Peeking through the `window`: Fingerprinting Browser Extensions through Page-Visible Execution Traces and Interactions

Shubham Agarwal
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
shubham.agarwal@cispa.de

Aurore Fass
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
fass@cispa.de

Ben Stock
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
stock@cispa.de

ABSTRACT

Browser extensions are third-party add-ons that provide myriads of features to their users while browsing on the Web. Extensions often interact with the websites a user visits and perform various operations such as DOM-based manipulation, script injections, and so on. However, this also enables nefarious websites to track their visitors by fingerprinting extensions. Researchers in the past have shown that extensions are susceptible to fingerprinting based on the resources they include, the styles they deploy, or the DOM-based modifications they perform. Fortunately, the current extension ecosystem contains safeguards against many such known issues through appropriate defense mechanisms.

We present the first study to investigate the fingerprinting characteristics of extension-injected code in pages' JavaScript namespace and through other observable side-effects like changed cookies. Doing so, we find that many extensions inject JavaScript that pollutes the applications' global namespace by registering variables. It also enables the attacker application to monitor the execution of the injected code by overwriting the JavaScript APIs and capturing execution traces through the *stacktrace*, the set of APIs invoked, etc. Further, extensions also store data on the client side and perform event-driven functionalities that aid in attribution. Through our tests, we find 2,747 Chrome and 572 Firefox extensions to be susceptible to fingerprinting. Unfortunately, none of the existing defense mechanisms prevent extensions from being fingerprinted through our proposed vectors. Therefore, we also suggest potential measures for developers and browser vendors to safeguard the extension ecosystem against such fingerprinting attempts.

CCS CONCEPTS

• **Security and privacy → Web application security**.

KEYWORDS

Client-side Security, Extension Fingerprinting, Browser Extensions

ACM Reference Format:
Shubham Agarwal, Aurore Fass, and Ben Stock. 2024. Peeking through the `window`: Fingerprinting Browser Extensions through Page-Visible Execution Traces and Interactions. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3670339

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3670339

1 INTRODUCTION

With the rising trend of cookie-less tracking, online trackers are in an arms race with Web Privacy advocates. They continuously compete with various anti-tracking and fingerprinting measures to uniquely identify users on the Web. Recent studies on browser fingerprinting techniques have shown many cookie-less vectors, such as Canvas APIs [4], WebGL APIs [5], and other side channels [1, 27, 30, 42, 50, 57]. These techniques allow a malicious website to effectively harvest client-side information specific to individual users on the Web and further track their activity across websites.

Browser extensions have emerged as one of the most interesting fingerprinting vectors for these trackers in recent times, owing to their unique position in the overall Web ecosystem. Extensions can perform privileged operations implemented in their background script or even execute code in the Web applications' execution context through the content scripts. Tailored to provide a specific set of features to their respective users, browser extensions may also inadvertently reveal personal information about their users, such as their geolocation, background, ethnicity, or social and personal interests [24]. These extensions perform a specific and often highly privileged set of operations, such as DOM-based modifications, changes to cookies, or script injections, to implement their intended functionality at runtime. These operations, however, could also expose them to being uniquely identified by online trackers.

Prior work has shown different ways of uniquely identifying browser extensions. Some of these include Web-accessible resources-based (WAR-based) fingerprints [47], the side effect of code bloating [52], behavior-based fingerprints [24, 49, 53], user-induced side effects [48], and stylesheet-based fingerprints [28]. On the other hand, several mitigation techniques also intend to thwart any fingerprinting attempts based on the above techniques [25, 46, 59]. The studies have led to a cat-and-mouse game of newly introduced fingerprinting vectors and subsequent defense mechanisms. A recent example is the optional usage of randomized runtime identifiers for extensions [39] to thwart WAR-based fingerprinting.

However, ultimately, every extension has some intended functionality, much of which is to interact with the loaded page. This also means that the extension needs to interact with the page through various APIs from the content script. However, when not done carefully, this can leave traces of the extension's execution in the JavaScript namespace of the document, i.e., within the reach

of a fingerprinting attacker. These range from client-side storage (e.g., *localStorage* or *cookies*) through the invocation of global APIs and properties (e.g., `Array.forEach`), and setting global variables to observable events caused by extension-sent postMessages. Notably, none of the existing defenses (i.e., Parallel DOM [25], Shadow DOM [28], or randomized extension URLs [46]) protect against these vectors due to shared *window* context.

In this study, we first discuss how an extension is susceptible to fingerprinting based on its execution trace and other JavaScript-observable side-effects (e.g., changed cookies or sent postMessages). We then build a fingerprinting page that sets up the JavaScript environment such that it can capture an extension's interaction with it. Doing so repeatedly allows us to detect which observed functionalities are consistently present to *deterministically* infer the presence of an extension. We run our analysis on a set of up-to-date Chrome extensions, showing that 2,747 extensions can be fingerprinted through our identified vectors, affecting over 169M users who installed these fingerprintable extensions. Over 59% of the reported extensions adhere to the *ManifestV3* standards, highlighting the fact that the issues are a threat to modern extensions. Moreover, our results transfer to the Firefox ecosystem, where we find 572 extensions that can be detected. Notably, by comparing with the labeled dataset from *Carnus* [24], we are not only able to detect 1,355 extensions, but importantly would still be able to detect 484 extensions their approaches would be unable to detect if dynamic runtime URLs were deployed. Our findings highlight that the discovered issues not only affect both major extension ecosystems but also add significant fingerprinting surface, which existing (and proposed) approaches could not readily defend against.

We summarize the key contributions of this study here:

- We identify and leverage two classes of fingerprinting vectors: execution traces and JavaScript-observable side effects that an attacker can abuse to detect browser extensions installed by users on the Web.
- We then build a dynamic analysis pipeline, *Raider*, to analyze all free extensions in the Chrome Web Store and identify those that are fingerprintable through our proposed vectors. We show that 2,747 extensions are uniquely identifiable.
- By applying our techniques to the *Carnus* dataset, we showcase that our techniques can overcome randomized WAR URLs. Further, our findings for Firefox highlight that the underlying issues exist in both major extension ecosystems.
- To facilitate future research, we will open source our analysis pipeline and the associated dataset [45].

2 TECHNICAL BACKGROUND

In this section, we provide an overview on the extensions' architecture and explain the key concepts relevant for our study: the (shared) global namespace in JavaScript, available client-side storage mechanisms such as cookies, and event-driven communication through postMessages.

2.1 Browser Extensions

Browser extensions are client-side add-ons, typically designed by third-party developers to provide additional features to Web users. Extensions have access to the powerful *Chrome APIs*, exposed by

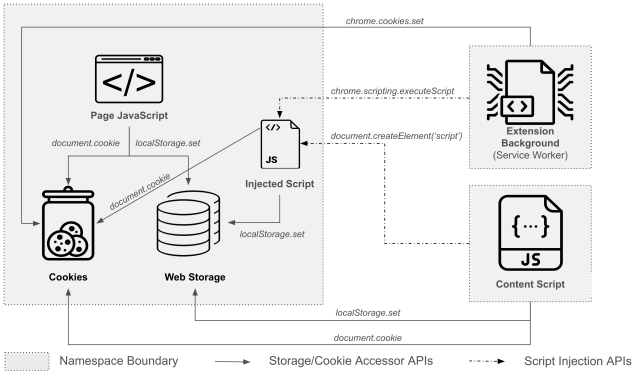

Figure 1: The access-control, capabilities, and isolation boundaries of different components in browser extensions

browser vendors, through which they can perform various operations on behalf of their users. For example, extensions can offer bookmark management, tab customization, text assistance, password management, or ad-blocking functionalities to their users across different browsing platforms.

Figure 1 shows different extension components and their isolation boundaries. An extension includes a mandatory *manifest.json* file, describing an extension's metadata, e.g., the API & host permissions it holds, as well as the scripts and other Web resources required by an extension for seamless execution. The privileged component, i.e., the service worker (background script, in the previous *ManifestV2* standards), has access to powerful APIs such as `scripting` or `cookies`, allowing extensions to inject scripts or get access to the cookies of the visited page, respectively. It runs in an isolated JavaScript namespace, can communicate through messages, and has the unidirectional ability to inject scripts into a page. The content script is less privileged since the JavaScript executes in the context of the visited Web page, although in a separate JavaScript namespace from that of the page. It gets a clean reference to the page DOM and can perform read and write operations through the APIs available in its own JavaScript namespace [34]. This is implemented to ensure that a malicious page cannot abuse the higher level of privileges of the content or even background scripts. Notably, while the actual APIs are in a separate namespace, the content script shares client-side stores such as cookies or localStorage with the page. Additionally, extensions can inject scripts into the page itself, either by programmatically creating and adding them to the DOM from the content script, or by calling `executeScript` from the background [14]. Importantly, the injected JavaScript then executes with the same privileges and in the same context and JavaScript namespace as the Web page.

An extension requires corresponding permissions to carry out privileged operations in its background. For example, it requires the `bookmarks` permission to create, search, update, or remove bookmarks from the browser of their users [11]. This, however, is not the case with content scripts that have access to all the Web APIs also available to a Web page by default. For instance, an extension can invoke any `IndexedDB` APIs without requiring any permissions.

```
1  window.foo = "bar";
2  // Overriding the default accessor of the defined variable.
3  Object.defineProperty(window, 'foo', {
4      get: function() { return "baz"; },
5      configurable: false
6  });
7  console.log(window.foo);
8  //Expected output: "bar", Actual output: "baz"
```

Listing 1: The dynamic nature of JavaScript allows to customize the default behavior of built-in APIs & properties.

Depending upon the nature of the API permission, both these components may still require appropriate host permission and restrict their capabilities to these hosts [15]. The `permissions` (or `host_permissions` in the *ManifestV3* standards in Chrome) key contains information on hosts an extension's background can operate on. The `content_scripts` key in the *manifest* contains script paths with corresponding host permissions. The `web_accessible_resources` key includes the definition of other auxiliary resources (e.g., CSS) required for an extension's functionality.

