The Dangers of Human Touch: Fingerprinting Browser Extensions through User Actions

Konstantinos Solomos[†], Panagiotis Ilia[†], Soroush Karami[†], Nick Nikiforakis[±], and Jason Polakis[†]

[†]University of Illinois at Chicago, {ksolom6,pilia,skarami,polakis}@uic.edu
[±]Stony Brook University, nick@cs.stonybrook.edu

Abstract

Browser extension fingerprinting has garnered considerable attention recently due to the *twofold* privacy loss that it incurs. Apart from facilitating tracking by augmenting browser fingerprints, the list of installed extensions can be directly used to infer sensitive user characteristics. However, prior research was performed *in a vacuum*, overlooking a core dimension of extensions' functionality: how they react to user actions. In this paper, we present the first exploration of *user-triggered* extension fingerprinting. Guided by our findings from a large-scale static analysis of browser extensions we devise a series of *user action templates* that enable dynamic extension-exercising frameworks to comprehensively uncover hidden extension functionality that can only be triggered through user interactions. Our experimental evaluation demonstrates the effectiveness of our proposed technique, as we are able to fingerprint 4,971 unique extensions, 36% of which are not detectable by state-of-the-art techniques. To make matters worse, we find that ≈67% of the extensions that require mouse or keyboard interactions lack appropriate safeguards, rendering them vulnerable to pages that simulate user actions through JavaScript. To assist extension developers in protecting users from this privacy threat, we build a tool that automatically includes origin checks for fortifying extensions against invasive sites.

1 Introduction

Web browsers have evolved into complex software delivery and execution platforms with an ever-expanding set of capabilities, while capitalizing on technological advancements for improving the user experience through novel functionality. Unfortunately, the continuous deployment of new functionality and features comes at a price, as new avenues for privacy loss can be introduced. In fact, prior work has demonstrated how browser mechanisms and features can be misused for exfiltrating users' personally identifiable or sensitive information [22,27,28,33,35]) and persistently tracking users [13,45].

Accordingly, the prevalence of web tracking [34] has heightened users' privacy concerns, pressuring browser vendors to provide better protections [40]. In fact, major browsers continue to deploy anti-tracking defenses that aim to hinder cookie-based tracking [21, 49, 53]. At the same time, this paradigm-shift towards cookie-less tracking has resulted in an increasing number of trackers adopting browser fingerprinting techniques [41,42] that aim to identify, and by extension track, devices based on the uniquely-identifying characteristics of the browsers and underlying operating systems and hardware [14, 15, 18, 20, 23, 25, 30, 31, 37–39, 52]. More recently, researchers have explored techniques for detecting which browser extensions are installed [32, 47, 48], which can be used for augmenting browser fingerprints but also for automatically inferring sensitive user characteristics [29].

Prior work on browser extension fingerprinting focused on features that can be detected statically (e.g., unique resources that are accessible to web pages) or dynamic behaviors that occur automatically when an orchestrated browser with an installed extension visited a specially crafted webpage (i.e., the *honeypage*). However, all these studies explored extension fingerprinting *in a vacuum*, without considering how user (inter)actions actually affect the fingerprintability of extensions. As extensions aim to extend browsers' functionality and offer additional, and often specialized features, it is natural that such actions may only occur *after* explicit user actions (e.g., highlighting some text, right-clicking it, and selecting an action from the context menu). In other words, the threat model considered by all prior work provides a limited view of extension fingerprintability in realistic settings, and overlooks how the presence of users introduces an additional dynamic.

In this paper we present the first, to the best of our knowledge, exploration of how *user actions can trigger unique behaviors in browser extensions*, thus allowing invasive or malicious pages to infer that the user has installed specific extensions. To that end, we first perform an analysis of extensions' metadata coupled with a static analysis of their code, in order to extract information about the extensions' behavior and potential triggers. Specifically, we focus on extensions that can run on any domain and include a set of permissions and entries in their manifest that define interactive extension

components such as the extension's browser icon and the context menu items. Subsequently, we analyze the extracted data and generate three different classes of *behavioral templates*. These templates are built on top of unique and exclusive user interactions that we categorize based on the actions that they represent (i.e., involving the mouse, keyboard, or browser interface). We follow a continuous testing approach so as to achieve broad coverage and create a comprehensive collection of interactions, which we implement as a dynamic extension-exercising module that can be easily incorporated into extension-analysis frameworks. Our module uses the behavioral templates and the extensions' metadata to exercise each extension, and detects unique fingerprintable behaviors that manifest as either changes to the honeypage's DOM or messages sent from the extension to the honeypage.

We evaluate our system's extension fingerprinting capabilities on three different datasets, that capture different chronological snapshots of Chrome's Web Store. Specifically, we use the dataset by Karami et al. [29] and the dataset of *detected* extensions by Lapperdrix et al. [32]. To enable a more extensive longitudinal analysis we also crawl the Chrome Web Store and collect recent versions of extensions and new extensions that were not included in the two datasets from prior work. The experimental evaluation of our novel user-driven triggering techniques results in the detection of 4,971 browser extensions. When comparing to state-of-the-art behavior-based fingerprinting [29], we find that ≈64% of the extensions can only be detected through user-driven interactions. For the other dataset used in our evaluation [32], we were only able to obtain their detected extensions, so we cannot calculate how many extensions missed by their approach are solely detectable through user actions.

We also identify the lack of appropriate safeguards for verifying the provenance of received `events` in extensions. This can be exploited by pages through JavaScript by simulating mouse and keyboard interactions that trigger identifiable behaviors in vulnerable extensions. In more detail, we find that ≈67% of the extensions that require mouse or keyboard interactions do not check the `isTrusted` attribute (a read-only attribute generated by the browser which denotes whether an event originates from a user action) and can thus be triggered by the page. Moreover, our performance evaluation revealed that this attack can be efficient, as a page can fingerprint 20 extensions using artificially crafted events in less than 400ms. Due to the severe privacy implications of this attack, we develop a tool that can be used by extension developers to retroactively fortify their extensions against this attack. Specifically, this tool incorporates our static analysis techniques for identifying relevant event listeners (mouse and keyboard), and injects safeguarding code that checks the event's provenance and ignores events simulated by the webpage.

In summary, our research contributions are:

- We introduce a novel fingerprinting technique that offers the first exploration of extension fingerprinting in a real-world setting, where user actions can trigger unique extension behaviors. Accordingly, we conduct a systematic analysis of such behaviors in practice and develop a module for dynamically exercising and analyzing extensions.
- We conduct an extensive evaluation of user-triggered extension fingerprinting, and find that our approach can be effectively used in conjunction with other state-of-the-art fingerprinting techniques as it enables the detection of a significant number of previously-undetectable extensions.
- We demonstrate that extensions lack the necessary security checks to prevent web pages from issuing simulated user events that trigger their fingerprintable behaviors. As a countermeasure, we develop a straightforward-yet-effective tool for extension developers that automatically incorporates safeguards into their code.Our tool is available at [10].

2 Background and Threat Model

This section provides pertinent background information on browser extensions and technical characteristics that enable the techniques that we present in this work.

Extension structure and components. A browser extension is a set of different components, that implement the extension's functionalities and programmatic logic. The `Manifest` file plays a crucial role as it allows developers to specify background and content scripts, external pages, and permissions that enable extensions to achieve their desired functionality. Listing 1 shows a simplified example manifest file.

```
{ "manifest_version": 2,
  "background": {
    "scripts": ["my-backgrnd.js"]},
  "browser_action": {
    "default_icon": {
      "19": "button/button-19.png",
    },
    "default_title": "My title",
    "default_popup": "popup/popup_page.html"},
  "content_scripts": [
  { "matches": ["<all_urls>"],
    "js": ["content-script.js"]
  }],
  "permissions":
        ["activeTab","contextMenus","storage"]}
```

Listing 1: Simplified manifest example.

Background scripts. When a background script entry is included in the extension's manifest file, it is automatically recognized by the browser, and the script runs as an individual process. The background script contains HTML and JavaScript code that implements the extension's functionality. Usually, the extension's main logic is implemented in the background, which operates independently from the rest of the components. The background script communicates with the content script through the browser's `Messaging` API, where it can issue individual requests and create long-lived connections with the content script. Moreover, if the `tabs`

permission is defined in the manifest, the background script can directly inject a content script or a CSS context in the page using the `chrome.scripting.executeScript` and `tabs.insertCSS()` functions, respectively.

Browser action: Default popup. A browser action's popup is only shown when the user clicks on the extension's action button in the toolbar. The popup supports the typical HTML elements and structures a webpage would support and is automatically resized to fit its contents in the browser. The popup is only initially set under the `default_popup` property in the manifest, where its path is the relative path within the extension's directory. It also runs individually and communicates with the content script through the Messaging API. The popup page can also modify the website visually by injecting a CSS context programmatically using `tabs.insertCSS()`.

