting extension is similar to Figure 3 with Droxy surrounding the content scripts and translating the extension's DOM requests and droplet injections. To a webpage, the results look similar to the HTML source shown in Figure 4.

The permanence of the cloaked anchorprint and structureprint depends on whether the extension is subject to static rewriting. For cloaked extensions that rely on purely dynamic mutations, the structureprint changes each time the cloaked extension is loaded. Droxy alters the structureprint by injecting new randomly generated noise into the DOM and re-randomizing the cloaked ID and class names. However, CloakX must statically alter extensions with WARs or droplets. As a result, the cloaked fingerprint of extensions requiring static rewriting remains the same until a new version of the extension is reprocessed by CloakX. Although guessing the name of a cloaked WAR is unlikely because CloakX generates a random alphanumeric value that is at least ten characters in length for each WAR; even if an adversary guesses the name of a WAR, the detectability would cease when a new version of the extension was released.

3.6 Deployment

Although we describe CloakX as a client-side mechanism (as this is where the fingerprint rewriting is done), to reduce end-user friction, we envision CloakX as the final step in an extension's release and update process, all of which can be performed by the extension store and would require no intervention by the users. Prior to releasing the extension to users, the store sends the extension to CloakX for preprocessing. During CloakX's preprocessing, CloakX installs Droxy and generates a cloaking-template for the extension. The cloaking-template contains a configuration file that identifies the static variable replacements necessary for WARs, IDs, and class names. When a user requests a preprocessed extension, CloakX uses the cloaking-template to quickly generate and implement random WAR, IDs, and class names for the current user.

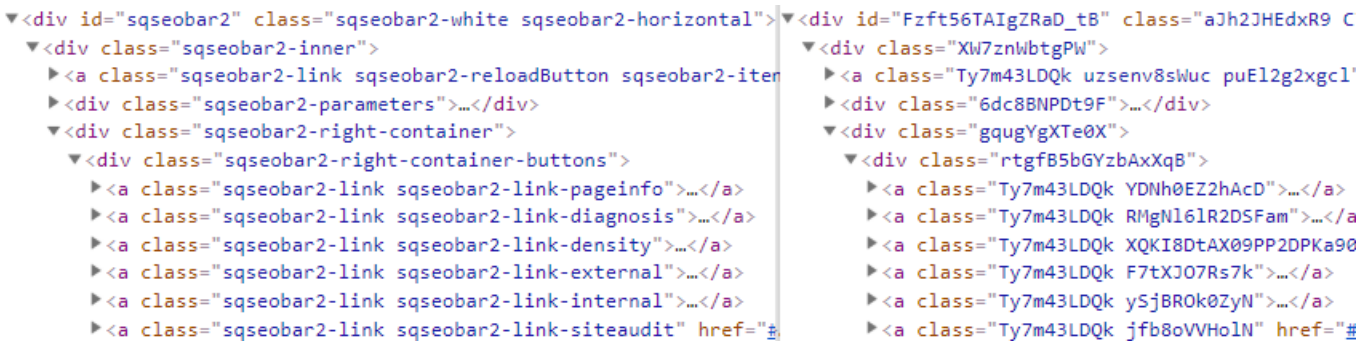

Figure 4: Original code of SEOquake extension (left) and SEOQuake extension when patched by CloakX (right).

4 Evaluation

Altering extensions without modifying the browser or relying on extension developers to make changes is a complex process, and while CloakX is a prototype and does not cover every possible scenario, we wanted to evaluate its current effectiveness. Thus, in this section we evaluate the efficacy of CloakX by (1) testing the breakage introduced by its use (2) the detectability of the cloaked extensions and (3) the performance of the cloaked extensions.

In November 2017, we extracted 59,255 extensions from the Chrome Store. Of those, we identified 13,693 extensions with only WAR fingerprints; however, 67 of the extensions had errors that prevented them from loading. Next, we identified 2,537 extensions having only DOM fingerprints, but Chrome could not load nine of the extensions. The last set of 2,786 extensions had both WAR and DOM fingerprints, one of which would not load in Chrome.

4.1 Functionality Experiments

Testing the functionality of a large set of applications is subject to two problems. First, the tests must explore all the relevant execution paths in the application. Second, the tests should test the entire set of applications. Furthermore, any testing approach will leave code unexplored and applications uncovered, and thus the results form an estimation of functionality breakage. In this work, we perform two different experiments to address both of these challenges: a low-fidelity and a high-fidelity experiment.

The low-fidelity experiment tested the entire population and the high-fidelity experiment randomly sampled from the population. The low-fidelity experiment automatically exercised the original and cloaked extensions and compared the error messages generated by each. The low-fidelity experiment provides a lower bound on the breakage across the entire population. The high-fidelity experiment involved manually — and extensively — exercising the extension, which provided deeper coverage of the extension's functionality. Due to the time-consuming nature of each high-fidelity run, we used a random sample of the extensions from each population.

