Harnessing Multiplicity: Granular Browser Extension Fingerprinting through User Configurations

Konstantinos Solomos
University of Illinois Chicago
Chicago, IL, USA
ksolom6@uic.edu

Nick Nikiforakis
Stony Brook University
Stony Brook, NY, USA
nick@cs.stonybrook.edu

Jason Polakis
University of Illinois Chicago
Chicago, IL, USA
polakis@uic.edu

***Abstract*—Browser extension fingerprinting poses a *dual* privacy threat to users, as it can be used for both tracking (e.g., as part of browser fingerprinting systems) and directly inferring sensitive user data (e.g., religion, medical issues). In this work, we conduct a novel study that expands the view held by *all* prior extension-fingerprinting studies, which were limited to detecting whether an extension is installed or not, and show that extensions can exhibit *diverse* behaviors and features when personalized by users. We introduce the concept of *multi-fingerprinting*, which aims to harness extensions that exhibit diverse behaviors due to such personalization. Accordingly, we develop Hecate, a system that employs multiple techniques, including static analysis and fuzzing, for generating diverse extension configurations and capturing the corresponding behavioral signatures. We conduct an extensive experimental evaluation of Hecate, and find that it triggers diverse behaviors by uncovering and fuzzing configuration options in extensions installed by millions of users. Additionally, we analyze the real-world impact of multi-fingerprinting through a pilot user study, in which 25% of the users can be uniquely identified through multi-fingerprinting. Our study demonstrates the impact of extension personalization on the fingerprintability of extensions, while also highlighting the significant real-world privacy risk posed by multi-fingerprinting.**

1. Introduction

Web browsers are the essential gateways to the Internet, granting users access to a multitude of services. Instead of simply fetching and rendering webpages, modern browsers incorporate a rich collection of internal features that enhance the users' browsing experience. At the same time, they also allow users to augment their capabilities by installing browser extensions; these have gained significant popularity due to their ability to customize web content and deliver new functionality. Lately, the integration of advanced AI language models like ChatGPT into Chrome extensions has contributed to the additional growth of the extension ecosystem, with those extensions already being installed by millions of users [1]. However, extensions also introduce privacy risks, since websites can employ specialized techniques for detecting their presence, using them to track users and infer private user data.

Contrary to traditional browser fingerprinting techniques that mainly rely on JavaScript for accessing browser features directly [2], fingerprinting browser extensions presents challenges due to the absence of a dedicated internal or external browser API. To tackle this, researchers have proposed various fingerprinting techniques that focus on different aspects of extension functionality and execution. Early attacks detected the presence of extensions by probing for the extension's resources [3]. More recent studies have shifted towards fingerprinting extensions based on the side-effects of their behavior and execution patterns. These techniques infer the presence of extensions by identifying extension-produced page modifications [4], [5], and stylistic alterations [6]. Extension fingerprinting has also seen extensive real-world deployment as part of the FingerprintJS framework [7], as well as on websites detecting the presence of ad-blocking extensions [8].

While prior studies have greatly advanced extension fingerprinting, they have overlooked a crucial aspect; extensions are highly dynamic and multifaceted, capable of exhibiting diverse behaviors. Similar to browser fingerprinting, where a fingerprint may change based on the browser's configuration, extension fingerprints also exhibit a similar behavior. In fact, a specific user's *personalization* of an extension's appearance and functionality can result in a different set of modifications to the webpage and, thus, a different extension fingerprint compared to other users. This, essentially, will result in the extension fingerprint collected during the attack phase *not matching* the fingerprint that was generated by the fingerprinting system during the extension analysis phase and, hence, the attacker being unable to fingerprint that specific extension. Crucially, this dynamic nature of extensions affects *all* prior fingerprinting approaches as they did not consider how extensions' behavior could diverge due to user customization and only collected a single signature for each extension (i.e., for the default configuration).

Interestingly, this dynamic behavior also creates an opportunity for attackers to amplify the discriminating power of detecting a given extension. Specifically, in prior work each extension could only provide a single bit of entropy

(whether it is installed or not) and all of an extension's users were considered part of an anonymity crowd of size N (where N is the number of downloads). However, an extension's personalization (including its specific settings, preferences, and configuration) can provide more entropy through variations in the generated fingerprints. In practice, `all` prior systems would only target a single fingerprint for an extension in the wild, limiting their practicality and accuracy. This leads to an overestimation of extensions that can be fingerprinted, since for customized extensions the resulting signature would be missed.

In this paper, we introduce the novel concept of *multi-fingerprinting* to address this gap in prior approaches. We develop Hecate, a comprehensive fingerprinting system that incorporates this concept for collecting extensions' different option artifacts and generating multiple unique fingerprints. Hecate utilizes different techniques to extract *extension options*, generate multiple configurations, and dynamically exercise extensions by applying these configurations. Specifically, our system extracts the extension options through the extension's storage (*Ext-Storage*) and applies *clustering* and *fuzzing* techniques to generate numerous configurations. Subsequently, it dynamically applies the configuration to each extension and multi- fingerprints it by collecting all of the associated signatures.

Subsequently, we explore Hecate's capabilities through an extensive experimental evaluation and find that ≈13% of fingerprintable extensions employ `Ext-Storage`. Subsequently, our system is able to infer the availability of configuration options in 2,303 extensions and collect multiple signatures for 597 (14.7% of which were not fingerprintable by prior state of the art), as Hecate triggers additional functionality in the extensions by fuzzing their configurations. This results in the collection of a wide range of additional unique signatures, with highly dynamic extensions exhibiting hundreds of signatures. These extensions have been installed by more than 10 million users, highlighting the privacy risk of multi-fingerprinting, as the extensions' anonymity sets are significantly reduced regardless of the size of their respective user base. Finally, we conduct an IRB-approved pilot user study and collect the extensions and fingerprints of 375 real users to assess the impact of multi-fingerprinting. Our study demonstrates that individuals do indeed personalize their extensions, and finds that 25% of the users are *uniquely* identifiable through their installed fingerprintable extensions. Surprisingly, we found 12 users with a popular customizable extension (Dark Reader), all of which had their own distinct and unique configuration.

Overall, by introducing the concept of multi-fingerprinting we provide additional insights and a previously unexplored dimension of the extension-fingerprinting landscape. Our study makes a significant contribution in the broader area of extension fingerprinting, as it highlights the dynamic nature of extensions and the privacy implications of extension personalization, a practice that not only increases the coverage obtained by extension fingerprinting but, crucially, can significantly increase the obtained entropy associated with detecting installed extensions.

In summary, our research makes the following contributions:

- We propose *multi-fingerprinting*, a comprehensive strategy that incorporates multiple techniques for uncovering diverse extension behaviors and increasing the entropy obtainable by any behavior-based extension-fingerprinting system.
- We develop and evaluate Hecate, an automated multi-fingerprinting system. We experimentally demonstrate its effectiveness, and conduct a detailed analysis and user study that illuminates the prevalence and implications of extension personalization.

2. Background and Threat Model

In this section, we present pertinent background information on extensions, and provide technical details regarding our techniques.

Extension architecture, scripts and components. Extensions are comprised of various components that interact with each other and ultimately perform the extension's functions. The `Manifest` file specifies the necessary dependencies, permissions, as well as resources needed for the extension to function properly and achieve its purpose. The core scripts responsible for interacting with the visited webpage, including content modification, resource loading, and communication between extensions, are commonly known as `Content Scripts`. Extensions utilize these scripts to interact with and alter the page, while also communicating with the other scripts and components through browser APIs. Content scripts are declared in the Manifest under an entry that designates the domains on which they are allowed to execute and other required permissions. Typically, content scripts employ DOM requests to manipulate the page, and may also inject additional custom scripts, functions and event listeners required for their functionality.