Content scripts are a crucial component since they are the only scripts that can be injected into the webpage. Essentially, extensions use content scripts to modify the webpage and communicate with the background script through the built-in APIs. Content scripts are typically declared statically in the manifest under the dedicated entry, or are programmatically injected. The manifest file can also define which domains the content script will execute on, either by explicitly listing them or defining a pattern that is matched to the visited domain. In more detail, content scripts use DOM requests to control the rendered page and can also inject custom event listeners in the page to listen for specific events. Listing 2, shows an example of a content script listening for specific user-driven events and then performing a series of DOM modifications. This provides flexibility to developers as it allows them to include additional extension functionality which can be triggered by various user behaviors.

```
1  //click event listener
2  element.addEventListener
       ("click" , function (event) {
3  //change the style of the element
4  element.style.color = "red";
5  });
6  //key event listener
7  document.addEventListener
       ("keydown" ,function (event) {
8   //check if the keycode matches
9   if (event.keyCode==65) {
10    //modify the page
11    document.style.color = "black";
12  }});
```

Listing 2: Mouse and key event listeners in a content script.

Permissions. An extension's ability to access websites and browser APIs is controlled through the "permissions"manifest entry. In general, permissions are restricted to those that the extension needs, and a subset of entries is shared between extensions. For example, the `contextMenu` allows the extension to include a context menu item (the menu that appears when the user right-clicks with the mouse) and to listen for these specific events in their content script. Finally the developer can also define the domain that an extension can run on (using `http://*/*`, `https://*/*`, or `<all_urls>`) if the content script is not present in the manifest.

Motivating example. Prior research has demonstrated various methods for fingerprinting extensions [29, 32, 44, 47, 48] and has explored the significant privacy risks they introduce. These techniques allow attackers to not only infer specific information about the user's browsing environment (which can be used to augment the user's browser fingerprint) but to also infer private and sensitive information about the user (e.g., health issues, religion, etc.). However, all prior studies overlooked the fact that many extensions are dynamic and reactive and may require user interactions prior to triggering their functionality. Since extensions may only modify the web page *after* receiving a specific user-driven event, extension fingerprinting frameworks that do not incorporate and systematically explore user actions are overlooking a core component of browser extension functionality.

This behavior is exemplified by the popular Chrome extension for Google Translate. When installed, a user can highlight a word on the page and the extension will automatically render a separate window on top of the page that includes the translation for different languages. The same functionality is triggered when the user highlights a term and fires the extension's context-menu item through the right-click menu. These behaviors are reflected in modifications and additions to the page's DOM, which would allow an attacker controlling the page to detect the changes and fingerprint the extension.

Threat model. We assume that the user visits a malicious or privacy-invasive web page that aims to infer which extensions the user has installed in their browser. Furthermore, we are interested in extensions that run on all domains and do not restrict their functionality to a specific set of domains, as these extensions can potentially be detected by any attacker. Additionally, we limit our focus to extension behaviors that interact with or modify web pages after being triggered by user actions (e.g., we do not explore Web Accessible Resources as they have been extensively explored in prior studies).

3 Methodology

Here we present our methodology for identifying extensions that exhibit fingerprintable behavior that is triggered by user interactions. Our approach consists of two phases: (i) a static analysis of extensions' source code and manifest files for identifying the types of interactions that can activate them, and (ii) a dynamic exercising phase that leverages our automation templates for simulating user interactions.

3.1 Preparatory Phase

We first analyze the extensions' manifest files to identify those that meet the requirements outlined in §2, indicating that they potentially expect user interactions. Subsequently, we statically analyze the extensions' source code so as to

identify the event listeners they implement and extract their arguments. This allows us to understand the types of events that extensions listen for. Based on the different types of events that we observe, we generate appropriate behavioral templates for automating the simulation of these interactions.

Manifest file. We are interested in extensions that (i) are fingerprintable due to modifications to or interactions with the visited page, and (ii) are not domain-specific (i.e., they run on all domains). Since dynamically exercising extensions is a time consuming process, we first parse the extension's manifest files and only select those that meet these criteria. This will allow us to speed-up experiments by avoiding the costly dynamic analysis phase for thousands of extensions which will not exhibit fingerprintable behavior on arbitrary sites.

Since we want to identify extensions that can access and modify a page's DOM, we search for extensions that include a `content-scripts` entry in their manifest. For such extensions the developer also has to include a `matches` entry in the manifest, specifying which domains the extension will run on. For extensions that are not domain-specific, the values typically used are "`<all_urls>`" and "`http://*, https://*`". Furthermore, as described in §2, extensions can dynamically execute a content script through their background scripts. To identify such extensions we parse their manifest files and select those that implement a background script and require the "`activeTabs`" and "`<all_urls>`" permissions. Subsequently, we statically analyze these background scripts for identifying the ones that use the `executeScript` and `insertCSS` APIs for dynamically running a content script or injecting a CSS file into the web page.

Categories of user interactions. Through the preliminary manual analysis of extensions we identified three general categories of potential user interactions; we categorize the different types of user actions as belonging to *browser*, *mouse* and *keyboard* actions. Next, we outline how we perform an initial selection of candidate extensions from each category through our manifest analysis.

Browser actions. The first category includes interactions that are initiated by the user when clicking on the extension's button (i.e. the extension's icon typically shown next to the browser's address bar). In the simplest case, a user will click on the extension button to activate it, which will result in the extension executing its intended functionality. Extensions can also include a *popup* page that is constructed by a separate HTML file and appears when the extension's button is clicked. The popup may provide an interface that allows the user to configure the extension, choose a mode of operation or alter its functionality. Additionally, the popup may also require the user to login, or even allow them to run specific functionalities directly (e.g., play a video, control the volume). Enabling this category of interactions requires that the extension has a background script and implements an event listener that captures click events on the extension's button. Furthermore, a `browser_action` entry needs to be included in the manifest. Extensions that implement a popup also need to define a `default_popup` in the manifest. As such, during the preparatory phase we can identify which extensions support interactions with their icon and popup, by parsing their manifest files and looking for the aforementioned entries.

Mouse actions. For this category, an extension can specify the `contextMenus` permission in its manifest to enable "right-click" interactions. When this permission is requested, the browser allows the extension to include additional entries in the context menu (i.e., the menu that appears in an overlay when pressing the mouse's right button). These newly included events are fired from the user's mouse and are processed by the respective extensions' content scripts.

Keyboard actions. To handle keyboard-driven user interactions the manifest can include a `commands` entry that defines one or more keyboard shortcuts expected by the extension. However, our initial exploration revealed that extensions do not always define these commands in the manifest; instead, it is more common to programmatically check for keyboard events by including the appropriate event listeners in their content scripts.

Static analysis. While analyzing the manifest files allows us to create an initial set of candidate extensions, this provides a limited view of extension's user-driven capabilities. In fact, extensions that leverage keyboard interactions are rarely evident from their manifest files. To uncover the user actions that can potentially trigger extensions we need to statically analyze extensions' content scripts. Specifically, we need to identify APIs and event handlers in the extensions' content scripts that expect events to be fired while the user interacts with the page. We build upon the methodology introduced by Somé [46] for detecting event listeners and extracting the events that they listen for. First, we use Python's `jsbeautifier` library to deobfuscate extensions' content scripts and obtain a more "human readable" form of their source code. Then we leverage Esprima [4] for parsing the content scripts' code and building their Abstract Syntax Trees (ASTs).

When the AST is created, we log the assignments to object properties and the function definitions and calls. This gives us detailed information regarding the type and value of each variable and function, which we use for locating the functions that expect events from the application (i.e., event listeners) and extracting their arguments. An event listener can exist as a standalone function or as a method for global objects and HTML elements, while there are also various ways that an event listener can be registered (e.g., `window.addEventListener`, `window['addEventListener']`). As such, we take into consideration all types of event listeners in each content script (i.e., for the global object names of `document`, `window`, `top`, `self`, `this`). Furthermore, the `addEventListener` API has two arguments: (i) the message, which denotes the actual event, and (ii) the function that is invoked when the event is fired. We are only interested in the first argument, which is a `Literal` specifying the type of the expected event.

Table 1: List of mouse and keyboard events compiled based on the findings of our static analysis.

Event	Action	Event	Action
Keyboard		**Mouse**	
Keydown Keyup Keypress	Key Press	Scroll Mousewheel Wheel	Scroll
Mouse		Cut Copy Paste ContextMenu	Right Click
Doubleclick Select	Doubleclick		
Click Mousedown Mouseup Blur Focus	Click	Mouseenter Mouseout Mousemove Mouseover	Movement

After identifying all the events expected by our extensions, we manually sorted through the list of expected events and determined which ones can be triggered via user interactions and which actions can generate these events. For a more complete and accurate mapping, we also cross-referenced our findings with official documentation [7, 8]. This was done once, after our preliminary analysis, and is a one-time cost as the generated list covers all relevant event listeners. Table 1 presents the list of interaction types that we compiled and the mapping between the various events and type of interactions (i.e., behaviors) that can trigger them. For instance, events like *mousedown*, *click*, *blur* and *focus* can all be triggered when the user clicks on the page and the included elements. In the following subsections we present how we design our *user interaction automation templates* that include actions that aim to trigger all the aforementioned events.