4.1.1 Low-fidelity Functionality Experiments

To measure functionality breakage introduced by CloakX broadly across all extensions, we performed automated experiments that measured the change in errors from the original extension to the cloaked extension. To execute the experiment, we created a headless browser session using Selenium's ChromeDriver with full logging enabled, which includes errors from the extension's content scripts. Next, we visited a triggering web page, which is similar to the webpage used by XHOUND to activate the extension's functionality. In addition, for those extensions with DOM fingerprints identified by XHOUND, the triggering webpage also included dynamically generated triggers. After the page loaded, the browser waited 30 seconds for any delayed actions to execute. Other than the static and dynamic triggers, the automated experiments do not simulate additional user actions, which might be necessary to execute all the extension's functionality. These steps comprise a *run*, which is completed once for the original extension and once for the cloaked extension.

After both runs finish, we compared the severe JavaScript error messages between the two runs. If the cloaked extension generated the same errors, then the extension passed. Otherwise, if the cloaked extension generated any new or different errors, then the extension failed. Because the automated tests exercise limited functionality and only compare errors, this experiment represents the best case scenario (i.e., the lower bound) on the errors introduced by CloakX. However, the automation allowed us to run the experiment across the entire population.

Table 1 shows the results for WAR and DOM cloaking separately. Note that at the time we ran the experiments, which took place several months after collecting the extensions, some of the original versions stopped working because of Chrome browser updates, obsolete back-end servers, etc. As a result, we only tested working extensions and, therefore, the results only contain errors introduced by CloakX.

In the low-fidelity experiments, CloakX retained equivalent functionality for 99.02% (13,493) of the WAR fingerprintable extensions, 98.69% (2,493) of DOM fingerprintable extensions, and 97.92% (2,727) of WAR and DOM fingerprintable

Table 1: Automated Test Results

Extension set	Total	Tested	Passed	Results	
				Pass	*Fail*
WAR Fingerprintable	13,693	13,626	13,493	99.02%	.98%
DOM Fingerprintable	2,537	2,526	2,493	98.69%	1.31%
WAR & DOM Fingerprintable	2,786	2,785	2,727	97.92%	2.08%
Totals	**19,016**	**18,937**	**18,713**	**98.82%**	1.18%

Table 2: Manual Test Results

Extension set	Random	Top 25	Overall
	Pass/Fail	*Pass/Fail*	*Pass/Fail*
WAR Fingerprintable	25 / 0	25 / 0	50 / 0
DOM Fingerprintable	24 / 1	24 / 1	48 / 2
WAR & DOM Fingerprintable	24 / 1	24 / 2	47 / 3

extensions. For the WAR fingerprintable extensions, we found that the most frequent cause of the failures was the loading of WARs from remote websites. For the DOM fingerprintable extensions, most of the new error messages generated by the cloaked extensions were severe JavaScript errors caused by (1) extensions loading remote content or (2) missing functionality in Droxy. For the WAR and DOM fingerprintable extensions, we found the same errors as seen in the WAR and DOM only tests. To verify the WAR and DOM cloaking did not interfere with one another, we also ran this group using only one of the modifications at a time. The total number of errors was the same for the joint run as it was for the two additional runs with the single modifications, which indicates the modifications did not interfere with one another.

4.1.2 High-fidelity Functionality Experiments

The high-fidelity experiments consisted of manually exercising and evaluating the operation of the cloaked extensions. The high-fidelity evaluation was inspired by the methodology used by Snyder et al. [39]. This methodology focuses on the extension's operation from the perspective of the user. If the cloaking process introduces an error, but the user does not perceive a difference in the extension's operations, then we deem the extension passes. This method of evaluation exercises much more of the extension's code than the automated tests and it provides an additional metric that evaluates the actual operation of each extension. The high-fidelity experiments were performed by the authors using the testing framework detailed next.

We built a custom framework to methodically follow a four-phase evaluation of each extension and advise the tester on the current step in the process. In phase one, the framework loads the original extension and gives the user five minutes to understand its basic operation (including the time necessary to read the extension's description in the Chrome Store). In phase two, the framework reloads the original extension and the user exercises its functionality for five minutes. In phase three, the framework loads the modified extension and the user spends five minutes completing operations similar to the ones completed in phase two to verify it is still operational. In the last phase, the user records any notes on the evaluation and chooses whether the extension passed or failed.

Similar to the automated tests, we divided the extensions into three groups based on the type of fingerprints they emitted. As a result, the populations for each of the high-fidelity tests were as follows: 13,626 WAR fingerprintable extensions, 2,526 DOM fingerprintable extensions, and 2,727 WAR and DOM fingerprintable extensions.