2.2 The Global Namespace in JavaScript

Whenever the browser renders a page, all scripts that operate on the page share the same global object called `window`. This means that both first- and third-party scripts can read each others' global variables, access global functions defined by each of them, and so on. This also allows modifications to the execution environment by any script since JavaScript's built-in functions (e.g., the `document` used to interact with the DOM or the `Array` constructor) are also merely global (automatically initialized) variables. This dynamic nature of JavaScript allows Web developers to overwrite the native definition of nearly all the built-in APIs and properties it offers to the Web page. For example, as shown in Listing 1 (lines 3–6), a developer can override the native accessors of the properties defined on the `window` object to alter its behavior (line 8). The overwritten behavior of APIs affects all the JavaScript that executes in the same global namespace. Notably, while the example shows how to overwrite a getter of a specific property, a developer can easily overwrite existing functions as well. Recall that the content script runs in an isolated namespace, whereas the scripts injected into the page by the extension (either from content or background scripts) share the page's namespace.

2.3 Client-Side Storage Mechanisms

We now briefly describe each client-side storage API accessible by both the Web applications and the extensions.

Cookies: Cookies used to be the traditional way of storing data on the client side, associated with respective hosts, before the `localStorage` and `sessionStorage` APIs were introduced. However, modern Web applications still use cookies to store information on users' machines. Specifically, a browser sends cookies with all the outgoing requests to a particular host. Extensions can set, delete, or modify cookies on a visited Web page through scripts executing in the context of a Web application (i.e., content scripts and WARs) using `document.cookie`, similar to native applications. Additionally, extensions can also set or get cookies in their background through the `cookies` permission and its corresponding APIs.

Web Storage API: The *Web Storage* API was introduced with the HTML5 standards and enabled applications to store comparatively large chunks of structured data, as *{key: value}* pairs, on the client side [32]. This API enforces origin-level isolation on the data storage and access. One can store data in two different contexts: 1) temporary data stored only for the current page session, using the `sessionStorage` API, and 2) data stored to persist across sessions, using the `localStorage` API. While `sessionStorage` only allows up to 5 MB of data storage per origin, `localStorage` allows comparatively more data storage. Websites utilize *Web Storage* APIs to store data such as user state, runtime configuration, personalization settings, or code-caching [54]. Unlike cookies, the data stored with these APIs remains on the client, and the browser never implicitly sends them to an application server. Extensions operating on a given Web origin can also access the *Web Storage* APIs, and store data keyed to this origin through their content or injected scripts [33].

IndexedDB API: `IndexedDB` is a JavaScript-based object-oriented database that allows client-side components to store extensive structured data that persists across sessions [31]. This API also enforces origin-level access control on the data storage and accesses, similar to the *Web Storage* API. This API allows operations based on individual transactions and executes asynchronously, i.e., without blocking other executable code in the event loop. Web applications may store data using `IndexedDB` for various purposes, such as caching of code, network responses, or other static resources, shared with the *Service Workers* API [18].

2.4 *postMessages* & Other Runtime Events

Extension components can communicate with each other (i.e., content scripts, web-accessible resources) or with a Web page through the *postMessage* API. Here, the sender, either an extension script or a Web page, sends the message data by also optionally specifying the message target [17]. The other party listens to the message by registering a corresponding *message handler*. It is pertinent to note that since any extension-initiated postMessages execute in the context of the application, the effective origin of these messages is the Web page's origin. Thus, the Web page can also register a corresponding event listener and listen to all the message exchanges. Similarly, a content script may also dispatch custom events to a page. While it is infeasible to a priori know which events may be fired, an extension's injected script still needs to register a listener for said events, which can be observed due to the necessary invocation of the `addEventListener` function in the global scope.

3 THREAT MODEL

We consider an attacker who is capable of having a victim visit their website but who cannot control a specific website. That is if an extension only operates on *Facebook*, this is outside of our threat model, as the adversary cannot gain control over that site. More concretely, any traditional Web attacker may try to detect the existence of one or more target extensions installed on the client side. The attacker can use the information associated with these extensions to infer privacy-sensitive characteristics of their visitors, such as their geo-location, ethnicity, or religion [25]. The attacker

can further share this information with third-party trackers or even use it themselves, all without the users' knowledge or consent.

Browser extensions can interact with and store data on the client side using storage APIs (e.g., `localStorage`) via content scripts and other web-accessible resources, as also shown in Figure 2a (line 2). In this case, an attacker application can detect any target extension(s) installed on the client side by probing for items within these data stores, accessible through the application JavaScript (Figure 2a, lines 4–8). Unfortunately, none of the existing defenses against fingerprinting prevent these observable side effects.

Next, as shown in Figure 2b, extension components may communicate with Web pages or even among themselves (i.e., popups or WARs) via postMessages (line 2). They could also register other event listeners to execute event-driven operations (e.g., *scrollup*, *onmouseover*, other custom events) either through content scripts or the injected code (lines 3–6). Here (as shown in lines 7–10), the attacker JavaScript can also listen to all the postMessage exchanges issued by the extension scripts since these scripts are injected and executed in the same context. Similarly, the page JavaScript can forcefully intercept or trigger runtime events registered by the extensions to induce observable side-effects, as they share the same execution context. In this case, the attacker must know the target events a priori to trigger them and cause any observable side effects.

As discussed in Section 2, extensions can inject JavaScript into the visited page, either from the content script or through the background. In this case, the injected JavaScript executes in the same context as the Web page, similar to the content script. However, in addition, the injected code also shares the global JavaScript namespace with the Web page, such that its execution may cause side effects to this namespace. For instance, as in Figure 2c (line 2), the injected script may register event listeners, set variables in the global scope, and access/modify native functions and properties directly accessible to the Web page JavaScript. In this case, the attacker can monitor the usage of these global APIs to detect the execution traces of extension-injected code. For example, as in lines 4–6, the attacker JavaScript can iterate over all global properties on the `window` to detect extension-defined ones. Thus, extensions that inject JavaScript into arbitrary Web pages are susceptible to fingerprinting through the execution traces of the injected code.

We assume our attacker to be sufficiently able to download all browser extensions from the extension store [10] and run offline analysis on them to observe their behavior and derive identifiable signatures. The attacker could then use these signatures to detect installed extensions and, subsequently, their users online. Notably, the attacker only needs to run the offline analysis step for new extensions on the store or when an existing one is updated. Here, we only consider extensions that operate on any Web pages for our analysis, such that an arbitrary attacker application can fingerprint them. However, it is pertinent to note that extensions that run only on specific pages may still be identifiable by the corresponding applications through any of the vectors proposed in our study (meaning that the results presented in this paper are a *lower bound* of the extensions we can fingerprint with our approach).

```
1 // content_scripts.js
2 localStorage.setItem('foo', 'bar');
3 // attacker-webpage.js
4 for (let index = 0; index < localStorage.length; index++) {
5   let key = localStorage.key(index);
6   let value = localStorage.getItem(key);
7   logStorage(key, value);
8 }
```

(a) *Storage* APIs scanning

```
1 // content_script.js
2 window.postMessage('Hello from CS!', '*');
3 // popup.js
4 window.addEventListener('message', function (event) {
5   event.source.postMessage('Message received!');
6 });
7 // attacker-webpage.js
8 window.addEventListener('message', function (event) {
9   logMessages(event.data);
10 });
```

(b) Intercepting postMessages

```
1 // injected-script.js
2 extension_key = "extension_value";
3 // attacker-webpage.js
4 for (let prop of Object.getOwnPropertyNames(window)) {
5   logProperty(prop, window[prop]);
6 }
```

(c) Global variables set by extensions

Figure 2: Observable Behavior of Extensions at Runtime

4 RESEARCH METHODOLOGY

Our overarching research question is: *how many extensions can be uniquely fingerprinted through the traces they leave in the global scope of a visited page or through their otherwise visible side effects?* To answer this, we first need to identify extensions that include scripts or have permissions that may allow them to store client-side data, send postMessages, or inject scripts into the page. We do so by statically analyzing the manifest files and filtering out extensions without the necessary permissions. Subsequently, for the remaining extensions, we need to learn which of them use the capabilities at runtime. For this, we spawn browsers to load each extension and visit our test page, allowing us to capture the extension's interactions with it. To then answer our main question, we determine traces (observable side-effects) unique to each extension.

4.1 *Raider*: Overview

We build an automated dynamic analysis framework, *Raider*, to answer the above question and detect extensions based on *a.)* the execution traces on the global JavaScript namespace from extension-injected scripts; and *b.)* their side effects through interactions with client-side Storage APIs or sent postMessages. Figure 3 depicts the high-level overview of our methodology. We start by *i.)* unzipping the extensions and statically analyzing their manifest to identify scripts that operate on *<all_urls>*. *ii.)* We then extract all valid content scripts and web-accessible resources as JavaScript that these extensions declare in their manifest and determine host permissions for individual scripts. *iii.)* We also check for extensions' capabilities

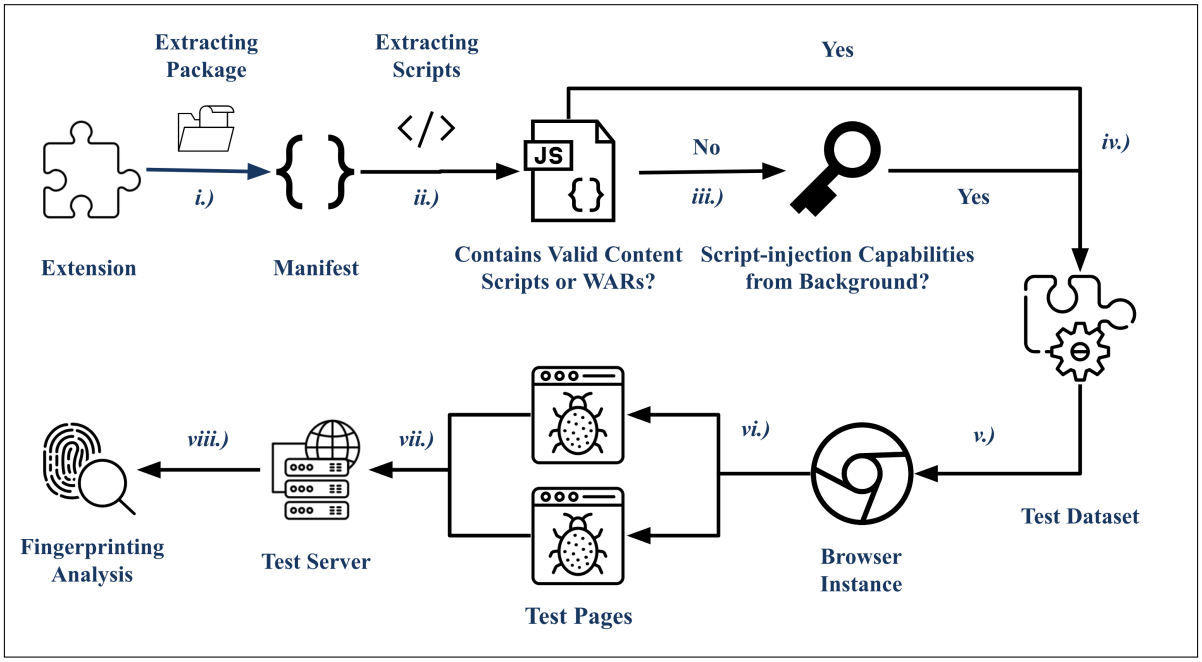

Figure 3: Overview of methodology: *i)* Extract package and parse manifest. *ii)* Check for valid content scripts and WARs that operate on all URLs. *iii)* Check for script-injection capabilities from the background, *iv)* Select those extensions that have script-injection permissions from either (ii) or (iii). *v)* Spawn browser instance and load extension. *vi)* Navigate to the test pages. *vii)* Collect signatures and store them. *viii)* Analyze the uniqueness of signatures to confirm fingerprintability.