3.2 User Interaction Templates

The previous stage provides information about the extensions' structure (i.e., whether they include a clickable button, a popup page and a content menu) and the type of events they listen for. We leverage that information for designing and generating behavioral user interaction templates that reflect human-driven user actions. Each template includes various types of actions that correspond to coarse or fine-grained interaction activities, aiming to fire relevant events that can trigger extension functionality. Based on our aforementioned categorization, we define three general templates that encompass actions related to the browser, mouse and keyboard.

Browser actions. This template includes event-driven actions related to the browser interface. In the simplest case we have extensions that include a clickable browser button (i.e., they are activated when a user clicks on their icon). Upon activation these extensions might exhibit behavior that would allow us to detect their presence, such as altering the page or exchanging messages with the page; a popular extension that exhibits this behavior is `Mercury Reader`. As such, the simplest interaction that is defined in this template is to locate and click on the extensions' button. Next, extensions that include a `popup` page will typically include elements such as buttons and checkboxes, and provide an interface for the user to initialize, configure or control the extension's functionalities. Indicative examples are Ublock and Ghostery, where users interact with the popup pages to specify their preferences and enable/disable them. For such extensions, our template first defines the action of clicking on the extension's browser icon, so that the popup page will appear, and then it interacts with the page's element by clicking, selecting elements, activating buttons, and navigating its content.

Mouse actions. Moving beyond the browser's interface, we define a template that covers the user's interactions with the visited page through mouse actions. To that end, we leverage the findings from the static analysis of content scripts regarding events that are fired by actions associated with the mouse. In this template we model behaviors as sequences of mouse actions that can trigger the aforementioned mouse event listeners. In the simplest case, the *click* and *doubleclick* events are fired when the user clicks or doubleclicks the mouse, respectively. We also include the *mousedown* and *mouseup* events in the click category, since these two events are fired when the mouse button is pressed and released during a click. The *focus* and *blur* events are content-related and can also be triggered with a click action (e.g., the user clicks on a text input area to focus or blur its content). Furthermore, the *select* event is fired when text in the page is selected. Since text selection can also be achieved by doubleclicking text, a doubleclick action allows us to trigger both the *select* and *doubleclick* events.

In a similar way, we categorize all the mouse events that can be triggered when mouse movement is involved. Although events such as *mouseenter* and *mouseover* have differences in how they are fired, in the general case they are both fired at an element when the mouse cursor moves over that element (e.g., one difference is that *mouseover* is also fired when the cursor moves over the element's children nodes). The interactions in this template are designed to trigger all movement events. Finally, the *scroll*, *mousewheel*, and *wheel* events can be triggered by a mouse scroll using the mouse wheel as well as the browser's scroll buttons.

The last type of mouse event covers all the events that are related to a `context menu` and are fired when a right-click is involved. The browser offers the *cut*, *copy* and *paste* functionalities in the context menu, and an extension can include the respective event listeners to detect these actions. Finally, the *contextmenu* event is fired when the user clicks on the context menu entry set by an extension.

Keyboard actions. These actions focus on events that are triggered when the user presses a keyboard `key` during the page's navigation. Our static analysis process uncovered three event listeners defined in extensions' source code that are related to keyboard actions, all of which can be enabled by a single action, as pressing and releasing a key triggers all three events that they listen for. We define key actions that vary from single keystrokes to combinations of multiple keys.

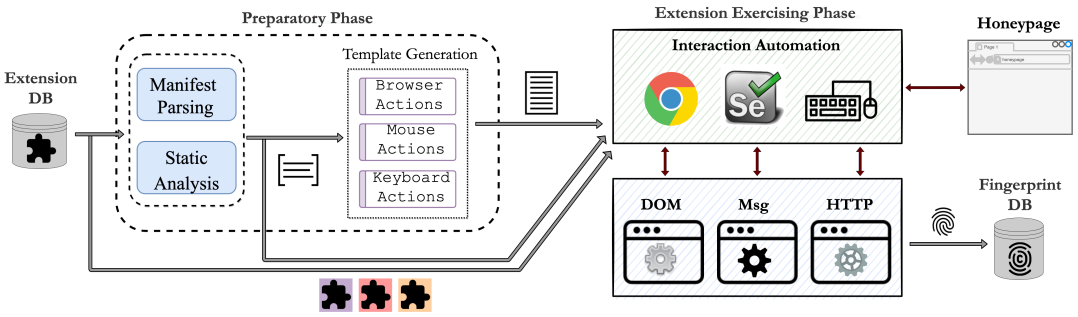

Figure 1: During the preparatory phase, our system analyzes each extension's manifest file and source code and extracts metadata that we use for designing the user interaction templates. During the exercising phase, our system leverages these user interaction templates to simulate realistic user actions, and creates behavioral signatures that enable fingerprinting user-triggered extensions.

4 Implementation Details

Here we present the implementation of our user-driven extension-fingerprinting system. A high-level overview of our system and analysis pipeline is provided in Figure 1.

Constructing a honeypage. Our goal is to trigger the highest possible number of extensions that expect user interactions. An extension that is triggered by such interactions may require the user to interact with specific types of elements on the page. Therefore, our testing framework needs to incorporate a `honeypage`, which we will visit and interact with when exercising each extension, and capture any modifications that occur. To that end, we leverage the code and dataset of Carnus [29], which was able to capture modifications from a large number of extensions, and build upon its honeypage.

We extend the honeypage by including various additional elements that we will interact with during the experimentation phase. Specifically, we include textual terms and phrases from the eight most used languages (i,e., English, Mandarin, Russian, Japanese, Hindi, German, Arabic and French). Since different languages share a subset of the same characters (e.g., English and Spanish), the terms we include may also trigger extensions that expect terms in a language that we have not explicitly included. Moreover, to satisfy all event listeners and behavior requirements, we also include a typical HTML form with username and email fields, and an input area where a user can potentially input text or paste information. Such elements are appropriate for triggering specific mouse events like `select` and `focus`, which are specifically designed for such elements. Moreover, our honeypage contains different `anchor` elements with link attributes, containing both inner domains and external 3rd-party pages. As we detail in the next subsection, we instruct the framework to interact with the above elements through a set of different user-simulated actions.

Applied user interactions. Having introduced the user interaction templates in §3, here we dig deeper into the framework design and detail the methods used to apply each template. We start by performing simple and straightforward actions that can trigger an extension on their own, and then move to more complex interactions that are composed using a sequence of actions. Furthermore, we distinguish actions that depend on the page's content (we refer to them as *targeted* actions) and those that are independent of the page and its elements (referred to as *generic* actions). For example, a click can be either targeted or generic – clicking somewhere on the page is a generic action while clicking on a term is a targeted action.

Browser interactions. First, we detail our process for generating actions based on the browser-event template.

Extension button. Our testing process starts by applying the most straightforward action for the browser interactions, i.e., the user clicks on the extension's icon. Without performing any other interactions, this mouse click is sufficient for triggering certain extensions. Specifically, when a user clicks on the extension's icon, the click event is captured by the respective event listener in the extension's background script (if there is such an event listener present), triggering the extension to run its intended functionality. For instance, this could result in the extension communicating with the page through its content script for injecting elements or modifying the page's code.

Next, we extend this simple action by including interactions with the page's content. In this scenario, the simulated user first selects a page element (i.e., highlights a term) and then clicks the extension's icon. In triggered extensions the selected value will be read by the content script and passed to the background script, which will perform any additional actions. For example, an extension that translates text would expect the user to select a word or a sentence on the page and then click the extension's icon, in which case a translation will be provided. We incorporate this type of interaction and behavior in our framework under the browser action category.

Extension popup page. We follow a similar approach for the external popup and option pages. In these extensions, once the user clicks the extension's icon, a separate HTML page will appear underneath the icon. We have empirically observed that developers typically design these popups to be visually simple and easily accessible so as to help users navigate. In our framework, once the simulated user clicks on the extension's icon and the external page loads, we focus

on the popup page and click all elements (e.g., radio buttons, checkboxes, and panels). Even without prior knowledge of the page's structure, we are able to interact with its elements and components. For completeness we also include a text-selection action in this template interaction i.e., we select a term in the page and then interact with the popup's elements.