To create samples for these groups, we created both random and systematic samples containing 25 extensions each. We created the first sample by randomly selecting 25 extensions from the population. We formed the systematic sample by selecting the top 25 most popular extensions based on the number of downloads listed on the Chrome Web Store. Throughout the manual tests, if we could not test an extension because the original version was broken or it was only available in a foreign language, then it was discarded and another one was selected according to the associated sampling method. The resulting samples contained quite a bit of diversity between the extensions. Although we found a few instances of overlapping functionality, we kept these extensions in the samples. However, when we found a duplicate extension, we discarded the duplicate and tested a different extension. Some example extensions included in the test samples included a utility for those who are color blind, a search bar tool, a product search by image, a data extraction tool, and a gesture utility for navigation.

Out of all 150 experiments, 145 of the cloaked extensions retained equivalent functionality (see Table 2). All of the WAR fingerprintable extensions retained their functionality. 96% (48 out of 50 extensions) of the DOM fingerprintable extensions and 94% (47 out of 50 extensions) of the WAR and DOM fingerprintable extensions retained their functionality.

After analyzing the broken extensions, we found three different causes for the broken extensions.

Remote source code using original resource name. The extension loads remote Facebook SDK, which looks for obfuscated ID and class values.

Extension relies on hardcoded values that Droxy alters. An extension relies on hardcoded logic that expects its content scripts to appear in a specific order. However, Droxy must be the first content script, which changes the position of all of the extension's original content scripts, and in one case, it broke the extension.

Droxy implementation limitation. Droxy does not currently support recursive iframe sourcing, *cloneNode*, and some advanced CSS rules that the `cssutils` Python library fails to properly parse.

With engineering improvements to Droxy, we can remediate each of the errors listed above and increase the success rate. For the remote source code, Droxy could intercept the remote source code request and parse it before it is executed. This, of course, would add additional performance overhead. The

hardcoded logic could be rectified by overriding the methods that accesses the content scripts. The implementation limitations can be addressed by adding logic to support them into Droxy.

4.2 Detectability Experiments

The detectability experiments evaluated the efficacy of the cloaking against an extension tracking webpage. In the first experiment, the tracker used anchorprints to detect extensions with either WAR or DOM fingerprints. In the second experiment, the tracker used structureprints to detect the extensions with DOM fingerprints. In the third experiment, we investigated the use of behaviorprints to detect cloaked extensions. Last, we explored different methods for detecting the use of CloakX on an extension.

For the first three experiments, we set the fingerprint matching threshold to three. To meet the matching threshold, the tracker must be able to match the extension's fingerprint to three or fewer extensions in its repository. When the tracker meets the matching threshold, it has successfully detected the extension.

We chose a threshold of three because thresholds higher than three showed a sharp decrease in the tracking benefit gained from an extension detection. The matching threshold represents the number of extensions that match a structureprint. The best threshold depends on the requirements of the web tracker and the resources available. The main purpose of the threshold for our experiments was to balance the search time complexity of the fuzzy searches with the increase in the matching of cloaked extensions. For example, by raising the threshold to 20, the web tracker matches three additional cloaked structureprints (one of which matches 18 extensions).

4.2.1 Detectability Experiment Using Anchorprints

The anchorprint detectability experiments focused on detection using WARs, IDs, and class names. In the first phase of the experiment, we harvested the anchorprints of the extensions. Next, we loaded each of the original extensions and used a tracking webpage to verify that the extensions were detectable using the anchorprint. Finally, we loaded each of the cloaked extensions and used a tracking webpage to evaluate the detectability of the cloaked extensions using its anchorprint. For a successful detection, the tracker must meet the matching threshold.

In our experiment, we found that none of the cloaked extensions were detectable using their WARs, IDs, and class names after cloaking. In the first phase, we harvested 17,833 anchorprints, which includes 16,411 extensions with WAR fingerprints and 1,422 that have DOM fingerprints with IDs and classes. However, we chose to limit the testing to the 17,678 extensions that could be executed after being cloaked and assumed that the 155 broken extensions were detectable (thus providing a lower bound on detectability).

In the second phase, we matched 17,534 of the 17,678 original extensions. The ID and class name functionality of the tracker failed to match 144 extensions because it either failed to trigger the extension's anchorprint or it found too many matching extensions. The ID and class name tracker did not find matches for 26 extensions because those extensions required dynamic triggering and the tracker could not use dynamic triggering and still extract the anchorprint; thus, the extensions did not inject their anchorprint into the webpage. The remaining 118 extensions did not count as a detection because the IDs and class names matched more than three other extensions, which exceeded our threshold for a detection.

Initially, the WAR functionality of the tracker failed to find 956 of the WAR fingerprinted extensions using XMLHttpRequest because none of the WAR declarations in the manifest file existed in the extension. However, we discovered we could reliably match these extensions by timing how long it took for three WAR requests to return. The first request is for the declared but missing resources of the extension. The second request was for the extension's manifest.json, which was not declared as a WAR. The third request was for a randomly generated resource that does not exist in the extension and is not a WAR. If the missing request (i.e., the first) takes the longest to return, then the extension has the resource defined as a WAR but the resource does not exist in the extension. Thus, we improved the tracker such that if the tracker failed to match an extension using any of the WARs, then it performs these three requests for each of the WARs in the 956 extensions and if the first request takes the longest it has detected the extension.