An extension's primary application logic is usually implemented through HTML and JavaScript in `Background Scripts`. These scripts operate as individual processes within the browser and handle most of the functions that content scripts cannot perform. Since they do not have direct access to the page, they interact with other components by using the appropriate APIs (e.g., Messaging, Runtime [9], [10]). However, since Manifest version 3.0 is now supported in the latest Chrome versions, and Manifest version 2.0 will eventually be removed, background scripts will be replaced by `Service Workers` [11]. Extensions will no longer be able to use the background page to execute code in the background, requiring instead the use of service workers that are registered with the browser. The new Service Worker API has been designed to offer improved security and performance when compared to the previous Background Page API [12]. It provides developers with an extensive set of capabilities and ensures the integrity and isolation of the code in the background.

Extensions allow users to customize their functionality through an `Options page`. The options page is a dedicated component and consists of a typical HTML web page. This page allows users to enable or disable features of the extension, customize appearance or functionality, and configure advanced settings related to network usage, security, etc. Typically, users access the options page directly by clicking the extension icon in the browser toolbar or by navigating to `chrome://extensions`, locating the target extension, and clicking the options link. Developers can declare the page options in the Manifest file, either under the `options_page` field if the page is displayed in a new tab, or under the `options_ui` field where the options are embedded inside the management page.

```
//Options page configuring color and position
var color = document.getElementById('color').
    value;
var position = document.getElementById('position
    ').value;
//Save in the extension storage
chrome.storage.local.set({colorSelection:color,
    positionSelection:position})
//Backend extension storage
{colorSelection:'red',positionSelection:'top'}
```

Listing 1: Simplified example of an options page interacting with Ext-Storage for saving and retrieving options.

In order to manage data (be it user-related, execution-related or otherwise), Chrome provides the `Storage API`, which is designed for storing persistent extension data [13]. Despite the absence of specific guidelines from Chrome regarding extensions using the storage API to store options (i.e., configurations), it is a commonly employed approach among developers – even code snippets from the official documentation making reference to handling extension-level options using this API [14]. Extension storage is similar to the typical web API of local storage, but it is dedicated solely for extension usage. If the user clears any browser cache, the extension data persists since it is stored in a separate database inside the client's filesystem (`Level DB`), that only the extension can access and modify. Storage API supports mainly two storage areas, the `local` and `sync`. The local area can hold up to 5 MB of data and is erased when the extension is removed, while the sync area allows the synchronization of data to any browser that the user has logged in, and can store up to 100KB of data. Moreover, developers can also listen to changes in the storage using the `chrome.storage.onChanged` event, which allows for updating its options immediately after the user changes them. Developers access the storage API after declaring the `storage` permission in the extension Manifest. Listing 1, shows an example of options-handling in the options page and the corresponding extension storage. For the remainder of our paper we will refer to the extension storage as `Ext-Storage`.

Motivating Example. There is a wide range of extensions that enhance the user's experience on websites by modifying each page and customizing the appearance based on the user's needs. An example of such an extension is

(a) Light mode activation. (b) Brightness adjustment. (c) Dark mode activation.

Figure 1: Different extension configurations and the resulting page modifications.

"`Dark Reader`" with over 5,000,000 users, which offers a set of configurations that allow users to modify the appearance of the page. Figure 1 illustrates the different extension configurations and the resulting changes to the page. In Figure 1a, the light mode option is enabled, resulting in a change from the webpage's default white color. In Figures 1b and 1c, the brightness and dark mode are modified and the appearance of the page changes accordingly. The motivation behind our work arises from the observation that the options provided by an extension can dynamically alter its behavior, leading to modifications in the appearance and functionality of the visited web page (which would cause prior fingerprinting systems to miss it). We note that our objective is not to infer sensitive data from each extension configuration. Instead, we aim to automatically generate multiple extension configurations that result in distinct behavioral fingerprints, moving from the binary fingerprinting strategy of all prior work (i.e., "Is the extension installed?") to a fine-grained strategy leveraging multiple distinct fingerprints (i.e., "Has this user installed and configured the extension?").

Threat model. We adopt the well-established threat model used in prior browser fingerprinting studies, and assume that the user simply visits a malicious web page that aims to infer the extensions installed in the user' browser. The attacker controlling the page has already gathered the configuration options for each extension during a preparatory preprocessing phase, and generated *multiple* signatures for each extension that capture the configurations that can be applied by users. We emphasize that the attack utilizes the extension storage *exclusively* during the preprocessing phase for inferring detailed information about the extension's behavior and examining its configuration and the resulting fingerprints. During the actual attack (i.e., when the user visits the attacker's webpage) our system collects the extension signatures *only* by observing the changes made to the webpage's DOM; the attack *does not* require access to the extensions' storages (which are, naturally, not accessible by the webpage). Furthermore, in line with prior research, we focus on extensions that operate on all domains and do not limit their functionality to specific domains, as these extensions can be detected by any attacker.

3. System design

Here we detail our approach to multi-fingerprinting (`MultiFP`) extensions based on their configurable charac-

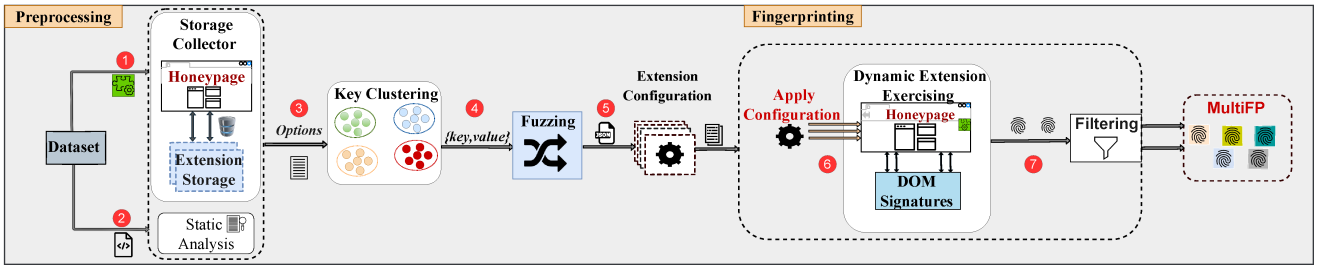


Figure 2: Overview of Hecate's two main workflows: During the preprocessing phase, the system analyzes extension's source code and Ext-Storage, to extract extension's options and generate multiple configurations. During the fingerprinting phase, the system dynamically applies various configurations and extracts DOM fingerprints.

teristics. Our methodology comprises of two phases; in the preprocessing phase, we analyze and extract the extension's configurations and apply fuzzing techniques for generating multiple option values. In the fingerprinting phase, we apply each configuration and collect the additional DOM signatures that result from the configuration. A detailed breakdown of our system, Hecate, and its components is illustrated in Figure 2.

3.1. Preprocessing Phase

Since we are interested in extracting each extension's options, we perform two types of analysis. First, we extract the extension's options from its backend storage (`Ext-Storage`) and, subsequently, we perform static analysis on the extension's source code to extract additional extension storage references.

As previous studies have shown [15], [16], navigating through a web or mobile application is far from trivial. Similarly, extracting an extension's options and categorizing them can be challenging, requiring sophisticated web-scraping techniques that employ heuristics for successfully locating the options page, interacting with it, and identifying its structure and components. Option pages can be arbitrarily constructed and may include complex elements and advanced HTML structures, as well as content in different languages.

We *bypass* these challenges and extract the extension's options by *directly* reading the extension's storage and collecting the necessary data. To ensure that our method is not bound to a specific operating system and can operate across browser vendors, we do not read the backend database directly from the filesystem but, instead, dynamically collect it while the extension runs in the browser. Crucially, to conduct the analysis, we extract the contents of each extension to directly access and modify its structure and code (e.g., Manifest, content scripts). We perform this step `solely` during the preprocessing phase. In the fingerprinting phase, we utilize the extension as it was provided by the Chrome Store and do not interfere with its intended execution and components.