to inject JavaScript from the extension background. *iv.)* Lastly, we select those extensions for our next stage where an extension either *a.)* has at least one content script or WAR operating on any URL (i.e., `<all_urls>`); or *b.)* has permissions to inject JavaScript into arbitrary hosts from its background.

With the selected set of extensions, the pipeline then analyzes their runtime behavior to collect fingerprinting signatures. The dynamic step (*v.)* – *viii.)*) involves two different data collection strategies: we collect the execution traces of the extension-injected code on the global JavaScript namespace differently from the way we capture the extension-driven interactions with the *Storage* and *postMessage* APIs. We do this by loading extensions individually and capturing any side effects they cause on the global namespace as signatures through our specially crafted test page. Then, we collect all the data set by individual extensions in any of the client-side data stores at runtime by individually loading them in the browser instance and polling these data stores periodically through another test page. After collecting all data, we determine whether these extension-driven interactions are *distinct* and appear *consistently*. To that end, we visit the test page nine times and only consider behavior to be relevant if it is observed in all nine visits and is unique to one extension.

4.2 Static Pre-filtering

The first stage of our proposed methodology consists of a static extension pre-filtering step. Here, we identify those extensions that can inject JavaScript into the DOM, interact with storage APIs, or send postMessages, and thus, require further scrutiny to determine if they are fingerprintable, based on our threat model. More specifically, our tool parses the manifest of individual extensions to detect any `content_scripts` or `web_accessible_resources` declarations to extract relevant JavaScript files. This is because an extension can only interact with the storage APIs or send postMessage from these scripts. Moreover, content scripts also enable the injection of other scripts into the page. Extensions may also choose to execute JavaScript in the page's context from their background/service worker at runtime without declaring them in their manifest, using the `scripting` or `tabs` API [14]. This permission also allows extensions to inject or update content scripts from their background at runtime. In the *Manifest V2* standards, this translates to the `tabs` permission [14]. Additionally, extensions may also set cookies on arbitrary domains from their background/core through the `chrome.cookies` API or even inject the `Set-Cookie` header within HTTP response headers, by requesting the `webRequest` or `declarativeNetRequest` permissions. While extensions can interact with the client-side storage, register event listeners, or cause side effects to the global namespace through different scripts, they should also operate on all URLs for any attacker application to fingerprint them. Thus, we only consider those extensions with sufficient host permissions that allow them to run on arbitrary hosts, i.e., `<all_urls>` and equivalent. We extract the host permissions specified for the content scripts, web-accessible resources, and background scripts, remove any wildcards, and normalize them to detect their actual operational set of hosts. This is necessary as developers may specify match patterns instead of fully-qualified

domain names [8], (e.g. *://*/*, or http://*/foo*) which may still effectively allow extensions to operate on arbitrary Web pages. Note that we intentionally skip extensions that only have the `activeTab` permission or `optional_host_permissions` (for MV3) in their manifest. While this gives an extension the same capabilities as an explicit host permission, it requires the user to actively engage with the extension. This is outside of our threat model, which does not require user intervention outside of the page. Appendix A shows an example of extensions with valid script declarations in the manifest relevant to our study. As shown, *content_script.js* will execute on all HTTPS URLs, while *storage.js* and *cookies.js* operate on all URLs, irrespective of the URL scheme. Thus, any website running on HTTPS can be a potential adversary in this case, as per our threat model described in Section 3. We note that applications may also directly send messages to the extension core using the `chrome.runtime` API [13]. However, this is only possible when the extension intentionally opts-in to be reachable from a given website by specifying the respective domains as `externally_connectable` in their manifest [7]. Hence, we discard such extensions here.

To sum up, we select extensions for our analysis, that: *i.)* have at least one valid content script or web-accessible resource running on <*all_urls*>, or *ii.)* have valid background script running on <*all_urls*>, and request for any of the following permissions: `scripting`, `tabs`, `cookies`, `webRequest`, or `declarativeNetRequest`.

4.3 Execution Traces of Injected Code

An extension's content script can interact with the given page through the `window` and the `document` handle. These two objects, however, are not shared with the page itself. While any changes through `document` apply to the page's DOM, changes to the content script's `window` object are opaque to the visited page. This way, the content script of an extension cannot, intentionally or accidentally, alter the behavior of the JavaScript APIs, properties, and variables declared by the Web page JavaScript, and vice versa. In order to run extension-specified code within the context of the page's `window` object, this code must be explicitly injected into the page. Extensions can perform script injections either directly through their content script – by using the `document.appendChild` API, or through their background script – by using the `scripting` (or the `tabs`) permission (`chrome.scripting.executeScript`).

Now, the extension-injected code could perform a wide array of operations in the context and the namespace of the visited application. More importantly, the JavaScript APIs the injected code may utilize or the `window` properties that the injected code may read or write during its execution are also shared and observable to the Web page through its JavaScript. This behavior enables an attacker Web application to observe, or even further, to overwrite the native behavior of nearly all the JavaScript APIs and properties, in that particular JavaScript namespace to monitor their usage. For example, an attacker JavaScript can overwrite the native definition of the `Array.prototype.forEach` API to actively observe any invocations of this API. If an extension-injected code now invokes the `forEach` operation on *any* array, the attacker will then be able to observe its invocation. This capability of the Web page's JavaScript code can be leveraged to detect an extension's behavior in various ways, which we discuss in the following.

```
1  function __hook(object, property, api) {
2    // Preserving native definition of the function.
3    let __originalFunc = object[property];
4    // Custom definition for Global APIs
5    function __customFunc() {
6      // Extracting API related information.
7      let context = this;
8      let args = Array(...arguments);
9      // Extracting the source code of the executing code.
10     let callerData = {};
11     let caller = arguments?.callee?.caller;
12     while (caller) {
13       callerName = caller.name;
14       callerFunc = caller.toString();
15       callerData[callerName] = callerFunc;
16       caller = caller?.arguments?.callee?.caller;
17     }
18     // Capturing the stack trace of the executing code.
19     let stacktrace = new Error().stack;
20     // Sending data to our test server.
21     logToServer({ api, context, args, stacktrace, callerData });
22     // Now, returning the result from executing native function.
23     return __originalFunc.apply(this, arguments);
24   }
25   // Replacing the native definition with custom definition.
26   object[property] = __customFunc;
27 }
28
29 //Instrumenting APIs now...
30 __hook(Array.prototype, "forEach", "Array.forEach");
```

Listing 2: Logic to overwrite globally-accessible APIs.

4.3.1 Overwriting global APIs. We begin by discussing the steps to overwrite the JavaScript APIs and the global property accessors, enabling us to capture their invocations or accesses, respectively. At the same time, we also intend to preserve the original behavior of the overwritten components to avoid any side effects of our instrumentation at runtime. For our purposes, this is particularly important to ensure an extension can fully execute all of its functionality, which provides ample chance to fingerprint it.

We demonstrate the steps to perform API overwrites through Listing 2. Here, the `__hook` method instruments the individual APIs to enable the logging mechanism. Concretely, it first stores the original definition of the API under instrumentation (as `__originalFunc` in line 3). Then, it defines a custom method (here, `__customFunc` as in lines 5–24) where we define the logic to capture all the relevant invocation-associated details. It also includes the logic to dispatch the collected data to our test server (line 21). Once the logging is complete, this `__customFunc` returns the result from the native definition of the called API through `__originalFunc` (line 23). Thus, our custom definition does not affect the natural execution flow of the injected code. In the end, we overwrite the native definition of the API with our custom-defined logic (line 26). This way, we instrument a total of 571 different JavaScript APIs, accessible to both the Web page and the extension-injected code in the global JavaScript namespace (similar to line 30) [38]. We follow similar steps to overwrite the native definition of JavaScript property accessors, such as `document.title` and `window.name`. Here, we use the `__lookupGetter__` and `__lookupSetter__` APIs to obtain the native definition of the property accessors. We then overwrite them using the `Object.defineProperty` API. This way, we overwrite 51 other globally accessible JavaScript properties. We selected the global JavaScript APIs and properties that are standard

built-in JavaScript objects [38].[1] Now, we elaborate on the relevant invocation-associated details from these instrumented APIs and the properties we extract.

4.3.2 Relevant contextual data from API invocations. Our instrumentation allows us to capture different sets of data based on the nature of the invoked API. For instance, as shown in the first example of Appendix A, when the `Array.forEach` API is invoked on `foo`, the value of `this` in the current execution context is the array passed to the API (i.e., [1, 2, 3]). The argument is the callback that processes the iterated element. Here, suppose an extension-injected code invokes this API. In that case, our logger will capture three critical pieces of information: the API name, the arguments passed, and the value of `this` in the execution context (Listing 2 – lines 7 and 8). There are other APIs that are static methods of their parent class and do not have their own context or `this` value (e.g., `Array.isArray` in Appendix A). Here, we only capture the name of the invoked API and its arguments.