As the popup page may include extension configuration options that either enable the extension or alter its default behavior, interacting with its elements can trigger the extension and lead to behavior that is observable by the page. The interactions in this template are sufficient for handling the vast majority of extensions. However, our template may not be able to handle complex popup pages that require additional user actions (e.g., installing other applications locally, registering and logging into an account). We also adopt the same set of interactions for `Options Pages`. The options page loads in a separate browser page when the user installs the extension and expect an initial configuration or modification of its current settings. We apply the same rules to initiate the page's behavior and log all the modifications that occur in the honeypage.

Mouse interactions. Next, we detail our process for generating actions based on the mouse-event template.

Clicks, doubleclicks & content selection. For the mouse actions template, we follow a building approach similar to that of the browser-event template. The first building block contains the simple left click (single or repetitive) that a user fires upon visiting the page. This action is generic since it does not interact with any page elements but fires an event to the page itself. While extensions that contain such event listeners are triggered by the fired mouse event, we have observed that extensions may also require `content-related` actions, including simple clicks or doubleclicks that select page elements and content. Following that principle, we incorporate the selection of page elements into this template's interactions. Since the extension's functionality could also rely on the language of the content, we include terms of various languages in our honeypage and emulate interactions with all these terms.

Copy, paste, scroll. The subsequent content-related actions include the context menu (right-click) actions provided by the browser interface (Copy, Cut, Paste) and the scrolling and wheel events that reflect the user's scrolling action. A `copy` or `cut` event is only available if the user selects a term on the page and then fires them through the context menu. We expand the previous set of actions, including the selection of a term followed by the copy and cut commands. Following those commands, the `paste` action is dependent on the previous activities; as such we instruct our framework to paste the copied content into an empty input area. We also replicate a user's behavior that copies information and pastes it into a specific empty area by activating and focusing on the input area. For completeness, we also trigger a selection event by highlight the content inside the input area, to trigger any additional event listeners. The last action that we include is scrolling; as before, we select a term on the page for completeness and perform the scrolling action, as a user would typically do. In practice, even if the term selection is not required by an extension's functionality it will not interfere with the scrolling action.

Context-menu items. In the last subset of mouse-related actions that our framework supports, we implement actions that trigger context menu items added by extensions. Similar to the left-click mouse events, the user might trigger the context-menu item through various actions. The framework replicates this behavior by triggering the extension element in different parts of the page. Specifically, at first, it fires the right-click on the page without specifying an element. Following the design principle of the previously implemented set of actions, it selects a term by highlighting it and then firing the same activity. Our framework also replicates similar context-related events by triggering the context-menu over a hyperlink of an anchor element present on the honeypage and an image element.

Keyboard interactions. Finally, we detail our process for generating actions based on the keyboard-event template.

Single, repetitive & combined keystrokes. Our framework adheres to a similar strategy for the keyboard event templates when simulating user interactions. The user will trigger a keyboard event directly on the page or after selecting and interacting with a page element. The framework performs the following actions to replicate this set of interactions: first, it sends a keyboard event directly on the page. Afterward, it selects and highlights a page element (term) and then sends the same key event again. Since we don't know which key event triggers the extension, we start by sending single actions for all the available keyboard characters and symbols (e.g., alphabet characters, numbers, and special characters). We have also observed that extensions may expect repetitive keyboard events used as a "special" combination of keys. For this, we expand the initial set of interactions, and also include repetitive keystrokes of the same character (e.g., an extension requires a repetitive keystroke of `b b` to get triggered). Moving a step further, we also include special keys (`ctrl, alt, ctrl-alt, ctrl-alt-shift`) combined with the aforementioned keyboard characters and numbers. In order for our system to not interfere with internal browser functionality we exclude shortcut key combinations already defined and allocated by the Chrome browser [2]. Our template is designed so as to exhaust all potential key interactions that a typical user could trigger, using this iterative process for creating keyboard events.

5 Experimental Setup

Interaction automation. Our framework for exercising extensions is driven by the Chrome browser, which we orchestrate using Selenium [11]. The most critical component of our framework is our User Interaction Automator, which leverages the PyAutoGUI module [9], a cross-platform GUI automation Python module that is used to programmatically control the mouse and keyboard. An important aspect of this module's functionality is that it uses the actual mouse

and keyboard devices and simulates actions similar to how a typical user would perform them. Additionally, since the honeypage and browser are under our control, we know *a priori* the position and size of each element and can replicate each action from the interaction templates by providing the `x-y coordinates` followed by the specific action. For example, successively calling `pyautogui.moveTo(100,500)` and `pyautogui.doubleClick()` will move the mouse to the specified coordinates and then perform a mouse doubleclick.

Using this approach we handle the majority of the browser, keyboard, and mouse interactions that we have defined, by providing the coordinates of each element that we want to include in our interaction and firing the respective events. We follow a different approach for browser interactions that result in a browser-external popup page; in such cases we rely on left mouse clicks, and `tab` and `spacebar` key events. We found that by combining these mouse and keyboard events we can successfully navigate the popup page without prior knowledge of its structure or content, changing the focus of elements, and selecting/enabling elements like buttons and radio boxes.

Fingerprint generation. To collect extensions' behavioral signatures we follow a similar approach to prior work [29]. We load each extension into the browser and visit the honeypage, wait for 15 seconds for the extension to initialize, load, and perform any initial modifications on the page, and then capture a snapshot of the page's state. This snapshot contains the page's `Outer HTML (DOM)`, the external resources loaded, and the messages broadcast by the extension to the page. We use the Performance API [12] to log any external resources fetched, and include a message event listener in the page (i.e., `document.addEventListener("message")`) for logging the messages that are broadcast. Finally, we store each snapshot into a separate `JSON` document for analysis.

After the initial snapshot extraction, we apply the appropriate interactions according to the entries in the extension's manifest file. For example, we start by applying the template for browser actions if a `browser_action` entry is defined in the manifest. If the extension has a popup page, we apply the template's interactions with the popup page. After that, we apply the templates that describe the mouse and keyboard interactions. These two templates are applied to all the extensions that we exercise. This allows our system to compensate for any event listeners missed during the static analysis of a given extension: even if we missed a listener for a specific type of events, our collection of actions curated from the static analysis of all the extensions will contain it.

After performing a given action, we wait for one second to allow for the extension to perform any modifications and our framework to capture them, before applying the next action. We compare the snapshot obtained after each interaction with the initial snapshot (i.e., the page's original DOM) and the one collected after the initial wait time. If any modification is detected, we store the current snapshot and kill the browser to remove any persistent modifications. When we finish exercising an extension with one of the three templates, we continue our process with the next template and repeat the aforementioned steps. When all templates have completed, we start a fresh browser instance to test a new extension.

Table 2: Number of extensions detected in each dataset.

Dataset	Extensions	Detectable(%)
D_1	27,342	2,932 (10.72%)
D_2	3,311	1,432 (43.24%)
D_3	9,446	1,167 (12.35%)
Total (all extension versions)		5,531
Total (unique extensions)		4,971

6 Experimental Evaluation

Here we assess our system's effectiveness at triggering and fingerprinting extensions through user-driven interactions.

Datasets. In our analysis, we use three different datasets:

- Dataset_1 (D_1): This includes the dataset used in the Carnus [29] study. Originally it contained 102,482 extensions – after applying our filtering rules (§3) we are left with 27,342 extensions.
- Dataset_2 (D_2): Includes the *detected* extensions from Fingerprinting In Style [32]. Originally this dataset contained 4,446 extensions. To avoid overlap, after our filtering we also removed extensions with identical versions included in D_1. We ended up with 3,311 extensions, which also includes extensions with different versions to D_1.
- Dataset_3 (D_3): In May and June of 2021, we conducted a crawl of Chrome's Web Store to collect a more recent snapshot of the store. After applying our filtering methods we ended up with 9,446 extensions, from which 2,736 are newer versions of the extensions included in the other two datasets, while the remaining 6,710 are new extensions.

We will interchangeably refer to the datasets with their identifiers and system or study name for the rest of our paper.

System setup. Prior to performing our experiments we first deployed our honeysite on a popular web service hosting environment. For our framework, we used two identical off-the-shelf desktop machines with a 6-core Intel Core i7-8700, 32GB of RAM, connected to our university's network. The PyAutoGui library [9] requires a connected monitor to perform any interactions; to bypass that limitations we modified our framework and built it into a Docker Container [1]. To reduce potential browser-configuration failures (e.g., an extension malfunctioning on a new browser version due to updated APIs), for each dataset we used a browser version contemporary to that dataset [3] (versions: 73.0.3683.68, 83.0.4103.39, and 92.0.4515.43).

Overview. Table 2 lists the number of detected extensions per dataset. For the oldest dataset (D_1), our framework detects ≈11% of the extensions. Interestingly, for D_2 the detection percentage is significantly higher at 43%. This

Table 3: Detectable extensions per behavioral template.