1 **Storage Collector**. We focus on extensions that employ the `Ext-Storage` for handling configuration options. Based on an initial manual analysis, we found that Ext-Storage is typically created when the extension is installed in the browser and first initiated. In other implementations, the storage is populated when the user clicks the extension button and navigates through the extension's options. In our approach, we attempt to extract the extension's storage by replicating the required user interactions that trigger the extension into creating its storage. First, we generate an additional content script that extracts the extension's storage and stores it in a separate JSON object. We follow this approach and choose not to inject the function that reads the storage in the extension's content script so as to avoid interfering with the extension's execution. We also alter the extension's Manifest and include our controlled content script. When the extension is active, our content script runs concurrently with the extension's content scripts, without infererring with their intended functionality.

```
chrome.storage.sync.get(null,function(items){
   var allKeys=Object.keys(items)
   var allValues=Object.values(items)
   var ext_local=JSON.stringify(allkeys)+JSON.
       stringify(allValues)
   //make object available to the page
   window.localStorage.setItem('ext_local',
       ext_local)
})
```

Listing 2: Source code of our controlled content script that extracts the entries of the Ext-Storage.

Listing 2, illustrates the functionality of the content script used by the Storage Collector to detect and extract the Ext-Storage. The script operates as a typical content script, running in parallel with the other components of the extension, and it detects the presence of Ext-Storage when available. The process of extracting the "sync" and the "local" extension's areas are identical and no other functionality or API is required. The data is made available to the page through the local storage entry, and the content-script does not interfere with the extension components or the page. After the content-script executes, the data is available to the page directly and can be collected and analyzed further by the other system components. It is designed to avoid additional overhead from message-passing between the extension and the page, by making the data available in JSON format via the page's Local Storage.

As detailed above, the extension creates its storage after the first initialization or when the user interacts with the options page. To extract it, we install the extension in the browser and navigate to a "honeypage", a page that we control which aims to trigger extension functionality through the presence of a wide range of specific HTML elements (forms, buttons, terms, etc.). This honeypage also extracts the extension's resources and behavioral signatures. During the first visit, we let the extension initialize itself and fetch any required resources, and then close the tab and revisit the honeypage in a fresh tab to minimize any interference with the first load. During the second visit, our controlled content-script runs in the page and retrieves the entries saved in the storage. We also automate the user's interaction by clicking on the extension icon to trigger the extension so it can populate the options in its Ext-Storage. After allowing some time for the extension to complete its functionality, our content-script retrieves the storage and makes it available to the webpage in the form of a JSON object containing sets of `key:value` pairs. Once the object is available, we store it for further processing.

2 **Static analysis**. The aforementioned component that is responsible for extracting extension storage data plays a fundamental role in our system. However, extensions may also reveal additional options and configurations under certain conditions. For example, an extension may initiate additional options whenever a specific element is present on the page or when a user interaction is triggered. Since we cannot exhaustively exercise an extension to exhibit all possible behaviors, we collect additional options through static analysis. Specifically, our goal is to gather further option *keys* that were not initialized in the previous stage, and also accumulate as many corresponding *values* as possible for both the newly collected keys and those previously acquired.

To that end, we design a methodology for detecting Storage API calls, and extracting their arguments and values. The methodology relies on a set of heuristics that identify API calls regardless of the code's structure. At first, we use Python's `jsbeautifier` API [17] to deobfuscate extensions' scripts code. Since options and Ext-Storage might be accessed by any extension component, we analyze all such scripts. We leverage Esprima [18], a popular static analysis framework, to build each script's Abstract Syntax Tree (AST). Once the AST is created, we focus on the calls to `chrome.storage.local.set()` and `chrome.storage.sync.set()`, as these API calls are associated with the Ext-Storage. To detect them, three conditions need to be satisfied. First, the node's `callee` type and the `callee.object.type` need to be MemberExpression. The second requirement is that the `object.object.name` must be "chrome" and the object's property name "storage". Finally the property needs to be labeled as "set".

Once we have located the function calls, we extract the arguments by accessing the arguments property of the `CallExpression` node. This property contains an array of argument nodes, each of which represents one of the arguments passed to the function. We then traverse these argument nodes to access the `Property` node where it stores the values of each argument. For example, for the call `chrome.storage.local.set({x:5})`, we access the first argument node in the "arguments" table and extract the key ("x") and "value" properties of the "Property" node representing the value of the argument (x=5).

Depending on the implementation, the source code might also contain cases where storage calls contain symbolic variables. For instance, the call `chrome.storage.local.set({foo:bar})` sets a variable "foo" and the value is another variable "bar", which can be of any type. To handle this, we categorize the values extracted from the previous step and exclude those that belong to a primary type (e.g., string). For the remaining values, we perform a traversal of the AST to extract all the potential values. Since we know the variable's name, on each traversal, we attempt to detect all the `ExpressionStatement` node types where the "left" property's label is the variable and the "right" part of the expression contains the value. We gather all the data extracted from the static analysis and the Storage Extractor component, remove any redundancies, and form a list of potential options. For the keys that we collect, even if there are extracted values, we include them in the options and handle their values in the subsequent phases of our analysis.

3 **Key Clustering**. After collecting all the options data, we analyze the keys of the option objects and categorize them based on their type and values. Since we aim to generate multiple extension configurations, term clustering is suitable for grouping all the keys of the same or similar context. Option keys do not follow any specific pattern, and can either be a typical English term (e.g.,"enable"), acronyms, initials, or even arbitrary strings. Clustering such terms is challenging since there is no dedicated algorithm for this type of non-uniform data. However, our objective in the data collection process is not to achieve perfect clustering and grouping of the data; instead, we aim to collect the values associated with each keyword and expand their value range. Given that the extension storage may include variables that are not linked to extension configurations, Hecate employs a best-effort approach to gather and classify key-value pairs, filtering out those unrelated to configuration settings. To ensure the formation of a representative dataset, we also manually analyze the extracted data. While unrelated variables may still be considered and tested, those that are not user-facing configuration options are unlikely to alter the extension's behavior and resulting signatures.

We employ the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm, which is a widely used algorithm for clustering data [19]. The algorithm starts by selecting a random data point, then examines its neighboring points within a specified radius, and if there are sufficient neighboring points, it identifies them as a cluster and proceeds recursively. The algorithm also requires a metric to compute the distances between keywords, and

TABLE 1: Cluster labels with examples of popular keys and representative values.

Cluster	Keywords	Values
`Boolean`	enable, enableInjection, activate, isLoaded, is_open	true, off yes, close
`Appearance`	transparency, sensitivity, opacity, decoration, shadow, blur, gradient	8px, center, underline, 0.5, hidden, 50%
`Color`	color, background-color, text-color, border-color	black, #5EAE64, rgb(255,255,0)
`Position`	position, width, height, resolution, margin	top, 20px, 800x600
`Language`	selected_lang, locale language, langToTranslate,	en, es-ES english, spanish
`Fonts`	font-family, font-style, font-variant, font-size	3em, bold, Arial, small-caps

for this we apply the Levenshtein distance metric. Finally, we also optimize the algorithm by applying the Silhouette Method to find the optimal number of clusters [20]. As shown in Table 1, we have identified six key clusters, each representing a different type of data stored in the options objects. The size of clusters varies, as some keywords are more common or share parts with other keywords (such as "color" and "background-color"), while others are less frequent and only appear in a small number of options. Regarding the Boolean cluster, we formed it manually before applying the clustering algorithm, based on the keys being associated with Boolean and Boolean-like values.

We manually verify each cluster extracted by the algorithm to ensure that it does not include any arbitrary data. We also appropriately merge smaller infrequently-referenced clusters with larger ones. For example, in the Appearance cluster, we group together the less frequent keywords and generate individual clusters, such as blur, transparency, opacity, since they all reflect similar attributes on the page and describe similar options. Likewise, we perform additions and alterations in every cluster to expand them effectively. In general, each keyword within the same cluster can have different values based on its initial value. For instance, opacity can be represented as either a float or a percentage. Clustering allows us to match keys without initial values to a set of potential values based on the other grouped keywords.

4 **Fuzzing**. After extracting and clustering the extension-option data, our next objective is to generate multiple extension options that represent potential extension configurations. To accomplish this, we draw inspiration from software fuzzing. The main idea behind fuzzing is to generate a large number of test cases with diverse inputs, in order to identify flaws in the software's behavior. In our case, we use this technique to extract potential values from the data, and apply them to the extension configuration with the goal of changing an extension's behavior and, ultimately, leading to additional extension signatures.