4.3.3 Obtaining the source code of the injected code. Whenever an executing script invokes a function, *fn*, the invoked function *fn* also contains the pointer to its *caller* [36]. That is, the *invoked* function has the pointer to the *invoking* function through the `arguments.callee.caller` property. This is also true for the JavaScript APIs and the property accessors in the global namespace that we consider in this study. Through this, an attacker Web page could extract unique caller functions or even leak the entire source code of the script and use them as vectors to detect extensions later. Concretely, suppose an extension-injected code invokes any instrumented APIs or accesses any property we instrument. In that case, our custom method also recursively extracts the *caller* of the invoked API until it is set to `null`. Lines 9–17 in Listing 2 show our approach to collect caller-associated details for an invoked API. Notably, if the *caller* is *a.)* top-level code; *b.)* an *arrow*, *async*, or *generator* function; or, *c.)* runs in the *strict* mode, it is always set to `null`, and our logger cannot capture anything. Note that reading the source code of injected scripts could also be done through a MutationObserver; this, however, can be easily defeated through a ShadowDOM [28], which is why we do not consider this vector.

4.3.4 Capturing the stacktrace. The `arguments.callee.caller` does not always return the handle to its caller, primarily in cases where the entire injected code is running in the *strict* mode, or the API invocation occurs on the top-level code. However, the attacker can still capture the execution stack of the injected code up to the point where the API invocation occurs. This stacktrace not only provides the names of the functions called in reverse order but also contains both the URL of the file (if the code is in an external script) and the line number and offsets. We collect this information by accessing the *stack* property of the `Error` object (as shown in line 19 of Listing 2). In cases of randomly generated runtime identifiers (both in Chrome and Firefox), the filename for scripts included as web-accessible resources contains a random identifier. Therefore, we consider two attacks: the full stacktrace (including runtime identifiers) and a normalized stacktrace (for a hypothetical case of widely adopted randomized runtime identifiers) for which we remove the extension IDs from the trace; leaving us with the filename, line number, and offset. As we discuss in Section 5, both are distinctly unique features across a vast amount of extensions.

[1]Please visit our repository for the complete list of hooked APIs and properties [45].

```
1  //Client-side storage state as polled on every 0.5 second for 10
   ↪ times after page load.
2  window.addEventListener('load', async function (e) {
3      let counter = 0;
4      setTimeout(async function run() {
5          if (counter++ < 10) {
6              await pollStorage();
7              setTimeout(run, 500);
8          }
9      }, 500);
10 });
11 // Polling data stores before page is unloaded/navigated away.
12 window.addEventListener('beforeunload', async function(){
13     await pollStorage();
14 })
```

Listing 3: Polling different data stores on page events and on specified intervals to log data.

4.3.5 Capturing global variables. To avoid polluting the global namespaces, functionality can be wrapped in an immediately-invoked function expression (*IIFE*). However, if the code injected by an extension either (a) does not use an *IIFE*, (b) defines a variable without a `var` keyword, or (c) explicitly sets `window.foo`, this results in a globally accessible variable. To detect these variables, the attacker Web page can enumerate all the properties available on the global scope. We collect all the identifiers (i.e., variables and function identifiers) that extension-injected code writes on the JavaScript namespace of our test page. Here, we utilize the `Object.getOwnPropertyNames` API to enumerate all the properties on the `window` handle and discard those that are artifacts of our test page or are also seen in the extension-less environment (e.g., browser built-in APIs). This way, an attacker can probe for variables associated with individual extensions to check for their presence on the client side.

4.4 Side Effects: Storage APIs and Messages

Besides direct changes to the global JavaScript scope and variables, extensions can cause other side effects which can be polled for or listened to from JavaScript.

4.4.1 Cookies, LocalStorage, and IndexedDB. In line with the security model of extensions, the content scripts do not share the same namespace with the Web page when accessing storage and cookies. That is, an extension can invoke `document.cookie` from the content script to set a cookie for the page, yet the invocation is not directly observable by the JavaScript running on the Web page. However, the underlying storage/cookie values are shared, i.e., the effect of a newly added cookie can be observed from the page's JavaScript realm. The same applies to both `localStorage` and `sessionStorage` as well as `IndexedDB`. Note that in Firefox, `IndexedDB` cannot be polled without prior knowledge of the name of the database since there is no implementation of the `indexeddb.databases` API [37], which is why we do not collect any data for it. To observe values injected by the extension, the attacker simply polls the storages to see whether they contain any content. This is shown in Listing 3, where the attacker's code iterates over all storages every 500 ms to see if the extension stored any data. For our purposes, we do not

```
non_unique_features = set()
unique_exts = set()
for feature, extension_ids in feat_to_exts.items():
  if len(extension_ids) == 1:
    unique_exts.update(extension_ids)
  else:
    non_unique_features.add(feature)

for id1, id2 in itertools.permutations(non_unique_features, 2):
  intersection = feat_to_exts[id1] & feat_to_exts[id2]
  if len(intersection) == 1:
    unique_exts.update(intersection)
```

Listing 4: Detecting uniquely identifying features

look into the exact values being written to the respective stores but instead focus only on their overall uniqueness.

4.4.2 PostMessages. Last, but not least, the content script can directly communicate with the page through *postMessage*. We note that this vector was already discussed by Karami et al. [24], yet falls into the category of JavaScript-visible side effects, which is why we consider it also in our work. Notably, the content script and the page itself receive any incoming *postMessage* to the page. While, again, the page's JavaScript cannot hook into the `addEventListener` API used by the content script, it can nevertheless register its own event handler to capture all incoming messages. This way, if an extension's content script sends a message to the other components, this can be recorded by the attacker's script. In our initial experiments, we found that while the exact message content often varies (e.g., because of timestamps or randomized values), the *keys* of messages (when using JSON messages) remained stable.

4.5 Data Collection and Identification

To test an extension, we install its *crx* file in a fresh browser instance and visit our specially crafted test pages. Here, the first test page constitutes hooks, as described in Listing 2, and captures the execution traces of any extension-injected code. In the second test page, we poll individual data stores for any data and enumerate global variables set by extensions, as in Figure 2. Since the test pages and the tools used by prior works are not publicly available [24, 48, 53], we could only obtain a prototypical *honey page* used by Karami et al. [24]. We include all the elements (e.g., iframes, audio/video tags, etc.) from the *Carnus* honey page. We further enhance our test pages by triggering a wide range of mouse and keyboard events on page load, corresponding to Table 1. in [48], through dispatching a series of JavaScript events. Here, we dispatch keyboard events for all possible keys and their hotkey combinations. Similarly, we also send mouse events for different elements (i.e., text-selections, image, form fields, etc.). Naturally, extensions that do not react to JavaScript-induced events (i.e., check the `isTrusted` property of the *event* object) will not be triggered. In the end, we collect and dispatch the execution traces to our backend for processing.

Our instrumentation and test pages provide us with the ability to observe changes that an attacker could also observe. However, extensions may use random variable names or use timestamps for keys and values, i.e., a single run does not suffice to identify *persistent* features of an extension. To account for that, we visit our pages *three times per run* for each extension to see which extensions leave any trace that an attacker might observe. We perform this run for a total of *three* times for each of these extensions that exhibited such a behavior. This way, if a feature occurs repeatedly, it cannot be due to random chance but is instead deterministic.

We first extract all the API invocations, variables, messages, etc., for each extension in the dataset and aggregate the number of occurrences (here dubbed visits) in which the feature was observed. Subsequently, we iterate over all the features to identify those that occur repeatedly. For each such feature, we use it as the dictionary key to store those extensions that use the given feature (repeatedly). From this dictionary, we can already trivially find all those extensions that are uniquely identified by a feature. If, for a given feature, the length of the set is exactly 1, this feature uniquely identifies an extension (as shown in line 4 of Listing 4). For all extensions that cannot be detected by a single feature, we try combinations of two features that might be unique to a single extension. Here, we iterate over all combinations of features (within each class, so, e.g., all cookie names are combined) to see if the intersection of the extensions that exhibit that feature is 1. In that case, the attacker who monitors an extension's behavior and observes these two features can conclusively say that the given extension must be installed. Note that the approach could be expanded to also contain 3-tuples of features. However, this significantly increases runtime (which is the cubic relative to the number of features), which is why we did not consider this in our work. Moreover, experimentally, we could verify that all but two extensions in our dataset were fingerprintable through only a single feature within the same class.

5 EVALUATION & RESULTS

With our framework, we now perform an analysis of three different datasets to showcase the potency of our attacks. For this, we first collected all the free extensions from the *Chrome Web Store* and *Mozilla Add-ons Store*, available as of January 3rd, 2024. We refer to them as *Raider* and *Firefox*, respectively. Then, we also gathered the dataset from Karami et al. [24], referred to as *Carnus*, along with the fingerprinting labels from the original findings. In our study, we use these datasets to run our experiments and understand the trend of fingerprinting behavior in the extension ecosystem. Table 1 shows an overview of the datasets we use. Note that our prefiltering step to identify extensions that do not have the necessary permissions reduces the total number of extensions to consider further. In particular, the largest datasets are ours (*Raider*) and the one from *Carnus* with almost 40k extensions each; *Firefox* contains less than 10k extensions. In the following, we analyze those datasets separately: first our Chrome dataset *Raider*, which we subsequently compare with *Carnus*. Finally, we consider *Firefox*.

For each dataset, we first conduct two runs on the entire dataset. In each run, we visit our test page *three* times. This total of six page loads is meant to ensure that as many extensions as possible show

Dataset	Downloaded	After Pre-Filter
Raider	156,997	37,697
Firefox	26,591	9,488
Carnus	104,484	39,890

Table 1: Extension datasets overview

Method	Usage	Repeated	Unique	Only	Installs
Global APIs	1,878	1,872	1,769	-	109,584,572
- Stacktrace	1,878	1,871	1,753	397	108,382,340
- Norm. Stacktrace	1,878	1,871	1,569	(237)	103,582,524
- Caller & Params	1,878	1,868	813	2	32,740,589
Variables	1,730	1,664	1,301	245	67,048,809
Cookies	201	198	154	78	4,709,235
Storage	634	623	391	266	8,317,933
IndexedDB	128	126	32	17	1,580,655
PostMessages	1,069	1,028	737	283	38,519,471
Cross-class	1,634	1,610	1,257	0	48,466,020
Total	3,398	3,308	2,747	-	169,093,032

Table 2: Results for the *Raider* dataset

any behavior. We then retain as our dataset for the next step all those extensions that exhibited any attacker-observable behavior at least once.