	Browser Actions			Mouse Actions			Keyboard Actions		
	DOM	MSG	Total	DOM	MSG	Total	DOM	MSG	Total
D_1	2,846	70	2,886	646	15	661	704	6	710
D_2	868	29	895	506	6	512	634	-	634
D_3	1,096	79	1,175	321	22	341	432	6	438

dataset is formed of extensions that inject CSS into the page; by leveraging user interactions, we trigger the injection or an interaction with already injected elements. Finally, in our most recent dataset, we detect 1,167 (12.35%) extensions, which is similar to the detection rate for D_1.

To gain insights about the different types of interactions and behaviors, in Table 3 we breakdown the different templates and detection methods. As detailed in §5, each fingerprint contains DOM modifications and/or internal browser communication. However, our system did not trigger any instances of external communication; this is expected since extensions load necessary resources when installed or at run-time. In regards to DOM modifications, the browser actions have the highest detection rate demonstrating our framework's ability to simulate interactions expected by extensions. Similarly, both keyboard and mouse events trigger a large number of extensions. This supports our initial motivation, as extensions often offer on-demand functionality that is explicitly triggered only once users interact with them.

On the message-modification front, fingerprintable extensions are significantly fewer than the other categories. Only a small fraction require complex communications between privileged and unprivileged extension components, resulting in only a few extensions being fingerprintable through message exchanges. Upon analyzing the messages exchanged between extensions and the page, we find that most include actions that either initialize a DOM modification (e.g., `showPopup`, `dictionary_window:1`) or include the type of interaction (e.g., `x:10,y:24`) required by the extension's functionality.

Modality. Extensions can be fingerprinted through multiple types of interaction. We found that 80% require one type of interaction, whereas 15% can be fingerprinted through two different templates. The remaining 5% can be fingerprinted by actions from all three behavioral templates.

6.1 Behavioral Templates

Browser actions. 53% of the extensions detected by browser actions, across all datasets, are triggered by simply clicking on the extension's button. Moreover, 15.6% are triggered through interactions with the extension's popup page. This demonstrates the importance of statically analyzing extensions' `manifests` and not limiting our analysis to event listeners.

Mouse actions. A detailed breakdown of the interactions specified in the mouse actions template is presented in Table 4. We find that the page's language can be an important factor, since several extensions are only triggered when a specific language is present. Language-specific behavior is common

Table 4: Unique set of extensions triggered per mouse action.

Mouse Action	D_1	D_2	D_3
Click/Doubleclick Page	4	6	4
Select English Term	20	9	4
Select Non-English Term	5	7	3
Copy-Paste-Scroll	8	5	6
Select Page Element	331	274	189
Right-Click Page	114	108	70
Right-Click Term/Link/Image	28	18	9
Right-Click Page Element	151	85	56

for extensions that offer, among others, dictionary-related and translation-related functionality. At the same time, the majority of extensions do not include such specializations and are triggered whenever a user selects an arbitrary word or DOM element. This behavior is consistent in all three datasets, with the generic term selection fingerprinting the largest number of extensions for "left-click" actions (90% on average).

We observe similar behavior for the context-menu functionality (three bottom rows of Table 4), where several extensions are triggered only by selecting the appropriate context-menu item without specifying any term or element on the page. This reflects extension functionality that modifies the visited page without any restrictions on its content. Nonetheless, 9.5%, 8.5%, and 6.6% of the extensions from the three datasets, respectively, require a specific element to be selected on the page (e.g., a term, link, or image) to be coupled with the context-menu action. These are extensions whose functionality is related to selected elements, and thus are not triggered in any other way. In general, our experimental results confirm our framework's ability to fingerprint extensions that require both simple as well as complex chains of user interactions.

Keyboard actions. Figure 2 shows the distribution of different types of key events that trigger extensions. The Hotkeys types `1,2,3` denote a combination of a key-character with one, two, or three special keys (i.e., `ctrl, alt, shift`). Our results show that single keystrokes and Hotkeys-2 have a high frequency of occurrences across all datasets, indicating that developers prefer the adoption of simple key shortcuts over more complicated combinations that users are likelier to mistype or forget. However, we detected an instance of an extension that employs 7 different single keystrokes and Hotkeys to provide users functionality. Finally, we also found extensions that rely on complex triggering using 3 Hotkeys (`ctrl-alt-shift-<character>`). Interestingly, the majority are not triggered by actions from the other templates.

Comparison to prior work. Prior work has explored different ways of detecting browser extensions, using behavioral modifications [29] and style modifications [32]. To better understand the capabilities of our newly introduced technique, we compare our detected extensions with the two previous methods. When comparing with Carnus [29], we only use the behavior-based detections (i.e., DOM, inter and intra communications); we do not include WAR-based detec-

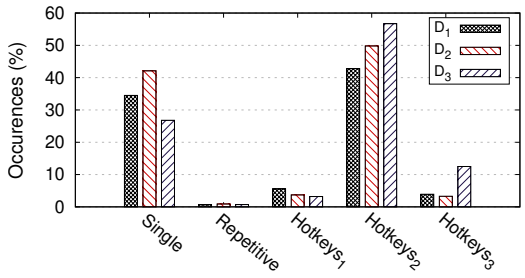

Figure 2: Types of keyboard events that trigger extensions.

tions in our comparison, since Firefox already defends against them [24] and Chrome recently introduced a new access-control mechanism for limiting the exposure of resources to specific pages [16]. Since we did not have access to the complete dataset of Fingerprinting in Style [32] for our experiment, we follow the authors' approach and compute the upper bound of the *potentially* fingerprintable extensions. Specifically, for each dataset we count the number of extensions that inject CSS in pages, as denoted in their manifests. An extension that does not inject CSS rules cannot, by definition, be fingerprinted via custom CSS properties. For the rest of our analysis we will use these subsets for any additional comparisons.

We are able to detect 2,932 extensions (2.8%) from the entire D_1 dataset compared to 6,381(6.2%) detected by Carnus, and 7,048 (6.8%) that could *potentially* be detected by Fingerprinting In Style due to injected CSS. However, 64% of our detected extensions are "invisible" to Carnus, and 63% to Fingerprinting in Style, while 45% are not detectable by any of these methods. Similarly, we compare the detection for the D_3 dataset, where we fingerprint 1,167 extensions (12.35%) while Fingerprinting In Style can detect at most 2,933 (31%); again, 45% of these extensions are only detected by our framework. It is worth noting that the extensions that are only fingerprintable by our system are highly dynamic and modify the page only after user interaction. The other methods only detect extensions modifications passively by observing the DOM and, thus, these dynamic extensions are invisible to them.

In total, we are able to uniquely fingerprint 1,820 unique extensions in datasets D_1 and D_3 that any of the approaches would miss. Overall, our results demonstrate that our newly proposed user-interaction-based fingerprinting technique is a powerful addition to existing techniques as it significantly expands coverage for previously-undetectable extensions.

6.2 Popularity & Longitudinal Analysis

Detected extensions types, prevalence & popularity. In order to classify the fingerprintable extensions, we categorize them based on their type as provided by the extension store. For each dataset, the most popular category is "Productivity", which is expected since different extensions fall under this category (e.g., translation and navigation functionalities). A detailed overview of the extensions' categories and popularity can be found in the Appendix A.

To gain more insight, we also calculate their relative popularity based on the number of installations. Specifically, we calculate the popularity for the 2,932 detected extensions of D_1 and compare it with those fingerprinted by Carnus, by Fingerprinting in Style, and extensions not detected by any method. The extensions detected by our method have been installed by 11,048 users on average, while for Carnus and Fingerprinting in Style the popularity is 6,775 and 9,462 respectively. For the remaining undetected extensions, their average number of downloads is 7,133. While this supports prior findings by Karami et al. [29] that popular extensions are likelier to offer more functionality (which can lead to being fingerprintable), it also indicates that more popular extensions are also more likely to include dynamic and customizable functionality that is triggered through user interactions.

Versions. Our most recent dataset (D_3) contains 2,736 extensions with newer versions of extensions included in the older datasets D_1 and D_2. Of those, $\approx 9\%$ are detected across all datasets, i.e., remained fingerprintable over the span of multiple years. Moreover, 5% were not detectable in the older datasets (i.e., became fingerprintable in more recent versions), and 6% were only detectable in older datasets (i.e., stopped being fingerprintable). This is due to extensions modifying their intended behavior or aspects of their functionality. We manually inspected 50 randomly selected extensions, and found that 32 either modified their source code or specified the "permissions" or "externaly_connectable" entries in their manifest so as to only run on specific domains. Also, 14 extensions offer the same functionality but without modifying the DOM (e.g., using the browser's popup window). Finally, four extensions offer completely different functionality and changed their behavior in the most recent version. In general, whenever an extension updates, the fingerprinting-derived signatures for that extension may also need to be updated. This is true for the attacks presented in this paper as well as for all prior techniques (web-accessible resources, DOM modifications, etc.) that use some form of a side-channel to infer the presence of an extension.