First we focus on the keys belonging to the `Boolean` group, which holds keys that have values that are directly stored as boolean, or the value is a string but stores binary information. This includes all the keys that store data such as ``` ``No'', ``Off'', ``Activate'' ```. Since these type of keys do not have a wide range of potential values, we directly generate all the values by applying the logical negation (not) to their initial value. For the binary strings, we analyze the form of the value, and also extend it accordingly (e.g., `isOn` is changed to `isOff`). The above analysis is straightforward yet fundamental for Hecate's data generation.Next, we focus on the clusters that hold diverse forms of data. First, we analyze each cluster's values, and we group together values that belong to the same category and type. For example, in the cluster that holds color keys, we find different values of colors ("red","green") and group them together. Using heuristics, we then detect the context of all other string values and categorize them accordingly. The main categories include hexadecimal (common representation of colors), language codes, country codes, percentage strings, CSS layout keywords, resolutions, and strings storing numerical values.

Next we expand each category and generate new potential values. For instance, in the case of language codes (e.g., initial value "en"), we include the 30 most common languages in the list of values. Similarly, for the color keys, we include multiple values in their required form (e.g., string representation or hex). Regarding position-related values and percentages, we randomly generate 100 values and we apply them in the list of values for each key. Similarly, we generate common resolution strings (in the form of `Width x Height`) as well as common CSS layout keywords (e.g., top-left, bottom-right).

The last value type that we handle is numeric values, whether they are stored as strings or directly as integers or floats. Since we cannot infer their usage based solely on the key name, we generate a list of representative values that are bounded in a way that reduces misconfigurations. Specifically, for each numerical value we produce four new values: two that are 10% over/under the original value, and two that are 100% over/under the original value. These transformations are meant to capture how minor and major adjustments affect an extension's signatures without using arbitrary values that may crash the analyzed extensions.

5 **Generating Extension Configurations**. In the final preprocessing step, we gather all the data extracted from the fuzzing process. Based on the initial structure of each extension's options, we apply the newly collected values to generate all available extension configurations. This allows us to simulate the actions where a user interacts with the options page and configures an extension, while circumventing all the aforementioned issues with handling arbitrary UIs. To be as comprehensive as possible, we generate a large number of configurations that will later be used as input for exercising each extension and extracting its signatures. To generate these configurations, we follow a top-down approach and start by modifying keys with a limited set of values (e.g., binary). We store each new value in the corresponding key and generate a new configuration containing it. We proceed by altering the keys that have a larger

range of potential values and, similarly for each value, we generate a new configuration. When all the values have been applied for each key, we then proceed with a combinatorial approach, where for each configuration that has more than one key, we select two keys and apply their values to generate all the possible combinations. We then proceed with modifying three keys and applying the same approach again. This process ends when *all* the keys are modified, and all corresponding configurations are generated.

3.2. Fingerprinting Phase

Once we have gathered all the required data related to an extension's configuration, we focus on identifying the extension's behavioral signatures through DOM modifications.

6 **Dynamic Extension Exercising**. For each extension configuration, we employ the same technique used in the Storage Collector component for save the data to storage. Specifically, we initially install the extension in the browser and allow some time for it to initialize before closing the tab. We then inject the options into the extension's storage through a separate content script, for the extension to read and apply them. The content script holds all the required information for saving all the option entries in the Ext-Storage. For example, to apply a modification to a binary value, it includes the following code: `chrome.storage.local.set({enable: false})`. During the second visit to the controlled page, the extension accesses the Ext-Storage and applies the configurations based on the stored options. Our system bypasses any user-based interaction that might be required for configuring an extension, and successfully stores the data directly. After applying the extension configuration, we let the extension again perform its intended functionality, and collect the extension's behavioral modifications (e.g., modifications introduced to the page's DOM). We follow the approach of *continuous* data collection introduced by Solomos et al. [21], and include a `Mutation Observer` [22] on the webpage to collect all the data. The resulting signature contains an ordered set of continuous modifications that are distinguished based on the type of the modification (e.g., addition/removal of page element, or alteration of an attribute).

7 **Filtering & Fingerprint Generation**. Here we gather all the DOM signatures in order to form the final fingerprints for each extension. We extract all the signatures and filter out all identical signatures, which allows us to form the `Multi-Fingerprint` set of signatures for each extension.

4. Framework Implementation

In this section, we provide an overview of the APIs, frameworks, and techniques used to develop the various components of Hecate. To enable interactions with the browser and webpages, we utilize the Selenium framework [23] to automatically control the browser. For the Storage Collector component, we additionally employ Python's PyAutogui API [24] to emulate the user's navigation to the webpage, discovery of the extension button, and click actions. To accomplish this, we provide the mouse movement with `x,y` coordinates that direct it to the extension icon and enable the click action. Since we have control over the browser and its automation, this process is straightforward and can be applied across all extensions.

We reached out to the authors of the Chronos framework [21] who provided us with access to their data, including datasets, honeypages, and fingerprint generation code. This allows us to directly contextualize our findings in comparison to the capabilities of existing state-of-the-art techniques. The honeypage employed by Chronos and other extension fingerprinting systems [25], [5] contains multiple resources from a realistic webpage, and is capable of triggering extensions. In various components of our methodology, we typically allow extensions to initialize themselves for a specific duration before collecting their Ext-Storage and fingerprints. We use a threshold of 8 seconds, which was found to be adequate for extensions to reveal themselves and perform their intended functionality according [21]. Finally, all of our analysis including static analysis, clustering and fingerprints was implemented in Python using built-in and external libraries [26], [27].

To enhance our system's efficiency, we deploy our framework in a Docker Container [28], that gives us the flexibility to run multiple parallel browsers, with different configurations and versions, under different scenarios. For all of our experiments we employed two desktop machines with a 6-core Inter Core i7-8700 and 32GB RAM connected to the university network. We host the honeypage in our university's infrastructure and network, using Python's Flask framework [29] under an nginx server [30].

5. Evaluation

In this section, we present our extensive experimental evaluation of Hecate as part of our exploration of multi-fingerprinting.

Dataset. For our evaluation, we utilize the dataset provided by Chronos [21], consisting of 38,482 extensions collected between 2018 and 2021 where 12,251 extensions are uniquely fingerprintable. Additionally, in March 2024, we collected a new dataset comprising 23,269 extensions that operate across all domains without restrictions. Among these, 15% are newer versions of extensions found in the original Chronos dataset. We also ran Chronos on the latest dataset and fingerprinted 5,124 extensions. Our analysis focuses on the following datasets: `FP_Ext`, which contains 17,375 uniquely fingerprintable extensions and their multiple versions, and `NonFP_Ext`, which includes 44,376 extensions that were not fingerprinted in any dataset or did not generate a unique signature. Since our focus is on extensions that are fingerprintable, we perform the majority of our analysis on the `FP_Ext` dataset.

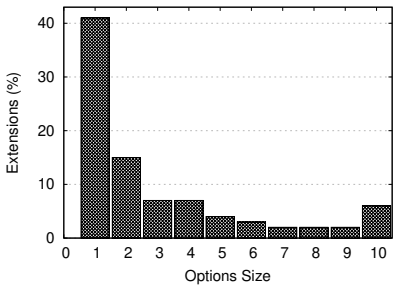


Figure 3: Number of entries in the options.

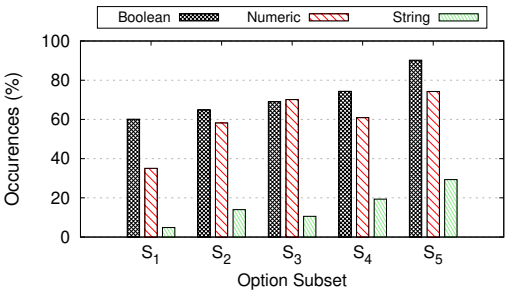


Figure 4: Types of values for each subset of options.