Subsequently, we run *three* more times for each extension that exhibited some behavior before to capture the consistent runtime characteristics across multiple runs. Thus, we visit our test page a total of *nine* times per extension. However, in reporting numbers for fingerprintable extensions, we only rely on those that showed precisely the same fingerprintable feature in all nine visits across the three runs, i.e., we provide a lower bound.

5.1 *Raider* Dataset

Extension Detection. Table 2 shows our findings on the Chrome dataset[2]. 3,398 extensions make use of some browser functionality that could be observable by an attacker. Over half of them are related to global APIs being invoked by the injected code. Notably, 1,730 of these extensions pollute the global namespace with variables. Overall, the usage of `IndexedDB` is very limited (128 extensions), yet we see that all of our analyzed features are in use by extensions.

Notably, as discussed before, not all invocations, storage accesses, and messages are deterministic. Consider the example of *postMessages*: here, extensions may send a message that includes a timestamp in a key. This means that the structure of the message changes. In our analysis, we found that 1,028 / 1,069 extensions send messages with the same structure repeatedly. However, even in that case, if two extensions send the exact same message, an attacker cannot tell those two extensions apart. This is highlighted by the fact that 737 / 1,028 of extensions that send deterministic messages actually send *uniquely identifying* messages.

Overall, we find that 3,308 / 3,398 (97%) extensions have some features that occur deterministically. Out of those, we can uniquely identify 2,747. We also see that the stacktrace is the single most significant contributor to identifying extensions uniquely. 1,753 / 2,747 (64%) of extensions that are fingerprintable within their group are detectable through the stacktrace alone. Notably, this is significantly higher than the caller's code, which initially seems counterintuitive. However, our manual analysis showed that extensions frequently leverage libraries. These libraries frequently make use of JavaScript's *strict* mode, which disables the usage of `arguments`, thereby rendering the caller attack infeasible. However, this does not turn off the stacktrace. Considering the normalized stacktrace (i.e., removing extension IDs from the traces) still allows us to detect 1,569 extensions. Note that out of the 397 extensions that could only be fingerprinted through the full stacktrace, 237 would have also been only fingerprintable through the normalized stacktrace. This is important given that if Chrome was to widely adopt randomized runtime identifiers for extensions, the vast majority would still be fingerprintable due to the unique nature of filenames, lines, and line offsets in the call stacks.

We note that while the stacktrace is the most potent attack, all of the other vectors, except for the parameters and the caller code, add fingerprinting surface. (The *Only* column in Table 2 represents the number of extensions fingerprintable exclusively through that individual vector.) Even the rarely occurring `IndexedDB` vector allows us to identify 17 extensions that could not otherwise be detected. Finally, we consider cross-class fingerprints, i.e., those where single features are insufficient to identify an extension, yet combining two features from different classes suffice (e.g., a registered variable together with a specific cookie). The fact that 1,257 extensions can be fingerprinted through cross-class features highlights that extensions frequently exhibit several types of attacker-visible actions. Finally, if we were to disregard the *postMessage* vector (as already discussed by Karami et al. [24]), 2,464 extensions would be fingerprintable (as extensions are often unique by multiple vectors), showing the impact of our newly proposed vectors.

The discovered fingerprinting attacks have an impact on a large user base (based on the installation counts from the Chrome Web Store). Note that the numbers are lower bounds, as the Store provides only inaccurate numbers for popular extensions (e.g., 1,000+). Overall, the extensions that our attacker models can fingerprint are installed by a total of over 169M users (please refer to Appendix A for more details). The most prominent examples are the Malwarebytes Browser Guard and MetaMask, each with over 10M users. Both are fingerprintable through their usage of global APIs by unique stacktraces. In particular, Easy Ad Blocker is another noteworthy example since adblocker blockers can easily detect the presence of the extension. The reported extensions also span across 22 different categories, as listed in Appendix A.

The number of extensions fingerprintable at least once in any of the runs is slightly higher (2,760). However, the additional 13 extensions did not show the same behavior in all three runs, which is why we exclude them and provide a lower bound.

Multi-Extension Analysis. So far, we only focused on individual extensions and their fingerprintability based on the vectors we proposed. However, users often install multiple extensions that may interfere with each other's behavior at runtime. In turn, this may impact the fingerprintability of these extensions in the presence of others or, vice-versa, detections of extensions that do not show

N	2	3	4	5	6	7	8	9	10	Avg.
TP (%)	99.7	99.4	99.3	98.8	97.5	96.6	96.4	97.5	97.4	98.0
FN (%)	0.3	0.6	0.7	1.2	2.5	3.4	3.6	2.5	2.6	1.9
FP (%)	0.4	0.4	0.4	0.4	0.4	0.4	0.5	0.4	0.5	0.4
F1 (%)	99.7	99.5	99.3	99.2	98.5	98.1	97.9	98.5	98.4	98.8

Table 3: Multi-extension results (average over five runs)

[2]Note that the *Total* row is not the sum of the other rows, as extensions often exhibit multiple classes of attacker-observable behavior.

any behavior when run in isolation (e.g., because they require a postMessage to trigger functionality). To test the robustness of our approach, we now analyze the fingerprintable extensions reported by *Raider* in two different multi-extension settings.

Setting 1: Fingerprintable extensions only First, for each extension, we combine it with a set of *N-1* other randomly selected extensions reported as fingerprintable, where *N* ranges from 2 to 10, and load them in the browser to visit the test page and collect the fingerprints. This is analogous to the tests performed by Karami et al. [24]. We then collect all potentially identifying features (i.e., all of the vectors we consider) and store them in a database. Since we know the ground truth of extensions that were installed, we then compare if the behavior observed at runtime can be attributed correctly to the extensions actually loaded during the test. Table 3 shows the results for our multi-extension test with different values of N. We observe that our proposed vectors are able to accurately detect extensions in ~98% cases, even when loaded with multiple other extensions with similar runtime behavior. Across all different values of N, we have an average of 1.9% false-negative cases where *Raider* cannot detect an extension when present. On manual inspection, we found that some extensions execute *blocking* JavaScript code which then delays the execution of the code injected by other extensions and our tool fails to capture their invocations after 30 seconds. In other cases, we saw that some extensions also mask the behavior of others through their operations (e.g., by themselves hooking into APIs, and thus, hindering attribution through the stacktrace).

We also measured an average of 0.4% falsely-labeled cases where our tool detected an extension that was not loaded in the tests (note that we count this relative to the number of fingerprintable extensions). We investigated this further and found that many extensions react to the operations executed by other extensions during the tests (e.g., code injections, global variables, `postMessages`, etc.) and thus, create new but overlapping execution traces for extensions that are not installed, which eventually leads to false attributions. For instance, one extension set the global variable (*web3*) only in the multi-extension tests but not during individual tests. Since this variable was only observed by one *other* extension in the single-extension tests, we falsely flagged said extension as being detected.

Setting 2: All candidate extensions Orthogonally to the previous case, which assumes a user would install N extensions out of a small set of the fingerprintable ones, we also investigated the case of randomly choosing nine other extensions from the 37k extensions which could potentially interact with the page (see Table 1). We instantiate fresh browser instances with these 10 extensions installed and then perform our tests. We do not select an extension more than once to cover as many extensions in our tests as possible. As before, we collect the data and analyze it in the backend to determine the uniqueness of the collected signatures. Note that not all extensions that fulfill the static requirements in their manifest can actually be loaded without error. Therefore, in some cases, less than 10 extensions were loaded. As with the previous multi-extension tests, we perform these tests five times here as well with different combinations of extensions.

Our tests indicated that out of 2,747 extensions, which were consistently fingerprintable (see Table 2), on average 2,680 (98%) were successfully loaded for each of the five runs. We believe this to be a side-effect of Selenium crashing for *some* extensions (for unknown reasons), the chance of which is exaggerated due to us testing ten extensions in parallel. We collected the information on the successfully loaded number of extensions for each test by navigating to the extension page and enumerating the loaded set of extensions using *Selenium*. Of these 2,680 extensions that were successfully loaded, our tool accurately detected 99.1% of them in the second multi-extension setting (averaged over five runs). For the remaining 0.9%, the side effects from other extensions masked the behavior that allowed us to fingerprint them in the single-extension case. We note that this may relate to our test pages loading significantly slower in the presence of multiple extensions.

Similar to the previous multi-extension setting, we found 0.5% false-positive cases where we detected an extension that we did not load for the tests. Here, the falsely detected set of extensions and the false-positive rate are in line with the previous multi-extension setting. Overall, the fingerprinting rates of extensions across different multi-extension settings are similar. Since we randomly sampled almost our complete dataset (i.e., 34,774 / 37,697 extensions with permissions to interact with the page across five runs) for our experiments, we do not believe that the false-positive rate would be significantly higher across other combinations of extensions installed by the user.

Method	Usage	Repeated	Unique	New	No WAR
Global APIs	894	889	712	37	207
- Stacktrace	894	889	699	37	-
- Norm. Stacktrace	894	889	627	37	186
- Caller & Params	894	885	429	31	120
Variables	1,136	979	638	47	208
Cookies	80	78	31	8	15
Storage	423	419	242	93	149
IndexedDB	15	14	11	3	8
PostMessages	497	474	273	14	34
Cross-class	775	764	474	18	125
Total	2,119	1,943	1,355	180	484

Table 4: Results for the *Carnus* dataset

5.2 *Carnus* Dataset

As a second dataset, we rely on *Carnus*' [24] set of extensions we received from the authors. Fortunately, the authors provided us with the complete test dataset, i.e., both fingerprintable and non-fingerprintable extensions, as reported by them. This labeled dataset allows us to see how many additional extensions our attacks can fingerprint on top of *Carnus*' methods. Overall, *Carnus* can detect 29,428 extensions, the vast majority of which is identifiable through WAR-based methods (25,866). We note that since their work, browsers have introduced the option to enable dynamic URLs for WARs [39]. This means that extensions can opt to no longer use a deterministic identifier for the WARs but, instead, a randomized one that resets on reloading the extension or restarting the browser. If this is set, an attacker can no longer probe for specific resources, as the random runtime identifier is not mapped to an extension.