In the third phase, we were able to detect 96 of the cloaked extensions using their anchorprints. After investigating several extensions that were detected, we found that matches occurred because CloakX was not translating the ID and classes for the extensions due to errors introduced through the cloaking process. In other words, the experiment found 96 additional cloaked extensions that did not maintain functionality equivalent to their original versions. Thus, with the additional errors but no actual matches, we found that 98.55% (17,582) of the extensions were undetectable using anchorprints.

4.2.2 Detectability Experiment Using Structureprints

The structureprint experiment tested the detectability of cloaked extensions using exact and fuzzy matching to detect the extensions. In the first phase, we ran each of the 5,311 DOM fingerprintable and WAR and DOM fingerprintable extensions through XHOUND to gather the structureprints. In the next phase, we ran each of the 5,223 cloaked extensions through XHOUND to gather cloaked fingerprints. We considered the extensions that failed the automated tests as detectable. Similar to the WAR detection experiments, we

Table 3: Structureprint Detection Test Results

Structureprint Key Type	Exact Matching		Fuzzy Matching
	Original	*Cloaked*	*Cloaked*
Tags, Attributes, Text	3,756 (71.91%)	91 (1.74%)	217 (4.15%)
Tags and Attribute Values	2,092 (40.05%)	91 (1.74%)	95 (1.82%)
Tags	1,420 (27.19%)	91 (1.74%)	91 (1.74%)

did not test the broken extensions, but we assume that they were detectable. In the last phase, we used the structureprints generated in phase one to match the cloaked fingerprints.

The accuracy and precision of detecting structureprints varies depending on both (1) the DOM elements used to build the structureprint and (2) the matching technique used to identify the extension. Therefore, to explore how CloakX can prevent the detection of various types of structureprints, we ran the last phase several times using three different structureprints (each one representing less information used in the structureprint) and two different matching techniques (one on exact matching and one on fuzzy matching) to ensure CloakX reduced detection for each of them.

The structureprints varied based on the contents used to build the fingerprint. The first type used all the XHOUND data, in other words, each fingerprint included added and changed tags, attribute names, attribute values, and text data. While these are the most accurate, they are also the most brittle; as a result, it is likely that the accuracy will degrade considerably in a real-world environment with dynamic HTML content and visitors that have several extensions installed. The second type of structureprint used only the tags and attribute names, which means the fingerprint did not use the attributes values or text. The third type of structureprint used only the tags.

For detection, the experiment extracted an extension's structureprint and then used exact and fuzzy matching against the structureprint database to identify the extension. Exact matching worked well for detecting uncloaked extensions; however, due to the preciseness required for an exact match, cloaked extensions evaded exact matching. Thus, we also tested using fuzzy matching with a 90% level of confidence. Fuzzy matching was successful when the match was made with a 90% level of confidence. Using either matching technique, if the tracker met the matching threshold (three or fewer matches) using the extension's structureprint then we counted the extension as detected.

Overall, we found that cloaking significantly limited the number of extensions detectable using structureprints. With the full structureprints (tags, attribute names, attribute values, and text) and exact matching, we were able to detect 3,756 of the 5,311 original extensions. The reason that 1,555 extensions were undetectable is because the number of matches made using the extension's structureprint exceeded the matching threshold for a detection (a structureprint must match three or fewer extensions for a successful detection). Using the full structureprints on cloaked extensions, none of the cloaked extensions were detected using exact matching and only 126 extensions were detected using fuzzy matching. Using partial structureprints (attributes and tags), we were able to detect 2,092 of the original extensions; however, the cloaked extensions were undetectable using exact matching and only four were detectable using fuzzy matching. Using the tag only structureprints, we detected 1,420 of the original extensions; however, we were unable to detect any of the cloaked extensions using either matching technique.

4.2.3 Detectability Experiment Using Behaviorprints

To understand the limitations of CloakX, we performed an experiment to test the detectability of cloaked extensions using behaviorprints. We chose ten of the most popular extensions with structureprints and to avoid duplication we excluded all ad-blocking extensions except AdBlock. In addition, we examined ten extensions that we randomly selected from those with structureprints. By analyzing their structureprints, we manually created their behaviorprints from portions of the structureprint that remain constant after cloaking.