5.1. Extension Options Characteristics

By running our Storage Collector, we are able to extract 2,303 extension options, and ≈13% of the fingerprintable extensions employ `Ext-Storage`. It is worth noting that extensions may utilize the storage for various purposes (e.g., execution-related data) and there is no distinction between stored options and the rest of the data. Nonetheless, our clustering approach enables us to collect all the keys that are part of an extension's configuration and extend their values accordingly.

Figure 3 shows the distribution of the number of keys for each option object collected. As can be seen, the lower number of keys is more frequent, with 41% of the extension options containing only 1 key, followed by 2, 3, and 4 keys accordingly. Since the distribution is not uniform, the frequency of keys beyond 5 is less than 5%, with the number of keys of size 9 having a value of 1%. Additionally, an aggregate of ≈ 6% of the lower frequencies represents the number of keys higher than 10, with the maximum number of keys found in options being 224. This result indicates that developers often offer users few and straightforward configuration options reflected through a limited number of option keys in `Ext-Storage`. The minority of cases where large numbers of option keys are involved, are associated with complicated extensions that expose more advanced features and dynamic behaviors. For the rest of our analysis, based on this distribution, we will distinguish the options dataset into two groups: 4 groups of options with keys up to 5, and 1 group of options with keys over 5 entries. We will refer to them as S_x where x defines the number of keys. To better understand the types of values stored by the options, we categorize them into `Boolean`, `Numeric`, and `String` and count their occurrences in each subset. Figure 4 illustrates the types of values for the different subsets of options. The Boolean category is the most dominant across all datasets, indicating that the majority of options contain at least one boolean key. Moreover, in the S_5 subset that contains the highest number of keys in options, ≈ 90% of the options include boolean keys. We expect that most user-exposed Boolean options are related to user-controlled enabling/disabling of specific functionality in each extension. The second most popular type is numeric values, followed by string values. Similarly, we observe that the subsets that contain multiple keys store a more diverse set of values, reflecting the presence of multiple options.

5.2. Multi Fingerprinting

In "multi-fingerprinting", our goal is to collect as many signatures as possible, reflecting different behaviors triggered by different extension configurations. Table 2 presents a breakdown of MultiFP detections of Hecate for each subset of the dataset. In summary, after applying each of the techniques, we are able to generate additional signatures for 509 extensions. Next, we detail the different exercising techniques and analyze various aspects of MultiFP.

Exhaustive Testing. For the first experiment we utilize an exhaustive strategy, which involves testing a large number of configurations for each extension. Since we do not know which of the potential value will effectively customize an extension, we apply *all* the values that we have extracted for each key through the fuzzing procedure, individually. By applying this technique, we are eliminating the need for choosing the appropriate configuration for each extension, as the exhaustive testing reveals an "upper bound" on an extension's signatures. We test each value of every key individually, one key at a time, without interfering with the simultaneous selection and modification of multiple options.

Table 2 shows the number of MultiFP extensions for each subset (column "Exhaustive"). For each subset Hecate is able to successfully trigger extensions into revealing additional signatures, based on the applied configuration. For the S_1 subset, which contains the higher number of extensions, we observe that ≈ 13% of the extensions generate additional signatures. For the subsets of the dataset that hold higher number of keys, the detection rate increases to 36% for the S_4 and 22% for the S_5, respectively. This confirms that extensions that include higher number of options will yield additional fingerprints. The options available in an extension determine its behavior, which after customization results in different visible modifications being made to the page.

Random Key Sampling. After exploring the exhaustive technique, we introduce a different strategy and focus on experimenting with different option subsets. In this experiment, we choose multiple keys simultaneously to assess the impact of selecting multiple options and customizing the extension accordingly. Since extensions may expect multiple keys to customize their behavior, this approach aims to

TABLE 2: Detailed number of extensions and their MultiFP signatures for each subset of the FP_Ext.

Subset	Extensions	Exhaustive	$Keys_{25\%}$	$Keys_{50\%}$	$Keys_{75\%}$	$Keys_{100\%}$	Unique
S_1	952	128	N/A	N/A	N/A	N/A	128 (13.4%)
S_2	355	82	N/A	N/A	N/A	65	94 (26.4%)
S_3	180	41	N/A	N/A	28	23	50 (27.7%)
S_4	180	65	N/A	30	42	39	76 (42.2%)
S_5	636	143	34	64	66	103	161 (25.3%)
Total	2,303	459	34	94	136	230	**509 (22.1%)**

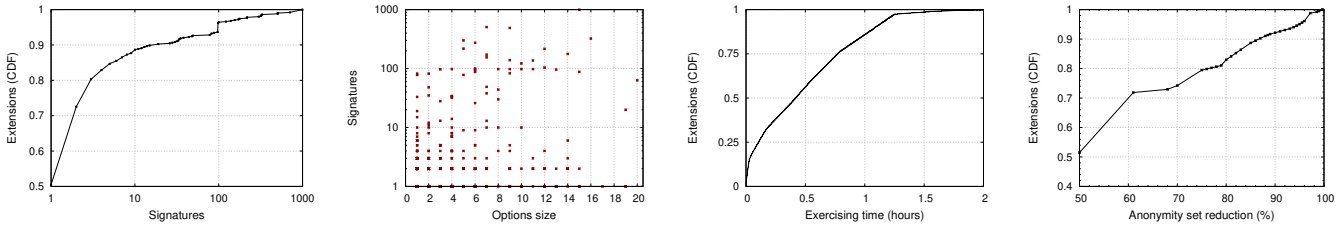
capture more complex extension constraints. To evaluate the technique, we randomly select N option subsets, where N represents the ratio of chosen options. Specifically, we conduct experiments with different ratios (25%, 50%, 75%, and 100%) in order to cover a range of option subsets. The selection of subsets is adjusted based on the size of each option object. For instance, for smaller sets of option keys we only select 75% and 100% when applicable since, e.g., choosing 50% of 1 option-key is not meaningful. For options storing more than 5 keys, we are able to apply all the percentage values. Regarding the value range, we narrow down the set of values that we previously tested by extracting only those that, when applied in the exhaustive strategy, resulted in MultiFP signatures. In cases where the method selects a key that was not associated with a MultiFP signature, we apply all the extracted values, with an upper limit of 20 values per key.

Table 2 presents the results of the random sampling technique for the different sizes. For the smaller subsets of our dataset (2 and 3) we observe that the sampling techniques reveals nearly the same number of extensions as the exhaustive technique. However, in the larger subsets, each technique reveals a different number of extensions, with the 75% and 100% selection subsets being more successful. This emphasizes the importance of utilizing multiple values when configuring extensions that have multiple options. These extensions rely on different values being assigned to their configuration in order to modify their behavior, which in turn modifies their DOM signature. Furthermore, when selecting options and assigning distinct values to each key, there is a possibility of unintentionally disabling certain functionality within the extension. Also, contradictory options may be applied, and this may lead to either an interruption in the extension's functionality or no discernible change to the extension's signature. In total, each technique has its own unique advantages and by combining them we are able to successfully multi-fingerprint 509 unique extensions. Overall, we conclude that extensions exhibit multiple behaviors when appropriately configured.

MultiFP Properties. After multi-fingerprinting each extension, we analyze the number of signatures – we show their distribution in Figure 5a. We find that 51% of the extensions exhibit one additional signature. This suggests that the majority of configuration-sensitive extensions introduce one more behavioral modification, resulting in one additional signature that differs from their initial. This observation may indicate that developers provide simple configuration options to make slight adjustments to the extension's behavior. Furthermore, we are able to trigger between two and five additional signatures for around 35% of the extensions. About 5% of the extensions exhibit more than 100 signatures, indicating that these extensions possess multiple widely configurable options, and each value assigned to these options influences the extension's signature. Extensions that alter the appearance of the page utilize keys with varying values, and each value directly modifies the page. For example, keys such as color, contrast, and brightness can have multiple values individually and, when combined, they all generate additional different signatures. However, these extensions are not restricted to this specific upper bound of signatures and can generate up to N signatures when different configurations are applied and advanced combination techniques are introduced.