In line with our approach for *Raider*, we confirm that extensions are fingerprintable in all three repetitions of the analysis run. We find that 1,355 extensions are consistently identifiable by our methods. Considering the direct comparison with *Carnus*, 180 of

Method	Usage	Repeated	Unique	Only
Global APIs	436	432	367	-
- Norm. Stacktrace	436	432	353	0
- Caller & Params	436	423	182	14
Variables	359	351	288	79
Cookies	25	24	19	14
Storage	85	84	55	43
IndexedDB	-	-	-	-
PostMessages	176	172	138	54
Cross-class	314	305	242	0
Total	689	682	572	-

Table 5: Results for the *Firefox* dataset

these extensions were not fingerprintable through *Carnus*' techniques (i.e., they were not contained in the list shared with us by the *Carnus'* authors). However, their approach relies on a significant fraction of extensions with unique WAR URLs. Therefore, Table 4 also shows how many extensions we could detect that *Carnus* could not if randomized runtime identifiers were enabled (dubbed No WAR). Here, we find that our approach would still be able to fingerprint 484 extensions that *Carnus* would not be able to fingerprint anymore. For a fair comparison, we considered only the normalized stacktrace here, as the full stacktrace would also be affected by randomized runtime identifiers. This highlights that even if WAR-based fingerprinting becomes infeasible, our attacks add significant fingerprinting surface to the state of the art, which can also not be overcome by existing defense mechanisms. More importantly, our fingerprinting techniques cannot be overcome easily by readily-available countermeasures in modern browsers.

5.3 *Firefox* Dataset

Last but not least, we turn our attention to the Firefox dataset. Overall, the number of considered extensions is significantly lower than the Chrome store datasets of both *Raider* and *Carnus*. This is also observed in the much lower number of extensions that have any behavior that our attacker model could observe. As with Chrome, the most potent vectors for Firefox are also *stacktraces*, which allow for the detection of 353 extensions. We note that Firefox already automatically randomizes runtime identifiers. Therefore, Table 5 also omits the stacktrace row, as we can only rely on the normalized stacktraces. The *variables* are the second-most potent vector as 288 of these extensions also set global variables in the shared namespace leading them to be fingerprintable. Our findings highlight that the potential for fingerprinting through our vectors does not only affect the Chrome extension ecosystem, but the patterns exhibited by the extensions also generalize to Firefox. Again, the numbers present a lower bound as all 572 extensions could be fingerprinted in *three separate runs* (i.e., *nine distinct visits*).

6 DISCUSSION

In this section, we discuss our disclosure and limitations, followed by an overview of existing defense mechanisms (and why they are insufficient for our attacks). Finally, we discuss potential countermeasures to mitigate the identified vectors.

6.1 Ethics and Responsible Disclosure

As we plan to open-source our pipeline to allow for follow-up research, adversaries could use these fingerprints to identify users. Thus, in October 2023, we followed best practices in notifications [55, 56] and informed the developers of the Chrome and Firefox extensions reported by our pipeline. We provided the developers with a proof-of-concept testbed, allowing them to test and limit the fingerprintability of their extensions against our vectors in the future [45]. So far, we sent out notifications for a total of 1,594 Chrome and 273 Firefox extensions. 30 developers replied to our notification. 16 of them positively acknowledged the underlying issue in their extensions. Six of them did not understand the threat and followed up further. In contrast, four developers mentioned that *security and privacy is not their primary concern* or that *"it should be the platform's responsibility to take care of such issues"*. Another four developers indicated that it is a known problem but also *unavoidable* for their functionality (e.g., crypto-wallet extensions).

Furthermore, we discussed with three developers who showed interest in understanding the problem in detail as well as finding potential solutions for individual cases. They indicated that they inject scripts for including third-party libraries (e.g., *React* libraries), creating overlays, loading fonts, and so on, which are crucial to the extensions' functionalities. They also mentioned the *lack of dedicated API* (in the current architecture) that injected scripts could use to communicate with other extension components (i.e., content scripts or popups) instead of using the postMessage API. To summarize, all three developers were unaware that their extensions were fingerprintable and positively acknowledged our findings. Two of them also affirmed that they will try to reduce the usage of the fingerprinting vectors we uncovered wherever possible.

6.2 Limitations

We utilize a hybrid analysis pipeline in this study to detect uniquely identifiable extensions with respect to the newly discovered fingerprinting strategies discussed in this paper. However, we strictly note that our tool only reports a *lower bound* of all the potentially identifiable extensions available in the stores due to certain limitations (and design choices) of our approach. In the first stage, we filter out extensions that do not contain a valid declaration of content scripts, background scripts or WARs with appropriate host permissions (as discussed in Section 4.2). However, extensions can also inject or update content scripts from their background, using the `scripting`, `tabs`, or `activeTab` permissions. While extensions may request any of these permissions to inject scripts at runtime, they may only exhibit this behavior on *specific hosts*. For example, OffShip - Online Shopping Carbon Offsets has script injection capabilities on `<all_urls>` but only injects something on the *Amazon* and *Walmart* domains, through the `location` check at runtime.

Next, our dynamic step necessitates extensions to exhibit *consistent* runtime behavior on our test page. This has certain limitations. *i.)* An extension may cloak its runtime behavior through runtime logic. *ii.)* In cases where the extension-injected code executes before the test page JavaScript, we do not capture any data from the tests. This, in particular, is due to the race condition when extensions inject code on `document_start`, and the injected code may execute before any script on the attacker page [16]. To not be impacted by

the side effects of race conditions or inconsistent runtime behavior, we only considered extensions that showed consistent results for nine visits on our test page and omitted other cases from our findings. However, other extensions could still be identifiable through multiple visits to the attacker page, thus, our results present a *lower bound* of the fingerprintable extensions we uncovered. Also, our polling approach does not work for extracting `IndexedDB` data in *Firefox* since the API to enumerate over all available databases, i.e., `indexeddb.databases`, is not implemented for *Firefox* [37].

6.3 Existing Fingerprinting Defenses

Prior work has suggested specific mitigation techniques for their detected fingerprinting vectors. For instance, creating Shadow or Parallel DOM, as proposed by Laperdrix et al. [28] and Karami et al. [25], that is inaccessible to the Web page JavaScript, can only prevent DOM and style-based fingerprinting. Similarly, the preventive strategies by Trickel et al. [59] only tackle DOM-based side effects. More importantly, none of the existing anti-fingerprinting defense strategies could protect extensions against the set of vectors we proposed in this study. This is due to the underlying architecture, such that the extensions, although in different processes, have shared access to the client-side storage. Moreover, the injected scripts also execute in the same JavaScript namespace as the visited Web page. Next, Sjösten et al. [46] suggest randomizing the pointers to the web-accessible resources included by extensions, which can be enabled nowadays through the `use_dynamic_url` key [39]. At the time of our study, we only observe 109 fingerprintable extensions using this option. Note, however, that our approach would still allow extensions to be fingerprinted, even if they all used randomized dynamic pointers to WARs. In particular, we found that even removing randomized parts of the URLs within the collected stacktraces leaves sufficient entropy through line numbers and offsets to uniquely identify extensions.

6.4 Extension Ecosystem & Standards

The *Manifest V3* standards for extensions hosted on the Chrome Web Store have certain restrictions and built-in protection mechanisms in place to protect against many critical security vulnerabilities and privacy leaks due to the underlying design changes. The features introduced, such as blocked remote-code inclusion and strict CSP rules, may help limit security issues on the client side that originate from extensions. However, extensions can still be fingerprintable with respect to our proposed vectors, as they often inject code that executes in the applications' context, thus causing observable side-effects. This supports our findings, given that 1,611 / 2,747 of the fingerprintable extensions we detected are, in fact, *Manifest V3* standards. Besides, our proposed fingerprinting strategies are not limited to the Chrome extension ecosystem and also apply to Mozilla Add-ons and other extension stores. We show the versatility of our approach by running our tests on Firefox extensions, as discussed in Section 5.3, as the underlying *Web Extensions* architecture is similar across the extension ecosystems [35].

6.5 Recommendations and Mitigation Strategies

In this section, we discuss ways in which developers can avoid exposing fingerprintable behavior to an attacker. We first discuss how to partially stop attacker code from hooking into APIs, discuss how to ensure variables do not leak to the global scope, provide alternative means of storing client-side information, and finally outline how to avoid postMessages being captured.

Global APIs. Unfortunately, it would be non-trivial for extension developers to prevent observable side effects caused by injected code, because of the underlying extension architecture. In fact, there are legitimate use cases where extensions may require code injection into the context and the namespace of the visited Web page. Thus, preventing extensions from injecting code will limit their functionality. To ensure that an attacker-controlled page cannot hook into APIs called by extension code, the extension developer has two options: run all their code which uses the APIs *before* the attacker's code executes or ensure that these APIs cannot be overwritten. Note that storing clean references for later use would most likely again leave traces (as this requires additional variables). To allow for this approach to work, the extension code, therefore, needs to run before the attacker's code.

An extension can, at the earliest, inject a script into the page at `document_start`, i.e., when the browser parses and renders the HTML content. For MV2 extensions, we empirically validated that if an extension injects an inline script (i.e., a programmatically created script element with an `innerText` property), this will execute before any page JavaScript. However, the scripts injected through their URL (i.e., the `script.src` property) execute after the page JavaScript; thus, they are observable by an attacker[3]. For MV3, injection of inline scripts is no longer possible since the minimum CSP constraints for content scripts does not allow inline script injections [40]. However, extensions can specify that a content script should run in the *MAIN* world [19] (i.e., is injected directly into the page). Here, we confirmed that the extension code reliably runs before the page JavaScript.

One can prevent a JavaScript API from being overwritten by freezing their native definition through the `Object.freeze` API, thus, maintaining the integrity of respective APIs [41]. Extension developers could use this mechanism to freeze the native definition of all global JavaScript APIs, through the content script that executes in the *MAIN* world, before any page JavaScript executes. This would prevent the attacker from capturing any execution traces at runtime. However, this only works for the APIs and properties associated with global JavaScript objects (i.e., *Array.prototype*, *String.prototype*, etc.), while the `window` APIs and properties (i.e., `postMessage`, `setTimeout`, etc.) cannot be frozen. We extracted the unique features that we collected for 1,769 extensions fingerprintable through the global APIs from the *Raider* dataset. We found that 829 of these will still be fingerprintable after freezing all possible JavaScript APIs in the global namespace. We note that freezing global JavaScript objects might also cause unintended side-effects to benign websites which may extend or overwrite these APIs for their functionality.

