Escaping the Confines of Time: Continuous Browser Extension Fingerprinting Through Ephemeral Modifications

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

Panagiotis Ilia
University of Illinois at Chicago
Chicago, IL, USA
pilia@uic.edu

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

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

ABSTRACT

Browser fingerprinting continues to proliferate across the web. Critically, popular fingerprinting libraries have started incorporating extension-fingerprinting capabilities, thus exacerbating the privacy loss they can induce. In this paper we propose *continuous fingerprinting*, a novel extension fingerprinting technique that captures a critical dimension of extensions' functionality that allowed them to elude *all* prior behavior-based techniques. Specifically, we find that *ephemeral modifications* are prevalent in the extension ecosystem, effectively rendering such extensions invisible to prior approaches that are confined to analyzing snapshots that capture a single moment in time. Accordingly, we develop Chronos, a system that captures the modifications that occur throughout an extension's life cycle, enabling it to fingerprint extensions that make transient modifications that leave no visible traces at the end of execution. Specifically, our system creates behavioral signatures that capture nodes being added to or removed from the DOM, as well as changes being made to node attributes. Our extensive experimental evaluation highlights the inherent limits of prior snapshot-based approaches, as Chronos is able to identify 11,219 unique extensions, increasing coverage by 66.9% over the state of the art. Additionally, we find that our system captures a unique modification event (i.e., *mutation*) for 94% of the extensions, while also being able to resolve 97% of the signature collisions across extensions that affect existing snapshot-based approaches. Our study more accurately captures the extent of the privacy threat presented by extension fingerprinting, which warrants more attention by privacy-oriented browser vendors that, up to this point, have focused on deploying countermeasures against other browser fingerprinting vectors.

CCS CONCEPTS

• **Security and privacy → Browser security**.

KEYWORDS

Online Tracking, Browser Fingerprinting, Extension Fingerprinting

CCS '22, November 7–11, 2022, Los Angeles, CA, USA.

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3560576

ACM Reference Format:
Konstantinos Solomos, Panagiotis Ilia, Nick Nikiforakis, and Jason Polakis. 2022. 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 (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3548606.3560576

1 INTRODUCTION

Modern web browsers offer an expansive collection of features and capabilities for improving the user experience, while still allowing users to further expand browsers' functionality or personalize their experience by installing browser extensions. The prevalence of extensions is made evident by a report from Google stating that "nearly half of all Chrome desktop users actively use extensions" [4]. However, this personalization suffers from inherent privacy risks: (i) the list of installed extensions can augment the browser fingerprint that websites generate for a given device, (ii) the intended functionality of extensions can reveal sensitive or personal data about the user (e.g., religion, medical issues, and nationality) [22]. In other words, extension fingerprinting presents an *additional* form of privacy loss compared to "traditional" browser fingerprinting vectors.

Nonetheless, while other browser attributes and characteristics that contribute to browser fingerprints can be trivially obtained through dedicated JavaScript APIs, no such capability exists for obtaining the list of installed extensions. Instead, the presence of a given extension needs to be inferred through implicit techniques. In fact, in recent years the research community has demonstrated various techniques for achieving that goal. Early studies relied on detecting the presence of specific web-accessible resources [42], a technique which can be rendered ineffective by countermeasures deployed by certain browsers or proposed by the research community [41, 48]. A more robust approach relies on inferring the presence of extensions based on the side-effects of their executed functionality (i.e., modifying the page's DOM [22, 46] or altering the page's stylistic properties [28]). More importantly, while extension fingerprinting has mostly been confined to academic studies,[1] recent versions of `FingerprintJS` [24] (the most prevalent browser fingerprinting library) actually incorporate such capabilities for fingerprinting extensions based on traces found in the DOM. This move has pushed extension fingerprinting into the realm of real-world privacy threats that can affect users at a wide scale.

A core limitation of *all* prior studies that infer the presence of an extension by detecting side-effects caused by its execution (i.e., DOM changes), is that they ignore their execution *life cycle* and

[1] LinkedIn being the one notable exception [36].

only analyze a *single* snapshot (i.e., the DOM's state at a single moment in time). This has significant implications for a fingerprinting system's effectiveness as it will essentially be "blind" against any extensions that make ephemeral modifications (e.g., injecting and then removing a script). Additionally, the configuration of *when* to take that particular snapshot (e.g., prior work compares the DOM 10-15 seconds after the test page is loaded [22, 46]) as well as environmental aspects that can affect an extension's execution (e.g., the number of installed extensions, CPU load) can further impact such snapshot-based approaches.