To further quantify the properties of MultiFP, for each option size we count the number of the detected MultiFP signatures. Figure 5b presents the distribution of MultiFP signatures for each extension, ranked by the option size. For extensions with a single configurable option, the distribution of signatures is diverse, ranging from 1 to 100. This indicates that, depending on the type of configurable option, a single option does not necessarily translate to a single additional fingerprint. For extensions with 5 or more keys, we observe that the total number of MultiFP signatures trend upwards. This suggests that these highly-configurable extensions change their page footprint depending on a number of properties which in turn results in additional signatures. At the same time, a larger number of options does not necessarily lead to a larger numbers of signatures. Even for options ≥ 10, there are extensions that generate only one additional signature. This indicates that only one of the multiple keys in the configuration directly affects the signature, and the others do not affect the extension's behavior/appearance in a detectable manner. Overall, we observe that the absence of a standardized definition of options by Chrome and the lack of developer guidelines for the options page, results in developers adopting different styles of options, resulting in different levels of MultiFP signatures.

MultiFP Performance. Next, after providing a comprehensive overview of the number of signatures for each extension and the strategies employed to collect them, we compute the exercising time required for MultiFP. Specifically, we measure the time by computing the number of configurations applied to dynamically exercise the extension and the time required for the extension's internal functionality. On average, the system requires approximately 10 seconds. This duration represents the time required to apply the configuration and inject the content script, load the page and enable the extension to execute its intended functionality, and the additional overhead of restarting the browser after each distinct configuration. Figure 5c presents the time distribution for collecting MultiFP signatures. Approximately 25% of the extensions can be successfully multi-fingerprinted within one minute, whereas 50% within 40 minutes. These extensions typically incorporate straightforward and easily adjustable options. The top 2% of extensions are more complicated as they contain more complex options with multiple potential values. These extensions require up to 2



(a) Extension's signatures. (b) Signatures & options's size (c) MultiFP exercising time. (d) Anonymity set reduction.

Figure 5: Figure 5a illustrates the number of MultiFP signatures found for each extension, Figure 5b presents a breakdown of the number of MultiFP signatures for each option size, Figure 5c presents the total MultiFP training time while Figure 5d, illustrates the Anonymity set reduction within each extension's user base.

hours of automated testing to trigger the desired behaviors and extract the corresponding signatures. Overall, we find that Hecate is efficient and can multi-fingerprint extensions within a practical timeframe. We note that this testing is done offline and only needs to be repeated whenever an extension is updated.

Anonymity Set Reduction. After analyzing MultiFP's characteristics, we explore its effectiveness in reducing the anonymity set within a given extension's user base. Figure 5d presents the reduction in extensions' anonymity sets (for simplicity we assume a uniform distribution of configurations). Nearly 50% of the extensions decrease their anonymity set by 50% indicating that these extensions exhibit a single additional signature that reduces the anonymity set by half. Furthermore, almost 30% of the extensions achieve more considerable reductions, ranging from 50% to 80%. For the remaining 15%, the reduction exceeds 90%, representing extensions with larger number of signatures and varying sizes of user bases. These results highlight the effectiveness of MultiFP, particularly for extensions with a large user base, since attackers can utilize MultiFP to reduce the anonymity sets of users of the same extension. In practice, the reduction in entropy may not be uniformly distributed across all anonymity sets and can vary; in that case, while for certain configurations the added entropy will be lower, for the remaining it will be higher.

Missed Extensions. Up to this point our focus has been on exploring the multi-fingerprintability of previously-fingerprintable extensions. This subset was extracted from 17,375 extensions, and represented extensions that were fingerprintable using state-of-the-art techniques [21] on all datasets. However, out of the total dataset, there are still 44,376 extensions that prior work could not fingerprint. This means that they did not have any signature or, if they did, it was common with other extensions and it was not possible to uniquely identify them. To evaluate Hecate's efficiency, we focus on this dataset with two objectives: (i) triggering extensions that were not fingerprintable, and (ii) resolving conflicting signatures. Using Hecate, we extract 1,717 extensions that employ Ext-Storage. We also conduct the same analysis as before to classify the options-sizes, and observe a comparable distribution.

For our evaluation, we apply the same MultiFP techniques to effectively activate the extensions through their configurations. In total, Hecate is able to detect 88 additional extensions that were not fingerprintable by prior work [21]. Specifically, 21 extensions were detectable but their signatures were not unique due to collisions. The extracted signatures from these extensions often conflicted with each other due to shared libraries among developers or similar behaviors, whether intentional or unintentional. We observe that by using MultiFP, we successfully resolve a subset of these collisions. This was achieved by fuzzing configurations and through dynamic exercising, which directly modified the extension's signature. Example configurations include the activation of a setting (e.g., `enable-data:true`) and the modification of default values that are directly reflected in the signature.

Similarly, Hecate was able to trigger the functionality of the remaining 67 newly detected extensions. These extensions required a modification in their default options page to initiate their functionality. This functionality includes script injections or modifications of the appearance of the page, which is a typical behavior triggered through MultiFP. Regarding the number of signatures, since a portion of the extensions did not have an initial signature, we find that $\approx 30\%$ of them have between 2 and 10 additional signatures, which could further increase with more exhaustive configuration testing.

User study. After analyzing and evaluating the characteristics and capabilities of Hecate's multi-fingerprinting, we aim to investigate whether real users personalize their extensions and how this impacts their fingerprintability. Prior to collecting any user data, we submitted a proposal to our institution's IRB detailing our intended data collection process and overall study's goals. After our proposal was evaluated and accepted, we conducted two separate studies; a larger one with participants recruited through the Amazon Mechanical Turk platform, and a smaller scale pilot study with colleagues and collaborators. Our studies require users to install an extension that retrieves the list of installed extensions, which we use as ground truth. We also deploy a website that incorporates Hecate's honeypage for generating and collecting fingerprints. When users visit our website, they are instructed to download our extension and, upon suc-

TABLE 3: Detailed option customization for the `Dark Reader` extension for the 12 participants sharing the extension. The RGB value denotes the value defined in the page's attribute after user customization, and the actual color represents the resulting color.

User ID	RGB Value	Actual Color
1	(79, 79, 6)	
2	(102, 102, 64)	
3	(125, 125, 64)	
4	(128, 128, 128)	
5	(156, 156, 64)	
6	(163, 159, 78)	
7	(153, 153, 0)	
8	(122, 122, 0)	
9	(66, 66, 3)	
10	(89, 89, 45)	
11	(112, 112, 45)	
12	(115, 115, 115)	

cessful installation, are redirected to the honeypage where their fingerprints are collected. To match the fingerprint of each extension when users install multiple extensions, we utilize the technique introduced by Chronos, which extracts multiple extension fingerprints from the same user. We employ pseudorandom identifiers for each user, and only collect the list of installed extensions and the generated fingerprints.

Overall, 375 users participated in our study with a total number of 879 unique installed extensions. On average, users have ≈7 extensions installed in their browsers, and the most popular were Tampermonkey, Mturk Suite, and PandaCrazy Max which are typical extensions installed by Mturk users. Importantly, 96 users (≈ 25%) are uniquely fingerprintable by Hecate. These users exhibit a higher average number of installed extensions (9), with the majority (98%) being uniquely identifiable by a single fingerprintable extension within their set. We note that while participants commonly install MTurk and survey management extensions, this does not otherwise affect the overall set of installed extensions. Each user customizes their extensions based on personal preferences, resulting in a diverse set of extensions, configurations and fingerprints across users.

Regarding extension personalization, we found that 12 users had installed the popular extension `Dark Reader`, which offers multiple configuration settings. Notably, *all* of the users exhibited unique extension fingerprints, reflecting the diverse nature of extension personalization. This verifies that users do personalize their extensions, and such customization can be directly reflected in an extension's signature, which effectively reduces its anonymity set.