Variables. Further, developers could avoid fingerprinting through global variables by either scoping them appropriately (e.g., through the `var` keyword) or wrap the injected code within an *Immediately Invoked Function Expression (IIFE)*, since the execution context of an *IIFE* is destroyed right after it executes. This way, no function

[3]More details on the tests at https://raider-ext.github.io/raider/tests/

definitions would pollute the global scope. Naturally, if an extension needs to register global variables for interaction with a page, developers might intentionally expose variables. Therefore, any change in such exposure would imply a negative effect on the functionality.

Storage APIs. While extensions may need to store runtime information related to visited websites (e.g., preferred UI settings), this can serve as a source of entropy for tracking users. In this case, extension developers should utilize the `chrome.storage` API instead (by replacing the `localStorage` invocation with `chrome.storage` in the code), where the storage container is accessible only to individual extensions. Further, suppose an extension needs to store a large chunk of data on the client side, i.e., in the `IndexedDB`. In that case, we recommend that developers store the data in the extensions' context, keyed with their origin, instead of setting them in the Web context [9, 12]. Similarly, we recommend that developers do not set cookies on arbitrary domains since an attacker can also observe them through JavaScript or in an incoming request to the attacker-controlled server.

Messages. Assuming the extension can inject its code before the page JavaScript executes, it can ensure that postMessages are only relayed after filtering out the extension-sent ones. Specifically, the extension can overwrite the `addEventListener` API and `window.onmessage` property to ensure that if an event handler is registered, the handler is only invoked if a message *does not* originate from the extension. Doing so, the attacker will be unable to recover extension-sent messages. However, this leads to an overwritten global API, which could be used as a vector for fingerprinting yet again. Thus, the defense only works if multiple extensions rely on the same approach to ensure a greater anonymity set.

Separated execution of extension-injected scripts. Orthogonally, similar to ShadowDOM, extensions could also have access to a "ShadowWindow" object, which would provide them with a separate global namespace. Thus, any extension-incurred changes (e.g., variable registration, API usage, etc.) would not be visible to the Web page JavaScript. This aligns with the ongoing discussion among the stakeholders of the *WebExtensions* framework [60]. Overall, we also urge browser vendors to take the necessary steps toward upgrading the isolation boundaries of the extensions in this regard. Unfortunately, evaluating the unintended side-effects of this stricter isolation approach would be non-trivial, as it is extremely challenging to automatically infer whether an extension developer actually wanted to interact with the page's global object or not.

7 RELATED WORK

Browser Extension Fingerprinting. In 2017, Starov and Nikiforakis [53] first quantified the fingerprinting characteristics of Chrome extensions based on the extensions' interaction with the DOM. They instrumented the content scripts of the extensions to create the required DOM structure on the fly, and they captured the extensions' runtime interactions with the DOM. However, this fingerprinting strategy does not work for content scripts now, as they share a different DOM handle and are not accessible to the Web page [34]. In 2020, Karami et al. [24] automated Chrome extensions' fingerprinting based on their interaction with the DOM and through their communication patterns with different client- and server-side components, including *postMessages*. They built *honeypages*, based on the extensions' description provided by the developers, to trigger the extensions' runtime behavior. In 2022, Solomos et al. [49] leveraged the *MutationObserver* to capture any DOM modifications attributed to individual extensions during execution. In fact, previous work only compared the DOM before/after execution, thus missing the invisible and transient interactions that happen during execution but are not observable after execution anymore. Finally, they also showed that user-induced runtime events, such as keyboard and mouse events, could increase the extensions' interaction with the DOM [48], and thus, their fingerprintability.

The above studies only focus on DOM-based interactions between extensions and websites. Instead, we focus on a set of fingerprinting vectors that are agnostic to DOM-based side effects. We show that the execution of extensions' injected scripts in the realm of Web application could leave traces in the global JavaScript namespace (e.g., global variables). This is because the attacker JavaScript can overwrite the global JavaScript APIs and properties to capture their invocations, which then adds to the fingerprinting surface for extensions [29]. Moreover, we highlight that any interactions with the client-side *Storage* APIs and other global JavaScript APIs in the applications' context further aid in fingerprinting extensions.

In 2017, Sjösten et al. [47] showed that browser extensions are identifiable through the web-accessible resources (WARs) they include on different websites. In 2019, they further found that tracking websites could probe for these included resources even when the browser randomizes the extension runtime identifier used to fetch these resources [46]. Fortunately, the current extension architecture mitigates any attempt of WAR-based fingerprinting by opting into *dynamic URLs* [39]. Around the same time, Starov et al. [52] showed that unnecessary code bloats within extensions could also serve as a fingerprinting vector. In 2021, Laperdrix et al. [28] showed that extensions could also be fingerprinted based on the stylesheet injection patterns observable from Web pages. Overall, these fingerprinting techniques are orthogonal to the set of vectors we introduce in this study. In fact, we investigate the observable side-effects of the execution of extension resources, even when the attacker application cannot probe for an extension's existence because of security measures in place at runtime.

Browser Extension Fingerprinting Defenses. In 2019, Trickel et al. [59] proposed a mitigation technique to counter DOM-based extension fingerprinting. By randomizing the DOM element identifiers, an attacker can no longer attribute them to individual extensions. In 2022, Karami et al. [25] suggested having a separate copy of the actual DOM, a *Parallel DOM*, for page-based interactions vs. a *User DOM* for the extension-based interactions and inaccessible to the Web page. This is similar to the concept of *Shadow DOM*, proposed by Laperdrix et al. [28] in 2021, to isolate the website's view and the extensions' view of the DOM. In 2022, Solomos et al. [48] suggested using the `isTrusted` property of *events* when listening to them to avoid side effects of fake events dispatched by the attacker. Unfortunately, none of these defense strategies protect extensions against the set of fingerprinting vectors we discuss in this paper. This is because extensions may cause observable side-effects on the `window` (e.g., `window.localStorage` or globally-accessible variables), even observable to an attacker with a restricted view of the DOM.

Malicious & Vulnerable Extension Analyses. Several researchers have discovered malicious extensions that perform unwanted actions on target websites, such as ad injections, social-media hijacking, malware downloads, etc. [3, 22, 23, 44, 58, 61]. In particular, Hsu et al. [21] conducted a longitudinal and comparative analysis of security-noteworthy extensions. Pantelaios et al. [43] also discovered that extensions could receive updates, making them turn malicious *after* being added to the Chrome Web Store. In particular, Hsu et al. [21] conducted a longitudinal and comparative analysis of security-noteworthy extensions. Chen and Kapravelos [6] utilized taint-tracking to find extensions that leak privacy-sensitive user data to third-party websites. Somé [51], Fass et al. [20], and Yu et al. [62] showed that message-passing APIs could be abused to exploit browser extension capabilities, allowing an attacker Web page to perform privileged operations on the client side. Agarwal [2] showed that extensions often alter security-related HTTP headers to implement functionalities, although by degrading the security of the target website. In 2023, Kim and Lee [26] asserted that malicious websites could exploit over-privileged extensions to escalate their privileges and perform different attacks. For our study, we do not explicitly analyze extensions to detect any malicious or vulnerable characteristics that may lead to security issues on the client. Rather, we detect installed extensions on a user's machine. An attacker can subsequently learn privacy-sensitive user information associated with an extension or even exploit known vulnerabilities within an extension to perform malicious operations.

8 CONCLUSION

Browser extensions are omnipresent and, therefore, a prime target for fingerprinting. We extend the state-of-the-art research by introducing two new fingerprinting vectors: (1) the execution traces on the global JavaScript namespace from extension-injected scripts; and (2) the side effects of extensions' interactions with client-side Storage APIs and postMessages. Doing so, we found that 2,747 current Chrome extensions, installed by almost 169M users, can be fingerprinted. Importantly, the discovered attacks affect the Chrome and Firefox ecosystems alike, highlighting that insecure coding practices that lead to exposing fingerprintable information to the attacker's page occur frequently and across browsers.

ACKNOWLEDGMENTS

We would like to thank our reviewers for their valuable feedback. This work was conducted in the scope of a dissertation at the Saarbrücken Graduate School of Computer Science.