For the popular extension sample, six of the extensions added elements to the DOM that made them uniquely identifiable. The extensions LastPass, Pinterest Save Button, and Grammarly all add a base64 encoded image to the DOM that makes them uniquely identifiable. The extensions Ghostery, Evernote, and Skype add a `style` tag to the `head` element with features that made them uniquely identifiable. The extension Turn Off the Lights adds a `data-video` attribute. Although the `data-video` attribute is detectable when the extension is cloaked, CloakX randomly includes this attribute even when the extension is not installed, which increases the attacker's false positive rate and makes it more difficult to correctly detect when the extension is truly installed. Even though the cloaked version of AdBlock was detectable, its behaviorprint was not distinguishable from other popular ad-blocking extensions (e.g., AdBlock Plus, uBlock Origin, and AdGuard AdBlocker) because they all perform the same behavior by deleting ads from the DOM and not injecting any other elements into the DOM. Thus, the detection of ad-blocking extensions exceeds the matching threshold for the identification of a user. Ace Script and Honey added `div` tags with an ID, which means CloakX obfuscated the behaviorprint, and the extensions were not detectable.

For the random sample of ten extensions, five extensions were detectable using behaviorprints and five were undetectable. Similar to popular extensions, five of the ten extensions added elements to the DOM that made them uniquely identifiable. For example, two of them added custom text to the web page. Two of the undetectable extensions performed actions on the DOM, which were duplicated by a number of other extensions. Thus, those extensions exceeded the matching threshold and were undetectable. Finally, the three remaining undetectable extensions only added class names, IDs, and common tags to the DOM, which are obfuscated by CloakX.

4.3 Detectability of CloakX

For our last set of experiments, we evaluated three different techniques meant to determine whether an extension was cloaked by CloakX, thus detecting CloakX. These detection experiments were limited to the 2,447 extensions with structureprints that contained at least one ID or class name.

In the first experiment, we created a method for detecting CloakX after analyzing the lengths of the IDs and class names in cloaked and uncloaked extensions. The IDs and class names generated by CloakX were initially six characters in length and comprised of random alphanumeric characters. However, the IDs and class names in uncloaked extensions averaged 15.1 characters for IDs and 15.9 characters for class names. In addition, we discovered that only 62 uncloaked extensions met the criteria of having all their IDs and class names with a length of six (most of which had only one ID or class name). As a result, we created a method for detecting CloakX that marked an extension as CloakX-enabled if all the IDs and class names injected by the extension were six characters in length. Next, we ran the CloakX detector on 2,447 uncloaked extensions and cloaked extensions, thus evaluating 4,894 extensions. The CloakX detector reported 2,509 cloaked extensions, which means we had 62 false positives, 2,447 true positives, and 2,385 true negatives with an accuracy of 98.7%. As a result, we modified CloakX to randomize the length of the IDs and class names it renames. After making this change, we were no longer able to accurately detect the existence of CloakX based on the length of IDs and class names.

As a follow up, the second experiment attempted to identify cloaked extensions by measuring the entropy of the injected ID and class names. Our hypothesis was that the entropy of randomly generated IDs and class names would be measurably different from those chosen by extension developers. We found that the uncloaked ID and class names exhibited nearly the same amount of entropy as their randomized counterparts. As a result, we could not accurately identify the existence of CloakX using the entropy of ID and class names even though the cloaked values contained randomly generated characters.

In the last CloakX detection experiment, we identified the use of CloakX by exploiting popular extensions that both exhibited a behaviorprint and injected an ID or class name into the webpage. In particular, we found Evernote and Grammarly offered a strong behaviorprint and a related ID. Once we identified the existence of the extension's behaviorprint we looked for the ID or class name, if it did not exist then we determined CloakX was likely installed. For instance, Evernote injects a style tag with unique elements and it uses an ID for the same style tag. When a style tag is found that contains Evernote's elements and the style's ID is not style-1-cropbar-clipper, then the tracker records that it found a cloaked version of Evernote. Similarly, when Grammarly's green icon is detected and the top level html tag does not contain a class starting with gr, the tracker records that it found a cloaked version of Grammarly. We tested this by running the tracker against all 2,447 uncloaked extensions and the two cloaked versions of Evernote and Grammarly. The tracker accurately identified the cloaked versions of both extensions with zero false positives.

4.4 Performance Experiments

CloakX minimally impacts the performance of Chrome in our automated tests. We tested CloakX's performance by randomly selecting 500 extensions that contain structureprints because their cloaking requires more resources. Each individual test loaded Chrome, loaded the extension, and ran a triggering webpage from the local machine, which either triggered a page load event or timed out. We executed the tests ten times on both the original and modified extensions. The tests were performed across 16 cores with each core running at 2.2 Ghz. On average, the original extensions took 12.3128 seconds and used 66,790 KB of memory whereas the modified extensions took 12.3221 seconds and used 67,123 KB of memory. Thus, the average increase in overhead for the cloaked extensions was a .07% increase in execution time (0.0093 seconds per extension) and a .49% increase in memory use (333 KB per extension).

5 Discussion

Using the highest failure rate for each of the fingerprint types and using fuzzy matching, CloakX retained the functionality and hid from detection 96.23% (18,222) of the tested extensions. For anchorprint detectable extensions, CloakX rendered 98.55% (17,574) of the extensions undetectable and with equivalent functionality. For structureprint detectable extensions, the tracker was unable to detect 95.91% (5,094) of the cloaked extensions.