It is worth noting that 15% of the newly detected extensions belong to the `Accessibility` category, which could potentially allow the inference of sensitive user characteristics. Our results indicate that an extension's fingerprintability is fairly stable over time and only a small number of extensions modify their functionality across versions in a way that affects that aspect of their behavior.

6.3 System and Attack Performance

Dynamic analysis. In Figure 3 we present the total time in seconds required by each template in our framework when dynamically exercising an extension. The mouse and browser actions templates require the lowest number of interactions,

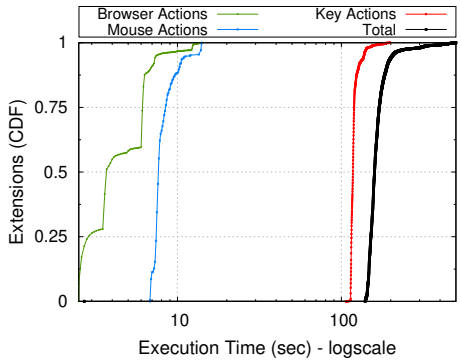

Figure 3: Performance for the different interaction templates.

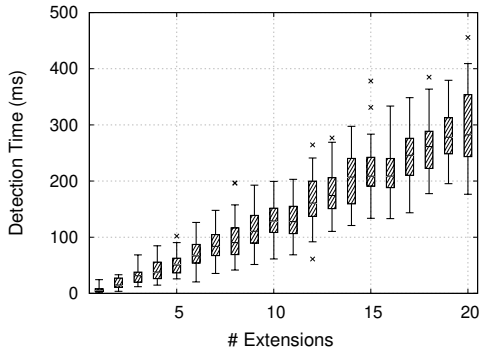

Figure 4: Detecting different subsets of installed extensions.

which is reflected in their execution times: for 90% of the extensions, triggering events can be dispatched and evaluated in less than 10 seconds. A longer execution time is expected for the key events since we need to trigger multiple keys which leads to a significantly larger number of potential key combinations. For the majority of extensions, keyboard interactions require approximately 2 minutes. Overall, our framework requires less than 200 seconds to complete testing the interactions of all three templates against an extension. Note that this is a one-time cost which only needs to be repeated whenever an extension is updated. The increased overhead for ≈5% of the extensions is the result of system's overhead due to the parallelization of docker containers, browser overhead, and system I/O operations. In summary, our system's performance is suitable for large-scale extension analysis, with multiple opportunities for further optimization via additional parallelization and the data-driven removal of events that rarely lead to DOM changes (e.g., the removal of keyboard combinations that did not trigger any extensions in our experiments).

Attack: Page-simulated events. In our analysis we detailed the different types of interactions and user behaviors that result in the successful triggering of extensions and their subsequent fingerprinting. Here, we draw attention to the fact that `mouse events` and `keyboard events` can also be simulated by the page (obviously, we cannot simulate the right click functionality of the context-menu item from the mouse actions template since this is a browser-controlled interface). More specifically, left-click and keyboard interactions (all key combinations including the `copy` and `paste` functionality) can be simulated by specifically crafted events that replicate user interaction. The JavaScript framework of `Dispatch Event` can be used to initialize different types of events that are targeted to specific event listeners [6]. For example, a `click` event is created and dispatched (fired) on a specific page element after its call. Using this API, one can craft artificial events that replicate user interaction to trigger extensions without the user actually interacting with the page. In practice, we can include various simulated events in our honeypage and attempt to trigger a specific subset of extensions requiring such interactions.

Since events that are typically initiated by users can also be dispatched via JavaScript, browser vendors have included a special property in the `Event` interface that can be used for verifying the provenance of an event. Specifically, each event carries with it a read-only, `isTrusted` property [5], indicating whether the event resulted from a user action or whether it was dispatched through JavaScript. The same property is also available through jQuery's `original Event` function and similarly distinguishes user events from script events.

Extension vulnerability. We perform the following experiment to identify the extensions that a page can trigger through simulated actions. First, we include all the event listeners related to the appropriate mouse and keyboard events in our honeypage (i.e., events shown in Table 1). After that, we visit our honeypage with the extensions found to be triggered by mouse or keyboard actions, perform again the actions that have previously triggered each one of them, and log all the events captured by the event listeners. Since a user action may generate multiple events, which activate different event listeners, we need to artificially trigger and dispatch all these events when simulating the user interaction through the page. For instance, the user action of clicking the mouse button fires the `click`, `mousedown` and `mouseup` events. While some extensions may be triggered by one of these events, others may be get triggered by a different one. As such, for us to accurately simulate user actions through the page's JavaScript, we captured how users' actions trigger all relevant events.

Finally, after identifying all the events that correspond to the actions triggering each one of the extensions, we modify our honeypage to dispatch these events automatically from within the page. We visit the modified version of the honeysite with a browser that has the aforementioned extensions installed, and check whether the events dispatched from the page trigger the extensions' functionality.

From the 2,234 extensions that are triggered by actions that can be simulated through JavaScript, we successfully trigger 1,513 (67%). Specifically, 88% of the extensions that require

mouse interactions and 65% of those requiring keyboard interactions were triggered successfully. As expected, the percentage is higher for the mouse events since the trusted flag is more commonly used for key events. Our results demonstrate that, for the majority of extensions, invasive pages can simulate user actions and deterministically identify the corresponding extensions without depending on users' behavior. Finally, a detailed overview of the vulnerable extensions' categories be found in the Appendix A.

Attack performance. To assess our attack feasibility in a realistic scenario, we measure the time that a page needs for detecting $N = 1,...,20$ extensions. Due to the variance of triggers across extensions, we randomly select extensions that leverage different types of interactions (mouse and keyboard). We include a script that performs the needed type of interactions in the page, which starts executing after the browser fires the `window.onload` event. The fingerprinting script then fires a user-action-simulating event and waits until there is a DOM modification before proceeding to the next event, while logging all times corresponding to these events. We used the `Performance API` to measure the time difference, with the starting point being before calling the dispatch function and the end being after the comparison between the DOM snapshots. Since we use different subsets of extensions, we measure the total time required to detect each subset of extensions and report it accordingly. Moreover, to collect a representative set of measurements, we repeat this experiment 50 times.

Figure 4 shows the results for all sets of installed extensions. As expected, there is a positive relationship between the number of installed extensions and the time required to fire all the appropriate user-simulating events and detecting the corresponding DOM changes. Even in the extreme case of a user having installed 20 extensions (Starov and Nikiforakis reported that the average user installs 4.8 extensions [48]), the entirety of the action-triggering and fingerprinting process takes less than 0.5 seconds. As we showed in §6.1, 90% of the mouse-triggered extensions require a generic term selection, while 88% of the keyboard-triggered extensions require either single keystrokes or a combination of two special keys.

In a real-world deployment, the attacker does not need to simulate all the available interactions since many of them do not trigger any extension, and a page would include a substantially larger number of emulated events that target as many extensions as possible. In our performance evaluation where we leverage unique combinations of events, a single combination triggers one extension in less than 6 ms. Subsequently, to trigger all the combinations of the 1,513 extensions that we detect through page-simulated events, the page would require less than 40 seconds for firing the events and detecting the modifications. This is practical since it is lower than the average time that users spend on a page (62 seconds) [17]. Finally, an attacker can apply different strategies to optimize the detection process and significantly reduce detection time (e.g., sending the most common events first or only targeting specific extensions of interest).

Attack Stealthiness. We need to consider two scenarios: (i) users' organic actions, and (ii) the page simulating user interactions. In the first case, our technique is completely stealthy as the interactions are performed by the user and we only detect the resulting changes. In the case of simulated interactions, keyboard events are invisible since there is no visual effect on the page (thus, matching the stealthiness of prior techniques). For mouse interactions, some are invisible (e.g., clicking) while others have a small visual effect (e.g., text highlighting). Additionally, attackers can employ techniques like tab-nabbing [19], to detect that the user has moved focus to a different tab before simulating these events, in which case the user would not witness the simulated mouse events. A demonstration of our attack is available at [10].

7 Countermeasure and Discussion

Here we present our defense and further discuss our attack.

Countermeasure. We develop a tool for extension developers that allows them to retroactively fortify their extensions against pages that simulate user actions. Our tool introduces appropriate safeguards in the extension's code without affecting its functionality or the user's browsing experience.

Specifically, we build upon our static analysis tool (§3.1) and the list of event listeners that can be misused by pages (§6.3), and create an extensive list of all mouse and keyboard event listeners. Given the extension's content-script source code, we inject a function at the beginning of the source file that will be executed first. Our function overrides the `addEventListener` function located in the prototype of the `EventTarget` interface. Listing 3 in the Appendix B provides an example of our strategy. We first check if the argument on the addEventListener is one of the mouse or key events; if we detect such an argument we subsequently verify the origin of the event and reject events that are not generated by users. If no such event is detected, the event listener is not affected and execution proceeds as expected. We manually verified that our approach works correctly on 50 randomly selected extensions by correctly handling both user-generated and page-simulated events without functionality being affected.