Table 3 presents the 12 different extension customizations applied by the study participants and the resulting color used to change the background of the page. Specifically, the `Dark Reader` extension injects a unique attribute element into the page that customizes the background whenever users adjust the extension's options. When users activate the extension it directly sets the background to black. Users can further customize it using attributes such as Brightness, Contrast, Sepia, and Grayscale, each with different values. The unique combination of these options results in a distinct RGB color that the extension utilizes to modify the page's dark background accordingly. We note that, despite some colors appearing similar, their RGB representations differ, resulting in distinct signatures. This behavior highlights that even minor or seemingly similar customizations lead to different signatures.

To further quantify the anonymity set reduction across different option sizes, we conducted an additional study. We invited 10 participants to install four different extensions, two of which offer multiple options and configuration values (over 10), and two with a smaller number of options (less than 5). We instructed participants to use the extensions for one week, become familiarized with them by using them on various websites, and customize their settings according to their personal preferences. After one week, the users revisited our study website. For the first set of extensions with multiple options, all users customized them accordingly, resulting in unique fingerprints that reflected the diverse configurations within the signatures. For the second set, we observed that even though users customized the extensions, their fingerprints reflected the limited options and led to fingeprinting collisions. Specifically, for the extension that only offered three predefined options to choose from, we observed that none of the user's fingerprints were unique. Nonetheless, the extension's anonymity set was effectively reduced, allowing an attacker to target a smaller group of users who applied specific options. These results demonstrate that MultiFP does augment the overall effectiveness of extension fingerprinting and the discriminating power of each fingerprint.

6. Discussion

Mitigation. Hecate builds upon prior fingerprinting systems, and employs a mutation observer in the page to continuously collect extension's modifications [21]. As detailed by prior work, traditional DOM fingerpriprinting defenses, such as the randomization of extension's modifications [31], are ineffective against continuous fingerprinting. However, recently proposed defense mechanisms [32] can directly impact our system by isolating the mutation observer within the context of the extension, effectively preventing extensions from being fingerprinted by the web page. Nonetheless, the concept of multi-fingerprinting could likely be applied to other behavioral fingerprinting vectors (e.g., CSS-based).

Moreover, our study sheds light on a previously unknown aspect of extension fingerprinting, highlighting the inherent tension between customizability and fingerprintability. Extensions operate in a standardized way, and consistently apply user customizations. Any change to this behavior will directly impacts the extension's functionality. Since there is no dedicated mechanism to mitigate the fingerprinting risks posed by personalization, developers must carefully balance customization and fingerprintability. This

is a crucial design decision, as integrating options into the core logic can significantly modify the extension's intended behavior. Developers and users alike must decide whether the gain in user experience that comes with multiple configurable extension options, justifies the potential reduction in privacy. Our results can guide developers in adopting better practices, while also encouraging browser vendors to strengthen their defenses against extension fingerprinting.

Manifest V3. The extension ecosystem is currently in transition due to Google Chrome's enforcement of Manifest version 3, which introduces additional security measures. Specifically, developers can define the page's frames and the origins where content-scripts are permitted to execute. This level of granularity is achieved through new Manifest properties and the addition of the Content Security Policy. However, despite these significant security enhancements that isolate the content-script, extension fingerprinting is not impacted, as extensions continue to operate within the web page, allowing their modifications to be fingerprinted.

7. Related Work

As browser fingerprinting has become a popular stateless tracking technique, a large body of works has studied its prevalence and feasibility and alternate applications [33], [34], [35], [36], [37], [2], [38], [39], [40], [41], [42]. Additionally, numerous studies have explored the dynamics and evolution of browser fingerprinting in realistic deployments [43], [44], [45]. More recently, extension fingerprinting has generated interest in both industry and academia as an additional fingerprinting vector that detects users' extensions and tracks them accordingly. Different techniques have been proposed, including using their metadata [3], [46], [47], [48], and the modifications that they introduce to the page [6], [4], [5], [21], [49]. Sanchez-Rola et al. [47], introduced a side-channel attack that identifies browser extensions with high accuracy. Their approach involves sending a request to unavailable extension resources and logging the response time to infer the presence of the extension. Various studies have also proposed techniques for detecting an extension's resources, known as Web Accessible Resources (WARs) [3], [50]. The presence of specific WARs can be used to infer the presence of extensions. However, for these attacks, browser vendors and the community have proposed defenses that are actively under adoption [51], [52], [53], [54], [55], [25].

Regarding behavioral extension fingerprinting, one of the first studies [4], showed that extensions exhibit unique behaviors in the way that they modify the page, while this behavior can be collected to create the extensions behavioral DOM fingerprints. In the same direction, Karami et al. [5] developed a system (`Carnus`) that automatically generates behavioral signatures by collecting all the modifications in the DOM and the communication messages between the extension, the page and external resources.

Moreover, Solomos et al. [49] explored the fingerprintability of extensions triggered through user interactions. Specifically, they designed a framework that emulates realistic user interactions on the page and fingerprinted extensions that required such interactions. In the most recent DOM fingerprinting study [21], the authors designed a system that generates extension fingerprints based on continuous DOM modifications. Their system was able to detect additional behaviors of extensions that perform ephemeral modifications to the page, which other systems could not detect. Finally, Laperdrix et al. [6], proposed fingerprinting technique based on the cascading style sheets (CSS) extensions inject, and detected extensions that were missed by prior fingerprinting systems.

As extension fingerprinting techniques have become more advanced, browser vendors and the research community have developed defenses to limit their effectiveness. Trickel et al. [55] introduced `CloakX`, a system that prevents DOM fingerprinting. CloakX focuses on the extension's injections and randomizes the injected elements. This approach introduces noise to the page and attackers cannot accurately collect the DOM modifications and fingerprint the extensions. In a recent work, Karami et al. [32] proposed a system that prevents DOM-based fingerprinting by creating a separate DOM only accessible to extensions.

8. Conclusion

Extension fingerprinting has garnered the attention of the security and privacy community while also seeing extensive real-world deployment. All prior work treated extension fingerprinting as set of binary inference tasks, where each extension is either installed or not. In this work we challenged this notion and demonstrated that extension fingerprinting should be treated as a multi-modal inference task, as extensions exhibit diverse behaviors and characteristics based on users' preferences and configuration (i.e., *personalization*). To that end, we developed Hecate and extensively explored *multi-fingerprinting*, which relies on triggering and capturing configurable extension behaviors and features. Consequently, prior and future extension fingerprinting systems can integrate our techniques so as to truly capture extension's diverging behaviors, and maximize the entropy offered by extension fingerprints. Finally, we emphasized the importance for extension developers to adhere to best practices in extension customization and browser vendors to adopt additional privacy protections for extension fingerprinting.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the National Science Foundation under grants CNS-1941617, CNS-2211574, CNS-2211575, and CNS-2143363, as well as the Army Research Office (ARO) under grant W911NF-24-1-0051. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the NSF or the ARO.

References

[1] Jon Martindale, "The best chatgpt chrome extensions to bring ai to your browse," https://www.digitaltrends.com/computing/best-chatgpt-chrome-extensions/, 2023.

[2] P. Laperdrix, W. Rudametkin, and B. Baudry, "Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 878–894.

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

[4] O. Starov and N. Nikiforakis, "Xhound: Quantifying the fingerprintability of browser extensions," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 941–956.

[5] S. Karami, P. Ilia, K. Solomos, and J. Polakis, "Carnus: Exploring the privacy threats of browser extension fingerprinting," in *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2020.

[6] P. Laperdrix, O. Starov, Q. Chen, A. Kapravelos, and N. Nikiforakis, "Fingerprinting in style: Detecting browser extensions via injected style sheets," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[7] Karl Hughes, "FingerprintJS - Empowering developers to solve fraud at the source," 2021, https://fingerprintjs.com/blog/browser-fingerprinting-privacy/.

[8] A. Nisenoff, A. Borem, M. Pickering, G. Nakanishi, M. Thumpasery, and B. Ur, "Defining" broken": User experiences and remediation tactics when {Ad-Blocking} or {Tracking-Protection} tools break a {Website's} user experience," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 3619–3636.

[9] Google Chrome Developers, "Messaging api - google chrome extensions," https://developer.chrome.com/docs/extensions/reference/messaging/, 2023.