CloakX rendered the detection of extensions using anchorprints significantly less accurate. CloakX increases user's anonymity by diversifying WARs, IDs, class names, and custom attributes used for anchorprints. In our experiments, cloaking the WARs, IDs, and class names destroyed the link between the published extension and the currently installed version. As a result, none of the successfully cloaked extensions could be detected based solely on their WAR, ID, or class name. Although it is possible to cloak custom attributes in a similar fashion, CloakX uses cross extension injection of custom attributes to cloak extensions. For trackers using custom attributes to perform anchorprint detection, CloakX increases the number of matches the tracker makes when evaluating an extension's anchorprint, which causes it to exceed the matching threshold and, thus, not detect the extension.

For detection using structureprints, CloakX obfuscated 95.91% (5,094) of the previously detectable extensions even when fuzzy matching with 90% level of confidence was used.

To prevent structureprint matching, CloakX diversifies the tags and attributes added to the DOM by the extension. While these changes were effective against exact and fuzzy matching, the changes only obfuscate the structureprint. Therefore, it is possible that a tracker could create a more sophisticated matching process (as has been the case in fingerprinting attacks and countermeasures) that limits the search to those DOM modifications that are constant.

For example, cloaked extensions are still sometimes identifiable with behaviorprints. In our experiments with twenty extensions, we were able to manually create unique behaviorprints for eleven of the twenty cloaked extensions.

However, behaviorprinting does not currently scale. First, the creation of behaviorprints requires human intelligence and no recent research has shown how to automatically generate a behaviorprint. Second, consistent human intervention is required to prevent the behaviorprints from going stale and no longer being able to identify the extension. For example, LastPass could update their icon, which would no longer match the saved behaviorprint. Third, due to the dynamic nature of the web ecosystem many of the behaviorprints will likely be difficult to use in practice. Lastly, the more popular an extension is the less value the detection of that extension offers towards the goal of identifying users. As a result, for a tracking website to effectively utilize behaviorprints they need to obtain a large number of behaviorprints from both popular and less popular extensions, which exacerbates the scaling problems.

Protection from behaviorprints is a fundamentally difficult problem because the extensions and the browser share the same view of the DOM. CloakX provides some protection from behaviorprints through its injection of noise into the DOM and with additional features could provide even more protection against behaviorprinting. For example, CloakX can make user identification via behaviorprints even more difficult by adding a feature that randomly injects the behaviorprints of the popular extensions, which increases the false positive detections and further dilutes the user's fingerprint. In addition, it is important to point out that only 3,756 extensions (of the 59,255 extensions we used in our study) have unique enough changes to the DOM to form behaviorprints and many of those are not unique enough to provide a robust means of detecting extensions. Nevertheless, the more complete privacy solution for extension fingerprinting is to modify the browser so that the extension and the website JavaScript see their own views of the DOM.

Although we were able to detect the use of CloakX, detecting the presence of a defense mechanism, like CloakX, is different than defending what the mechanism is explicitly trying to protect against (i.e., the presence of specific browser extensions). The fingerprinting value realized by detecting an extension cloaked with CloakX diminishes with each user that uses cloaked extensions. However, it is unlikely any attackers will try to detect cloaked extensions until CloakX becomes popular enough to warrant the attention. As a result, the fingerprint value of detecting CloakX is limited. However, as CloakX becomes more popular it is possible that malicious or shady websites could deny service to users with CloakX-enabled extensions, but this issue exists with any defensive mechanism (similar to what users experience with ad-blocking extension detection).

Thus, despite the limitations described above, CloakX takes a large step forward towards protecting users from wide-spread automated fingerprinting using anchorprints and structureprints.

5.1 Case Study of Failures

Despite their large size and complexity, CloakX cloaks and retains equivalent functionality of 97.88% of the extensions detectable through their anchorprints and structureprints.

Functional breakage caused by the remote loading of scripts was common in broken extensions. One approach to address this issue is to find droplets that load remote scripts which CloakX could download, cloak, and save inside the extension. While an improvement over our current CloakX prototype, the downside of this approach is that the remote scripts might be dynamically generated and copying them inside the extension would not solve the problem. At a high level, we consider the loading of remote code in browser extensions an open problem because remote code can drastically alter the extension's logic *after* that extension has been vetted by the extension store.

Droxy relies on hash values calculated from a static version of the JavaScript code; however, some extensions dynamically change their inline JavaScript code each time it is produced. As a result, Droxy was unable to find the inline code because the current script would not match the one stored in CloakX's metadata. In most cases, we observed that the differences between the original and live scripts were minor which suggests that alternative search routines that allow for fuzzy-matching would be able to handle most of the observed code-matching issues, such as the one used by Soni et al. [40].