Extension obfuscation. A limitation of our static analysis process (§3.1) is that in cases of heavily obfuscated scripts that employ sophisticated obfuscation and minification techniques, it might generate incomplete ASTs. However, this does not ultimately affect our attack's effectiveness, as during our exercising process every extension is tested against all mouse and keyboard action templates. These templates were generated based on the results of the static analysis process as well as the corresponding developer documentation for completeness. As such, our dynamic extension exercising provides a comprehensive assessment and is not affected by issues during the generation of a given extension's AST.

8 Related Work

Users' increasing demand for online privacy, which resulted in significant efforts by the community and browser vendors for preventing cookie-based tracking, has also led to the emergence of stateless tracking and browser fingerprinting techniques. A large body of prior work has explored various aspects of browser fingerprinting and demonstrated the feasibility of such techniques [14, 15, 18, 20, 23, 25, 30, 31, 37–39, 52].

More recently, extension fingerprinting has caught the attention of the research community as a new fingerprinting vector. Over the last few years, several works have explored extension fingerprinting, proposed various extension enumeration techniques and countermeasures, and demonstrated how the users' list of installed extensions can enable the inference of sensitive user information [26, 29, 32, 43, 44, 47, 48, 50, 51].

In one of the first works in the area of extension fingerprinting, Sjösten et al. [44] demonstrated how websites can detect the presence of extensions in the user's browser based on the Web Accessible Resources (WARs) that these expose. Gulyas et al. [26] used the WAR-based technique from [44] and conducted a large-scale study on the uniqueness of users that visited their website. They found that they can uniquely identify 54.86% of the users that have at least one extension installed. In a different line of work, Sanchez-Rola et al. [43], as well as Van Goethem and Joosen [51], proposed a timing-based side-channel attack for detecting the presence of extensions. Specifically, they issue a request for accessing an extension's non-existent resource and measure the time that it takes for the browser to respond. The response takes longer in the case where the extension is present, as the browser first parses the manifest to determine if the resource is accessible.

The works that are most closely related to ours are those that explore behavior-based extension fingerprinting. In the first study in this area, Starov and Nikiforakis [48] showed that extensions can be detected based on the DOM modifications that they perform to the visited page. Furthermore, by surveying 854 users, they also found that many users tend to install unique sets of extensions, thus becoming uniquely identifiable. Karami et al. [29] developed `Carnus`, a framework that employs both static and dynamic analysis for the generation of extensions' behavioral-based fingerprinting signatures in an automated fashion. Moreover, they explored how the detection of extensions can lead to the inference of sensitive information (e.g., ethnicity, religion).

Trickel et al. [50] proposed `CloakX`, a defense that diversifies the extensions' behavioral fingerprints to prevent detectability. More specifically, it substitutes the injected DOM elements' identifiers and class names, while also inserting random tags in the page as noise. However, this approach cannot prevent detectability for the majority of extensions that are fingerprinted by `Carnus` [29]. In another work, Starov et al. [47] investigated whether the extensions' behavior and page modifications, that in turn make these extensions fingerprintable, are needed for their intended functionality. Similarly to Karami et al.'s [29], this work also accounts for extensions that are fingerprintable due to the messages they exchange. Finally, Laperdrix et al. [32] has recently proposed an extension fingerprinting technique that detects extensions based on the style sheets that these inject in the visited page. Using this technique, the authors of [32] were able to uniquely identify 4,446 extensions, from which 1,074 (24%) have not been fingerprinted by any previously proposed techniques.

All prior work only considered behaviors that extensions exhibit automatically and by *default* did not take into account the dynamic of user interactions. To the best of our knowledge, our work is the first that incorporates user interactions and attempts to *actively* trigger extensions' functionalities, aiming to make them exhibit fingerprintable behaviors.

9 Conclusion

More than a decade has passed since the seminal works of Mayer [36] and Eckersley [20], and yet browser fingerprinting remains an open problem. The fingerprinting of browser extensions is particularly concerning since, in addition to offering bits of entropy, they also reveal sensitive personal and socioeconomic characteristics of the users who *chose* to install them. In this paper, we drew attention to a limitation that has been common to all prior research on the fingerprinting of browser extensions. Namely, we showed that prior work has ignored the aspect of users interacting with browser extensions and how these interactions can be abused to fingerprint extensions. Through the use of static and dynamic analyses, we were able to take advantage of user interactions to fingerprint 4,971 extensions, including more than a thousand extensions that remained invisible to prior fingerprinting methods. Moreover, we demonstrated that due to developer error, the majority (67%) of extensions that are triggered by mouse or keyboard events can be fingerprinted via artificial user actions that the page itself can generate, as opposed to requiring a user's unwitting help. Finally, to at least partially ameliorate this common developer mistake, we proposed a tool that can add appropriate event-provenance checks wherever they are missing. We hope that future research into browser fingerprinting will take user-interactions into account, both in terms of an attacker's capabilities, as well as in proposed countermeasures.

Acknowledgements: We would like to thank the anonymous reviewers and our shepherd Anupam Das for their valuable feedback. This work was supported by the National Science Foundation under grants CNS-1934597, CNS-1941617, and CNS-2126654 as well as the office of Naval Research under grant N00014-20-1-2720. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the NSF or the ONR.

References

[1] Accelerate how you build, share, and run modern applications. https://www.docker.com/.

[2] Chrome keyboard shortcuts. https://support.google.com/chrome/answer/157179?hl=en&co=GENIE.Platform%3DDesktop#zippy=.

[3] ChromeDriver - WebDriver for Chrome. hhttps://chromedriver.chromium.org/downloads.

[4] Esprima - ECMAScript parsing infrastructure for multi-purpose analysis. https://esprima.org/.

[5] Event: isTrusted . https://developer.mozilla.org/en-US/docs/Web/API/Event/isTrusted.

[6] EventTarget : dispatchEvent. https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/dispatchEvent.

[7] Keyboard events. https://developer.mozilla.org/en-US/docs/Web/API/Element#keyboard_events.

[8] Mouse events. https://developer.mozilla.org/en-US/docs/Web/API/Element#mouse_events.

[9] PyAutoGUI : cross-platform GUI automation Python module. https://pyautogui.readthedocs.io/en/latest/#.

[10] Repository for the artifacts and defense mechanism of our attack. https://github.com/kostassolo/dangers-of-human-touch.

[11] Selenium is a suite of tools for automating web browsers. https://www.selenium.dev/.

[12] Using the Performance API. https://developer.mozilla.org/en-US/docs/Web/API/Performance_API/Using_the_Performance_API.

[13] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 674–689, 2014.

[14] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. Fpdetective: dusting the web for fingerprinters. In *Proceedings of the ACM SIGSAC conference on Computer & communications security*, pages 1129–1140, 2013.

[15] Yinzhi Cao, Song Li, and Erik Wijmans. ((cross))-browser fingerprinting via os and hardware level features. In *NDSS*, 2017.

[16] Chrome. Manifest - web accessible resources. https://developer.chrome.com/docs/extensions/mv3/manifest/web_accessible_resources/.

[17] Contentsquare. 2020 digital experience benchmark.behavioral benchmarks based on 7bn user sessions to help you beat kpis and win at digital experience. Technical report, Technical report. Available at: https://go.contentsquare.com/hubfs/eBooks/20202020.

[18] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. The web's sixth sense: A study of scripts accessing smartphone sensors. In *Proceedings of ACM CCS*, 2018.

[19] Philippe De Ryck, Nick Nikiforakis, Lieven Desmet, and Wouter Joosen. Tabshots: Client-side detection of tabnabbing attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 447–456, 2013.

[20] Peter Eckersley. How unique is your web browser? In *Proceedings of the 10th International Conference on Privacy Enhancing Technologies*, 2010.

[21] Steven Englehardt and Arthur Edelstein. Firefox 85 Cracks Down on Supercookies. https://blog.mozilla.org/security/2021/01/26/supercookie-protections/, 2021.

[22] Steven Englehardt et al. Automated discovery of privacy violations on the web. 2018.

[23] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of ACM CCS*, 2016.

[24] Firefox. web_accessible_resources. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/web_accessible_resources.

[25] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale. In *Proceedings of the world wide web conference*, pages 309–318, 2018.

[26] Gabor Gyorgy Gulyas, Doliere Francis Somé, Nataliia Bielova, and Claude Castelluccia. To extend or not to extend: on the uniqueness of browser extensions and web logins. In *Proceedings of the Workshop on Privacy in the Electronic Society*, pages 14–27. ACM, 2018.