[10] G. C. Developers. (2023) Runtime api - google chrome extensions. https://developer.chrome.com/docs/extensions/reference/runtime/.

[11] Google Chrome Developers, "Migrate to manifest v3," https://developer.chrome.com/docs/extensions/migrating/, 2024.

[12] G. C. Developers, "Migrating to service workers," https://developer.chrome.com/docs/extensions/migrating/to-service-workers/, 2024.

[13] Google Chrome Developers, "Storage - google chrome extensions," https://developer.chrome.com/docs/extensions/reference/storage/, 2023.

[14] G. C. Developers, "Chrome storage examples," https://developer.chrome.com/docs/extensions/reference/storage/#examples, 2023.

[15] M. Diamantaris, E. P. Papadopoulos, E. P. Markatos, S. Ioannidis, and J. Polakis, "Reaper: real-time app analysis for augmenting the android permission system," in *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, 2019, pp. 37–48.

[16] P. Carter, C. Mulliner, M. Lindorfer, W. Robertson, and E. Kirda, "Curiousdroid: automated user interface interaction for android application analysis sandboxes," in *Financial Cryptography and Data Security: 20th International Conference, FC 2016, Christ Church, Barbados, February 22–26, 2016, Revised Selected Papers 20*. Springer, 2017, pp. 231–249.

[17] "js-beautify," https://pypi.org/project/jsbeautifier/, 2009.

[18] "Esprima - ECMA Script parsing infrastructure for multipurpose analysis," https://esprima.org/, 2024.

[19] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "Dbscan revisited, revisited: why and how you should (still) use dbscan," *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 3, pp. 1–21, 2017.

[20] K. R. Shahapure and C. Nicholas, "Cluster quality analysis using silhouette score," in *2020 IEEE 7th international conference on data science and advanced analytics (DSAA)*. IEEE, 2020, pp. 747–748.

[21] K. Solomos, P. Ilia, N. Nikiforakis, and J. Polakis, "Escaping the confines of time: Continuous browser extension fingerprinting through ephemeral modifications," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22.

[22] W3C, "Mutation event types," 2000, https://www.w3.org/TR/DOM-Level-2-Events/events.html#Events-eventgroupings-mutationevents.

[23] Selenium, "Selenium is a suite of tools for automating web browsers." 2023, https://www.selenium.dev/.

[24] "PyAutoGUI : cross-platform GUI automation Python module." 2024, https://pyautogui.readthedocs.io/en/latest.

[25] S. Karami, F. Kalantari, M. Zaeifi, X. J. Maso, E. Trickel, P. Ilia, Y. Shoshitaishvili, A. Doupé, and J. Polakis, "Unleash the simulacrum: Shifting browser realities for robust {Extension-Fingerprinting} prevention," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 735–752.

[26] Scikit-learn developers, "scikit-learn Machine Learning in Python." 2024, https://scikit-learn.org/stable/.

[27] "PyPI JSON API client library." https://pypi.org/project/pypi-json/, 2024.

[28] Docker, "Accelerate how you build, share, and run modern applications." 2023, https://www.docker.com/.

[29] "Flask is a web framework, it's a python module that lets you develop web applications easily." https://flask.palletsprojects.com/en/2.3.x/, 2010–present.

[30] I. Sysoev, "Nginx," https://nginx.org/, 2004–present.

[31] E. Trickel, O. Starov, A. Kapravelos, N. Nikiforakis, and A. Doupé, "Everyone is different: Client-side diversification for defending against extension fingerprinting," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1679–1696. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/trickel

[32] S. Karami, F. Kalantari, M. Zaeifi, X. J. Maso, E. Trickel, P. Ilia, Y. Shoshitaishvili, A. Doupé, and J. Polakis, "Unleash the simulacrum: Shifting browser realities for robust extension-fingerprinting prevention," in *31th {USENIX} Security Symposium ({USENIX} Security 22)*, 2022.

[33] P. Eckersley, "How unique is your web browser?" in *Proceedings of the 10th International Conference on Privacy Enhancing Technologies*, ser. PETS'10, 2010.

[34] K. Mowery and H. Shacham, "Pixel perfect: Fingerprinting canvas in HTML5," in *Proceedings of W2SP 2012*, May 2012.

[35] M. Mulazzani, P. Reschl, M. Huber, M. Leithner, S. Schrittwieser, E. Weippl, and F. Wien, "Fast and reliable browser identification with javascript engine fingerprinting," in *Web 2.0 Workshop on Security and Privacy (W2SP)*, vol. 5, 2013.

[36] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel, "Fpdetective: dusting the web for fingerprinters," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 1129–1140.

[37] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *Proceedings of ACM CCS 2016*, 2016.

[38] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "Cookieless monster: Exploring the ecosystem of web-based device fingerprinting," in *IEEE Symposium on Security and Privacy*, 2013, pp. 541–555.

[39] A. Gómez-Boix, P. Laperdrix, and B. Baudry, "Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale," in *Proceedings of the 2018 world wide web conference*, 2018, pp. 309–318.

[40] A. Durey, P. Laperdrix, W. Rudametkin, and R. Rouvoy, "Fp-redemption: Studying browser fingerprinting adoption for the sake of web security," in *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2021.

[41] X. Lin, F. Araujo, T. Taylor, J. Jang, and J. Polakis, "Fashion faux pas: Implicit stylistic fingerprints for bypassing browsers' anti-fingerprinting defenses," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 987–1004.

[42] X. Lin, P. Ilia, S. Solanki, and J. Polakis, "Phish in sheep's clothing: Exploring the authentication pitfalls of browser fingerprinting," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1651–1668.

[43] S. Wu, P. Sun, Y. Zhao, and Y. Cao, "Him of many faces: Characterizing billion-scale adversarial and benign browser fingerprints on commercial websites," in *30th Annual Network and Distributed System Security Symposium, NDSS*, 2023.

[44] S. Li and Y. Cao, "Who touched my browser fingerprint? a large-scale measurement study and classification of fingerprint dynamics," in *Proceedings of the ACM Internet Measurement Conference*, 2020, pp. 370–385.

[45] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy, "Fp-stalker: Tracking browser fingerprint evolutions," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 728–741.

[46] O. Starov, P. Laperdrix, A. Kapravelos, and N. Nikiforakis, "Unnecessarily identifiable: Quantifying the fingerprintability of browser extensions due to bloat," in *The World Wide Web Conference*, 2019, pp. 3244–3250.

[47] I. Sanchez-Rola, I. Santos, and D. Balzarotti, "Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies," in *Proceedings of the 26rd USENIX Security Symposium (USENIX Security)*, August 2017.

[48] T. Van Goethem and W. 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} 17)*, 2017.

[49] K. Solomos, P. Ilia, S. Karami, N. Nikiforakis, and J. Polakis, "The dangers of human touch: Fingerprinting browser extensions through user actions," in *31th {USENIX} Security Symposium ({USENIX} Security 22)*, 2022.

[50] G. G. Gulyas, D. F. Somé, N. Bielova, and C. Castelluccia, "To extend or not to extend: on the uniqueness of browser extensions and web logins," in *Proceedings of the 2018 Workshop on Privacy in the Electronic Society*. ACM, 2018, pp. 14–27.

[51] G. C. Developers, "Manifest - web accessible resources," https://developer.chrome.com/docs/extensions/reference/manifest/web-accessible-resources, 2023.

[52] Chrome, "Manifest - web accessible resources," https://developer.chrome.com/docs/extensions/mv3/manifest/web_accessible_resources/, 2024.

[53] Firefox, "web_accessible_resources," https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/web_accessible_resources, 2023.

[54] A. Sjösten, S. Van Acker, P. Picazo-Sanchez, and A. Sabelfeld, "Latex gloves: Protecting browser extensions from probing and revelation attacks," in *26th Annual Network and Distributed System Security Symposium*. The Internet Society, 2019.

[55] E. Trickel, O. Starov, A. Kapravelos, N. Nikiforakis, and A. Doupé, "Everyone is different: Client-side diversification for defending against extension fingerprinting." in *USENIX Security Symposium*, 2019, pp. 1679–1696.