5.2 A New Avenue of Security Exploration

Due to the often-misaligned incentives between extension developers and end users, it is desirable to be able to perform late-stage customizations of browser extensions not only to make extensions less fingerprintable, but to also improve their overall security and privacy. In this work, we showed that despite the complexity of the rewriting process, we were able to automatically modify extensions, without requiring browser changes or changes to the development process of browser extensions. Therefore, our approach could be used in additional contexts, such as removing unnecessary third-party trackers and PII leaks [41, 46] or automatically patching vulnerabilities discovered in browser extensions [17].

6 Related Work

To the best of our knowledge this paper proposes the first client-side countermeasure against the fingerprinting of browser extensions. In this section, we briefly describe prior work on generic browser fingerprinting and the related countermeasures.

Eckersley conducted the first large-scale study that showed browser fingerprinting was sufficient to uniquely identify users without cookies or other stateful identifiers [18]. Since then, researchers studied several related topics including tracking the adoption of fingerprinting in the wild [11, 12, 19, 31, 36], proposing new vectors for browser fingerprinting [32, 33, 37, 38, 42, 45], and describing potential defenses against it [29, 30, 35].

Of all the new vectors proposed for browser fingerprinting, in 2017, researchers discovered three different types of side-channels for detecting the presence of specific browser extensions. Sjösten et al. used WARs to determine whether a browser extension is installed [38]. Using this method, they found unique fingerprints for 12,154 extensions and more than 50% of the 1,000 most popular extensions. With the fingerprint, extension detection is straightforward for an attacker to execute (one check per extension) and works even if the user utilizes incognito mode. To defend against this attack, CloakX dynamically renames all of an extension's WARs and rewrites all references to these WARs from the extension's code. As such, every different installation of the same cloaked extension will now have different WARs.

Starov and Nikiforakis utilized the changes in a webpage's DOM to detect extensions. Similar to the WAR detection technique, the attacker pre-processes all the extensions of interest to extract the DOM fingerprints that can be later used to detect the extension's presence [42]. As with WARs, CloakX dynamically rewrites the IDs and class names of all injected DOM elements, which changes the extension's fingerprint and makes it undetectable.

The last browser extension fingerprinting technique, proposed by Iskander-Rola et al. [37] relies on timing channels to detect the presence of files associated with a browser extension. Their method works regardless of whether the extensions declares the files as web accessible. Similarly, Van Goethem and Joosen propose a variation of the same technique using different timing side channels [45]. Because these attacks abuse the access-control mechanisms of a browser, no amount of extension rewriting can counter them. As such, we consider these attacks as out-of-scope for CloakX because our goal is to counteract the detection techniques without modifying the browser.

7 Conclusion

In this paper, we presented the first client-side countermeasure for defending against the detection of browser extensions. Our system, CloakX, uses the principle of diversification so that two installations of the same extension expose different fingerprintable attributes. CloakX operates in an extension-agnostic fashion by rewriting extensions on the client-side, without requiring any modifications to the web browser. Overall, through a combination of large-scale experiments and manual testing, we showed that our CloakX prototype can successfully handle the majority of browser extensions while causing minimal breakage.

Acknowledgements: We thank the anonymous reviewers for their helpful feedback. This work was supported by the Office of Naval Research (ONR) under grant N00014-17-1-2541, as well as by the National Science Foundation (NSF) under grants CNS-1527086, CNS-1617593 and CNS-1703375.

References

[1] Automatically find and apply coupons. https://chrome.google.com/webstore/detail/honey/bmnlcjabgnpnenekpadlanbbkooimhnj.

[2] Chrome.runtime - getbackgroundpage(). https://developer.chrome.com/extensions/runtime#method-getBackgroundPage.

[3] Content scripts. https://developer.chrome.com/extensions/content_scripts.

[4] Detect adblock – most effective way to detect ad blockers. https://www.detectadblock.com/.

[5] Extension overview. https://developer.chrome.com/extensions/overview.

[6] Github - tajs. http://nicolas.golubovic.net/thesis/master.pdf.

[7] Manifest - web accessible resources. https://developer.chrome.com/extensions/manifest/web_accessible_resources.

[8] Manifest version. https://developer.chrome.com/extensions/manifestVersion.

[9] Sizzle javascript selector. https://sizzlejs.com/.

[10] W3 dom overview. https://www.w3.org/TR/DOM-Level-2-Core/introduction.html.

[11] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. 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.

[12] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel. FPDetective: Dusting the Web for fingerprinters. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.

[13] E. Andreasen, A. Feldthaus, S. H. Jensen, C. S. Jensen, P. A. Jonsson, M. Madsen, and A. Møller. Improving tools for javascript programmers. In *Proc. of International Workshop on Scripts to Programs. Beijing, China:[sn]*, pages 67–82, 2012.

[14] E. Andreasen and A. Møller. Determinacy in static analysis for jQuery. *ACM SIGPLAN Notices*, 49(10):17–31, 2014.

[15] E. S. Andreasen, A. Møller, and B. B. Nielsen. Systematic Approaches for Increasing Soundness and Precision of Static Analyzers. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, (June), 2017.