[27] Artur Janc and Lukasz Olejnik. Web browser history detection as a real-world privacy threat. In *European Symposium on Research in Computer Security*, pages 215–231. Springer, 2010.

[28] Soroush Karami, Panagiotis Ilia, and Jason Polakis. Awakening the web's sleeper agents: Misusing service workers for privacy leakage. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2021.

[29] Soroush Karami, Panagiotis Ilia, Konstantinos Solomos, and Jason Polakis. Carnus: Exploring the privacy threats of browser extension fingerprinting. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2020.

[30] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. Browser fingerprinting: A survey. *ACM Transactions on the Web (TWEB)*, 14(2):1–33, 2020.

[31] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *IEEE Symposium on Security and Privacy (SP)*, pages 878–894. IEEE, 2016.

[32] Pierre Laperdrix, Oleksii Starov, Quan Chen, Alexandros Kapravelos, and Nick Nikiforakis. Fingerprinting in style: Detecting browser extensions via injected style sheets. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[33] Sangho Lee, Hyungsub Kim, and Jong Kim. Identifying cross-origin resource status using application cache. In *NDSS*, 2015.

[34] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *25th USENIX Security Symposium (USENIX Security)*, 2016.

[35] Xu Lin, Panagiotis Ilia, and Jason Polakis. Fill in the blanks: Empirical analysis of the privacy threats of browser form autofill. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[36] Jonathan R Mayer. "any person... a pamphleteer": Internet anonymity in the age of web 2.0. *Undergraduate Senior Thesis, Princeton University*, page 85, 2009.

[37] Vikas Mishra, Pierre Laperdrix, Antoine Vastel, Walter Rudametkin, Romain Rouvoy, and Martin Lopatka. Don't count me out: On the relevance of ip address in the tracking ecosystem. In *Proceedings of The Web Conference*, pages 808–815, 2020.

[38] Keaton Mowery and Hovav Shacham. Pixel perfect: Fingerprinting canvas in html5. pages 1–12, 2012.

[39] Martin Mulazzani, Philipp Reschl, Markus Huber, Manuel Leithner, Sebastian Schrittwieser, Edgar Weippl, and FC Wien. Fast and reliable browser identification with javascript engine fingerprinting. In *Web 2.0 Workshop on Security and Privacy (W2SP)*, volume 5, 2013.

[40] Lily Hay Newman. Wired - the fractured future of browser privacy, 2020. https://www.wired.com/story/chrome-firefox-edge-browser-privacy/.

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

[42] Valentino Rizzo, Stefano Traverso, and Marco Mellia. Unveiling web fingerprinting in the wild via code mining and machine learning. *Proceedings on Privacy Enhancing Technologies*, (1):43–63, 2021.

[43] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies. In *Proceedings of the 26rd USENIX Security Symposium (USENIX Security)*, 2017.

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

[45] Konstantinos Solomos, John Kristoff, Chris Kanich, and Jason Polakis. Tales of favicons and caches: Persistent tracking in modern browsers. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2021.

[46] Doliere Francis Somé. Empoweb: empowering web applications with browser extensions. In *Symposium on Security and Privacy (SP)*, pages 227–245. IEEE, 2019.

[47] Oleksii Starov, Pierre Laperdrix, Alexandros Kapravelos, and Nick Nikiforakis. Unnecessarily identifiable: Quantifying the fingerprintability of browser extensions due to bloat. In *The World Wide Web Conference*, pages 3244–3250, 2019.

[48] Oleksii Starov and Nick Nikiforakis. Xhound: Quantifying the fingerprintability of browser extensions. In *IEEE Symposium on Security and Privacy (SP)*, pages 941–956. IEEE, 2017.

[49] David Temkin. Google Ads - Charting a course towards a more privacy-first web. https://blog.google/products/ads-commerce/a-more-privacy-first-web/, 2021.

[50] Erik Trickel, Oleksii Starov, Alexandros Kapravelos, Nick Nikiforakis, and Adam Doupé. Everyone is different: Client-side diversification for defending against extension fingerprinting. In *28th {USENIX} Security Symposium ({USENIX} Security)*, pages 1679–1696, 2019.

[51] Tom Van Goethem and Wouter Joosen. One side-channel to bring them all and in the darkness bind them: Associating isolated browsing sessions. In *11th {USENIX} Workshop on Offensive Technologies ({WOOT})*, 2017.

[52] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. Fp-stalker: Tracking browser fingerprint evolutions. In *IEEE Symposium on Security and Privacy (SP)*, pages 728–741. IEEE, 2018.

[53] John Wilander. WebKit - Intelligent Tracking Prevention (ITP). https://webkit.org/blog/9521/intelligent-tracking-prevention-2-3/, 2019.

A Appendix: Extension Statistics

Here we present additional details and statistics about the extensions detected by our system.

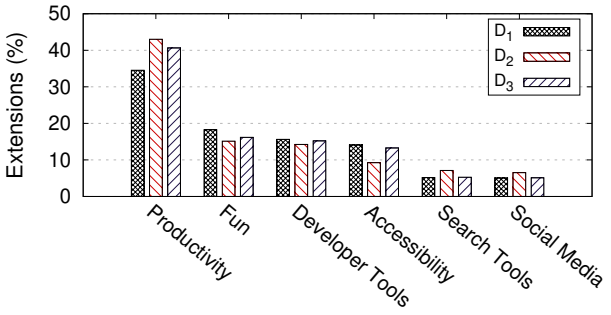

Figure 5: Categories of extensions for the corresponding datasets.

In Figure 5 we present the main category types of the detected extensions. The most popular category is that of "Productivity" with ≈ 40% of the extensions of each dataset. The next most popular category is "Fun" with ≈ 15% of the extensions. Also, ≈15% of the extensions are categorized as "Developer Tools" and "Accessibility".

Figure 6 reports the total number of installations for the extensions of the three datasets in our analysis. As can be seen, 50% of the extensions of the D_1 and D_2 have at least 100 downloads, while half of the extensions of D_3 have approximately 1,000 downloads. Moreover, 10% of the extensions of all datasets are installed by 10,000 users, and the most popular extensions have over 2 Million users.

Figure 7 reports the category types of the extensions that are fingerprintable by our techniques. Similarly to the overall

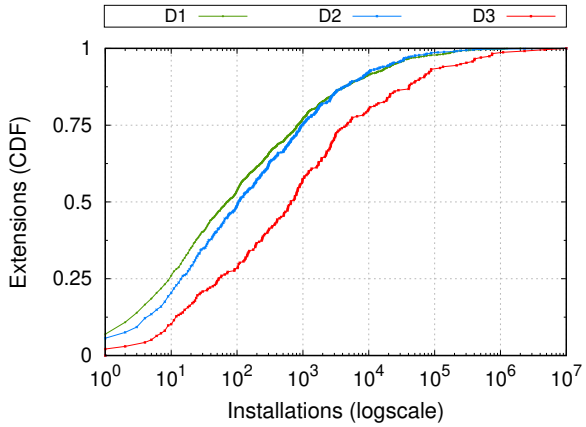

Figure 6: Number of installations for all the extensions in our datasets.

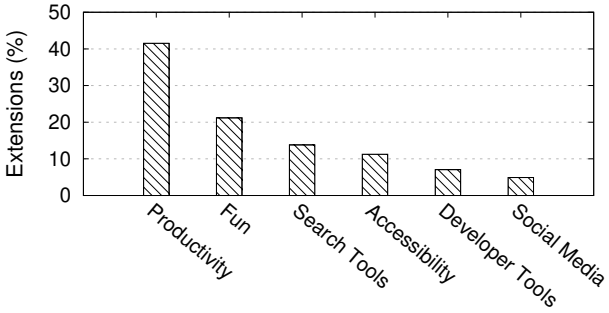

Figure 7: Categories of extensions that are fingerprinted by our system.

breakdown of extensions as shown Figure 5, "Productivity" and "Fun" are the most common categories for vulnerable extensions. Also, ≈15% of the vulnerable extensions categorized as "Search Tools" and ≈10% are under the category of "Accessibility". Finally, the least popular category is "Social Media". One difference compared to the overall distribution of extensions found in Figure 5, is that of "Developer Tools" which are less likely to be fingerprintable.

B Appendix: Countermeasure

Listing 3 shows an example of our proposed countermeasure tool for automatically injecting event-provenance checks in extensions' source code.

```
// All the mouse and key events
Events = new Set(['click', <...>])
orig = EventTarget.prototype.addEventListener;
EventTarget
    .prototype.addEventListener = function(){
  if ( Events.has(arguments[0]) ){
    let handler = arguments[1]
    arguments[1] = function(){
       let event = arguments[0];
       //event's origin
       if (event.isTrusted == false)
         return;
       else
         return handler.apply(this,arguments)}}
  return orig.apply(this, arguments);}
```

Listing 3: Code for verifying events' origin by overriding the `addEventListener` function.