REFERENCES

[1] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. 2014. The web never forgets: Persistent tracking mechanisms in the wild. In *CCS*.
[2] Shubham Agarwal. 2022. Helping or Hindering? How Browser Extensions Undermine Security. In *CCS*.
[3] Anupama Aggarwal, Bimal Viswanath, Liang Zhang, Saravana Kumar, Ayush Shah, and Ponnurangam Kumaraguru. 2018. I spy with my little eye: Analysis and detection of spying browser extensions. In *IEEE Euro S&P*.
[4] Pouneh Nikkhah Bahrami, Umar Iqbal, and Zubair Shafiq. 2022. FP-Radar: Longitudinal measurement and early detection of browser fingerprinting. In *PETS*.
[5] Yinzhi Cao, Song Li, Erik Wijmans, et al. 2017. (Cross-) Browser Fingerprinting via OS and Hardware Level Features. In *NDSS*.
[6] Quan Chen and Alexandros Kapravelos. 2018. Mystique: Uncovering Information Leakage from Browser Extensions. In *CCS*.
[7] Chrome Developers. 2014. externally_connectable. https://developer.chrome.com/docs/extensions/mv3/manifest/externally_connectable/
[8] Chrome Developers. 2017. Match Patterns. https://developer.chrome.com/docs/extensions/mv3/match_patterns/
[9] Chrome Developers. 2023. Can extensions use web storage APIs? https://developer.chrome.com/docs/extensions/reference/api/storage#can_extensions_use_web_storage_apis
[10] Chrome Developers. 2023. Chrome Extensions Sitemap. https://chrome.google.com/webstore/sitemap
[11] Chrome Developers. 2023. chrome.bookmarks. https://developer.chrome.com/docs/extensions/reference/bookmarks/
[12] Chrome Developers. 2023. chrome.offScreen. https://developer.chrome.com/docs/extensions/reference/api/offscreen
[13] Chrome Developers. 2023. chrome.runtime. https://developer.chrome.com/docs/extensions/reference/runtime/
[14] Chrome Developers. 2023. chrome.scripting.executeScript. https://developer.chrome.com/docs/extensions/reference/scripting/#method-executeScript
[15] Chrome Developers. 2023. Declare Permissions. https://developer.chrome.com/docs/extensions/mv3/declare_permissions/
[16] Chrome Developers. 2023. Inject with dynamic declarations. https://developer.chrome.com/docs/extensions/mv3/content_scripts/#dynamic-declarative
[17] Chrome Developers. 2023. Message Passing. https://developer.chrome.com/docs/extensions/mv3/messaging/
[18] Chrome Developers. 2023. Offline Data. https://web.dev/learn/pwa/offline-data/
[19] Chrome for Developers. 2024. Inject Scripts. https://developer.chrome.com/docs/extensions/develop/concepts/content-scripts#functionality
[20] Aurore Fass, Dolière Francis Somé, Michael Backes, and Ben Stock. 2021. DoubleX: Statically Detecting Vulnerable Data Flows in Browser Extensions at Scale. In *CCS*.
[21] Sheryl Hsu, Manda Tran, and Aurore Fass. 2024. What is in the Chrome Web Store?. In *AsiaCCS*.
[22] Nav Jagpal, Eric Dingle, Jean-Philippe Gravel, Panayiotis Mavrommatis, Niels Provos, Moheeb Abu Rajab, and Kurt Thomas. 2015. Trends and lessons from three years fighting malicious extensions. In *USENIX Security*.
[23] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. 2014. Hulk: Eliciting malicious behavior in browser extensions. In *USENIX Security*.
[24] Soroush Karami, Panagiotis Ilia, Konstantinos Solomos, and Jason Polakis. 2020. Carnus: Exploring the Privacy Threats of Browser Extension Fingerprinting.. In *NDSS*.
[25] Soroush Karami, Faezeh Kalantari, Mehrnoosh Zaeifi, Xavier J Maso, Erik Trickel, Panagiotis Ilia, Yan Shoshitaishvili, Adam Doupé, and Jason Polakis. 2022. Unleash the Simulacrum: Shifting Browser Realities for Robust {Extension-Fingerprinting} Prevention. In *USENIX Security*.
[26] Young Min Kim and Byoungyoung Lee. 2023. Extending a hand to attackers: browser privilege escalation attacks via extensions. In *USENIX Security*.
[27] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. 2016. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *IEEE S&P*.
[28] Pierre Laperdrix, Oleksii Starov, Quan Chen, Alexandros Kapravelos, and Nick Nikiforakis. 2021. Fingerprinting in style: Detecting browser extensions via injected style sheets. In *USENIX Security*.
[29] Sebastian Lekies, Ben Stock, Martin Wentzel, and Martin Johns. 2015. The Unexpected Dangers of Dynamic JavaScript. In *USENIX Security*.
[30] Xu Lin, Frederico Araujo, Teryl Taylor, Jiyong Jang, and Jason Polakis. 2022. Fashion Faux Pas: Implicit Stylistic Fingerprints for Bypassing Browsers' Anti-Fingerprinting Defenses. In *IEEE S&P*.
[31] Mozilla Developer Network. 2023. IndexedDB API. https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API
[32] Mozilla Developer Network. 2023. Web Storage API. https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API
[33] Mozilla Developer Network. 2023. Window.localStorage property. https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage
[34] Mozilla Developer Network. 2024. DOM Access. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Content_scripts#dom_access
[35] Mozilla Developer Networks. 2023. Browser Extensions. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions
[36] Mozilla Developer Networks. 2023. Function.prototype.caller. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/caller
[37] Mozilla Developer Networks. 2023. IDBFactory: databases() method. https://developer.mozilla.org/en-US/docs/Web/API/IDBFactory/databases
[38] Mozilla Developer Networks. 2023. Standard built-in objects. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects
[39] Mozilla Developer Networks. 2023. web_accessible_resources. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/web_accessible_resources

[40] Mozilla Developer Networks. 2024. CSP for content scripts. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Content_Security_Policy#csp_for_content_scripts
[41] Mozilla Developer Networks. 2024. Object.freeze(). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze
[42] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. 2013. Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting. In *IEEE S&P*.
[43] Nikolaos Pantelaios, Nick Nikiforakis, and Alexandros Kapravelos. 2020. You've Changed: Detecting Malicious Browser Extensions through Their Update Deltas. In *CCS*.
[44] Raffaello Perrotta and Feng Hao. 2018. Botnet in the browser: Understanding threats caused by malicious browser extensions. In *IEEE S&P*.
[45] Raider. 2024. *Artifacts*. https://github.com/raider-ext/raider
[46] Alexander Sjösten, Steven Van Acker, Pablo Picazo-Sanchez, and Andrei Sabelfeld. 2019. Latex Gloves: Protecting Browser Extensions from Probing and Revelation Attacks.. In *NDSS*.
[47] Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld. 2017. Discovering browser extensions via web accessible resources. In *CODASPY*.
[48] Konstantinos Solomos, Panagiotis Ilia, Soroush Karami, Nick Nikiforakis, and Jason Polakis. 2022. The dangers of human touch: fingerprinting browser extensions through user actions. In *USENIX Security*.
[49] Konstantinos Solomos, Panagiotis Ilia, Nick Nikiforakis, and Jason Polakis. 2022. Escaping the Confines of Time: Continuous Browser Extension Fingerprinting Through Ephemeral Modifications. In *CCS*.
[50] Konstantinos Solomos, John Kristoff, Chris Kanich, and Jason Polakis. 2021. Tales of favicons and caches: Persistent tracking in modern browsers. In *NDSS*.
[51] Dolière Francis Somé. 2019. Empoweb: empowering web applications with browser extensions. In *IEEE S&P*.
[52] Oleksii Starov, Pierre Laperdrix, Alexandros Kapravelos, and Nick Nikiforakis. 2019. Unnecessarily Identifiable: Quantifying the fingerprintability of browser extensions due to bloat. In *WWW*.
[53] Oleksii Starov and Nick Nikiforakis. 2017. Xhound: Quantifying the fingerprintability of browser extensions. In *IEEE S&P*.
[54] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. 2019. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild.. In *NDSS*.
[55] Ben Stock, Giancarlo Pellegrino, Frank Li, Michael Backes, and Christian Rossow. 2018. Didn't you hear me? — Towards more successful Web Vulnerability Notifications. In *NDSS*.
[56] Ben Stock, Giancarlo Pellegrino, Christian Rossow, Martin Johns, and Michael Backes. 2016. Hey, You Have a Problem: On the Feasibility of Large-Scale Web Vulnerability Notification. In *USENIX Security*.
[57] Junhua Su and Alexandros Kapravelos. 2023. Automatic Discovery of Emerging Browser Fingerprinting Techniques. In *WWW*.
[58] Kurt Thomas, Elie Bursztein, Chris Grier, Grant Ho, Nav Jagpal, Alexandros Kapravelos, Damon Mccoy, Antonio Nappa, Vern Paxson, Paul Pearce, Niels Provos, and Moheeb Abu Rajab. 2015. Ad Injection at Scale: Assessing Deceptive Advertisement Modifications. In *IEEE S&P*.
[59] Erik Trickel, Oleksii Starov, Alexandros Kapravelos, Nick Nikiforakis, and Adam Doupé. 2019. Everyone is different: Client-side diversification for defending against extension fingerprinting. In *USENIX Security*.
[60] WebExtensions. 2023. User Scripts API. https://github.com/w3c/webextensions/blob/main/proposals/user-scripts-api.md
[61] Xinyu Xing, Wei Meng, Byoungyoung Lee, Udi Weinsberg, Anmol Sheth, Roberto Perdisci, and Wenke Lee. 2015. Understanding Malvertising Through Ad-Injecting Browser Extensions. In *WWW*.
[62] Jianjia Yu, Song Li, Junmin Zhu, and Yinzhi Cao. 2023. CoCo: Efficient Browser Extension Vulnerability Detection via Coverage-guided, Concurrent Abstract Interpretation. In *CCS*.

A APPENDIX

```
// API Type 1
var foo = [1, 2, 3];
foo.forEach((element) => {
    console.log(element);
})
// API Type 2
var bar = new Set([1, 2, 3]);
console.log(Array.isArray(bar));
```

Listing 5: JavaScript APIs and their execution contexts.

```
"content_scripts": [
    {
        "js": ["content_script.js"],
        "matches": ["https://*/*"]
    }
],
"web_accessible_resources": [
    {
        "resources": ["storage.js"],
        "matches": ["<all_urls>"]
    },
    {
        "resources": ["cookies.js"],
        "matches": ["*://*/*"]
    }
]
```

Listing 6: Extensions with relevant content scripts & WARs.

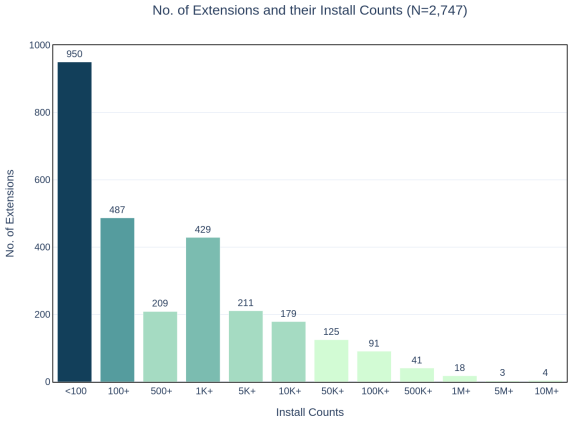

Figure 4: The install counts for 2,747 Chrome extensions reported by *Raider*.

Categories	# Extensions	Categories	# Extensions
Workflow & Planning	1,074	Fun	55
Developer Tools	600	Just for Fun	52
Tools	270	Privacy & Security	39
Accessibility	174	Education	27
Shopping	144	Communication	22
Social Networking	132	Functionality & UI	14
Productivity	72	Social & Communication	12
Art & Design	8	Entertainment	7
News & Weather	7	Well-being	2
Photos	2	Games	2
Household	1	Travel	1

Table 6: Categories of extensions reported by *Raider* to be fingerprintable. We could not extract any explicit category for 30 extensions from the Chrome Web Store.