[16] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[17] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*. Citeseer, 2010.

[18] P. Eckersley. How Unique Is Your Browser? In *Proceedings of the 10th Privacy Enhancing Technologies Symposium (PETS)*, pages 1–18, 2010.

[19] S. Englehardt and A. Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1388–1401. ACM, 2016.

[20] Google Chrome Extension. Trump Filter. https://chrome.google.com/webstore/detail/trump-filter/lhondapiaknegjpellpodegmeonigjic.

[21] Google Chrome Extensioon. Hillary Blocker. https://chrome.google.com/webstore/detail/hillary-blocker/kiblhkcoiojbdhhnjaekompfecgelfja.

[22] N. Jagpal, E. Dingle, J.-P. Gravel, P. Mavrommatis, N. Provos, M. A. Rajab, and K. Thomas. Trends and lessons from three years fighting malicious extensions. In *24th USENIX Security Symposium*, 2015.

[23] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 34–44. ACM, 2012.

[24] S. H. Jensen, P. a. Jonsson, and A. Møller. Remedying the Eval That Men Do. *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 34–44, 2012.

[25] S. H. Jensen, M. Madsen, and A. Møller. Modeling the html dom and browser api in static analysis of javascript web applications. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 59–69. ACM, 2011.

[26] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for javascript. In *International Static Analysis Symposium*, pages 238–255. Springer, 2009.

[27] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural analysis with lazy propagation. In *International Static Analysis Symposium*, pages 320–339. Springer, 2010.

[28] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 641–654, San Diego, CA, Aug. 2014. USENIX Association.

[29] P. Laperdrix, B. Baudry, and V. Mishra. Fprandom: Randomizing core browser objects to break advanced device fingerprinting techniques. In *International Symposium on Engineering Secure Software and Systems*, pages 97–114. Springer, 2017.

[30] P. Laperdrix, W. Rudametkin, and B. Baudry. Mitigating browser fingerprint tracking: multi-level reconfiguration and diversification. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 98–108. IEEE Press, 2015.

[31] P. Laperdrix, W. Rudametkin, and B. 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, May 2016.

[32] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham. Fingerprinting information in javascript implementations. In *Proceedings of W2SP*, volume 2, 2011.

[33] K. Mowery and H. Shacham. Pixel perfect: Fingerprinting canvas in html5. *Proceedings of W2SP*, pages 1–12, 2012.

[34] Nicolas Golubovic. Attacking Browser Extensions, MS Thesis, Ruhr-University Bochum. http://nicolas.golubovic.net/thesis/master.pdf, 2016.

[35] N. Nikiforakis, W. Joosen, and B. Livshits. Privaricator: Deceiving fingerprinters with little white lies. In *Proceedings of the 24th International Conference on World Wide Web*, pages 820–830. International World Wide Web Conferences Steering Committee, 2015.

[36] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Security and privacy (SP), 2013 IEEE symposium on*. IEEE, 2013.

[37] I. Sanchez-Rola, I. Santos, and D. Balzarotti. Extension breakdown: Security analysis of browsers extension resources control policies. In *26th USENIX Security Symposium*, 2017.

[38] A. Sjösten, S. Van Acker, and A. Sabelfeld. Discovering browser extensions via web accessible resources. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 329–336. ACM, 2017.

[39] P. Snyder, C. Taylor, and C. Kanich. Most websites don't need to vibrate: A cost-benefit approach to improving browser security. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 179–194. ACM, 2017.

[40] P. Soni, E. Budianto, and P. Saxena. The sicilian defense: Signature-based whitelisting of web javascript. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1542–1557. ACM, 2015.

[41] O. Starov and N. 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.

[42] O. Starov and N. Nikiforakis. XHOUND: Quantifying the fingerprintability of browser extensions. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 941–956. IEEE, 2017.

[43] M. Stockley. The web attacks that refuse to die. https://nakedsecurity.sophos.com/2016/06/15/the-web-attacks-that-refuse-to-die/.

[44] K. Thomas, E. Bursztein, C. Grier, G. Ho, N. Jagpal, A. Kapravelos, D. McCoy, A. Nappa, V. Paxson, P. Pearce, et al. Ad injection at scale: Assessing deceptive advertisement modifications. In *IEEE Symposium on Security and Privacy (SP)*, 2015.

[45] T. Van Goethem and W. Joosen. One side-channel to bring them all and in the darkness bind them: Associating isolated browsing sessions.

[46] M. Weissbacher, E. Mariconti, G. Suarez-Tangil, G. Stringhini, W. Robertson, and E. Kirda. Ex-ray: Detection of history-leaking browser extensions. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 590–602. ACM, 2017.

[47] X. Xing, W. Meng, B. Lee, U. Weinsberg, A. Sheth, R. Perdisci, and W. Lee. Understanding malvertising through ad-injecting browser extensions. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 1286–1295, 2015.