In this paper we introduce the concept of *continuous fingerprinting*, a fundamentally different strategy that overcomes the aforementioned limitations, whereby the fingerprinting system captures the entire life cycle of extensions' execution. We implement this concept into Chronos,[2] a novel fingerprinting system that collects fine-grained information about *all* the changes that occur within the DOM, including ephemeral modifications (i.e., short-lived changes that do not leave permanent evidence behind). This enables our system to capture previously-undetectable behaviors that subvert all prior fingerprinting approaches which searched for behavioral fingerprints within a snapshot confined to a single moment in time. Since our technique necessitates changing the type of information used in signatures, we leverage the Mutation Observer [10] JavaScript interface and generate fine-grained signatures storing the order and the type of each modification as the extension introduced it. Moreover, by applying a set of optimization and compression techniques, our system generates signatures containing the required fine-grained information effectively.

We conduct a comprehensive experimental evaluation of Chronos, and demonstrate the effectiveness of continuous fingerprinting in uncovering "stealthy" extensions that exhibit ephemeral modifications. We find that such transient modifications are extremely prevalent, as Chronos is able to fingerprint 11,219 unique extensions, resulting in a 66.9% improvement over state-of-the-art DOM-based fingerprinting [22]. Our signatures are optimally constructed since 94% contain at least one unique modification that is adequate to distinguish them. Moreover, our system is highly accurate and efficient in a multi-extension environment since it has an average signature matching accuracy of 98%. At the same time, it can also perform the most demanding signature matching in less than 1.5 seconds.

Overall, our study demonstrates that prior techniques significantly undercount the threat of extension fingerprinting by missing 40% of the extensions detected by our system. Moreover, adoption by popular fingerprinting libraries and services will push extension-fingerprinting into the mainstream, further exacerbating the privacy risks demonstrated by the research community. We hope that our findings attract more attention from privacy-oriented browser vendors that are deploying defenses against general browser-fingerprinting techniques, and incentivize them to also explore countermeasures against extension fingerprinting.

In summary, we make the following research contributions:

- We propose *continuous fingerprinting*, a novel fingerprinting concept that overcomes the time-based confines of prior approaches and captures extensions' execution life cycles.
- We develop Chronos, a novel system that implements continuous fingerprinting. We explore multiple aspects of continuous fingerprinting and develop strategies for optimizing its performance both in terms of storage and generated network traffic, as well as detection accuracy.
- We experimentally evaluate Chronos and demonstrate that our approach outperforms the state-of-the-art fingerprinting technique as it enables the detection of a significant number of undetected extensions.

[2]Named after Chronos from Greek mythology, a deity that embodied the concept of sequential time and was associated with the duration of an individual's life cycle.

2 BACKGROUND AND THREAT MODEL

Here we provides pertinent background information on extensions and technical details relating to the techniques that we introduce.

Extension structure. Extensions are comprised of different components that implement the extension's functionalities and programmatic logic. The `Manifest` file allows developers to specify the background and content scripts, external pages, and permissions that enable extensions to achieve their desired functionality.

Background scripts. Typically, the extension's main logic is implemented in the background script using HTML and JavaScript. These scripts run as individual processes in the context of the browser and handle the majority of the functionality that content scripts cannot. Since they cannot access the page directly, they communicate with the other components (e.g., content scripts) through the `Messaging` API and fetch any resources or data required for their functionality.

Content scripts are the only scripts that are injected into the webpage and directly run on the page. Extensions use them to interact with and modify the page, while they communicate with the background script through browser APIs. These scripts are declared statically in the manifest under an entry that also defines the set of domains on which the content script will execute. In general, content scripts use DOM requests to control the page and can also inject other custom scripts or event listeners that listen for specific events.

Mutation observer interface. The concept of mutation observers was initially introduced by browsers in the early 2000s to allow developers to monitor DOM changes [53]. Even though it was not widely used initially, the API was later updated into a fine-grained JavaScript interface that monitors the DOM for alterations and modifications [10]. Developers can employ it in their web applications and use specific options that allow them to observe the DOM modifications that occur on the target elements, especially when dynamic changes occur due to users interacting with the page. Listing 1 shows an example of how the mutation observer can be used on a target DOM node.

The initial API call of `observe` configures the MutationObserver to begin receiving and logging notifications through the callback function when the DOM change is fired on the target element. The `options` object defines the type of mutations that are recorded through the mutation object and it includes:

- `subtree`. Monitors the entire DOM subtree of the nodes connected to `target`.
- `childList`. Monitors the target node for additions of new child nodes and removals of existing nodes.
- `attributes`. Monitors the changes to the value of attributes on the target node.

```
function callback(mutationList, observer) {
  mutationList.forEach( (mutation) => {
  switch(mutation.type) {
    case 'childList':
    /* One or more
          nodes have been added to and/or removed */
        break;
    case 'attributes':
    /* An attribute value changed. */
        break;}});}
const target = document.getElementById("myelement");
const observerOptions = {
  childList: true,
  attributes: true,
  subtree: false
}
observer = new MutationObserver(callback);
observer.observe(target, observerOptions);
```

Listing 1: Example of the initialization and usage of a mutation observer.

There are also other options depending on the expected type of modification. These include the `attributeFilter` which takes an array of specific attribute names to be observed, `characterData` which monitors the changes to the character data contained in the target node, and `attributeOldValue`, `characterDataOldValue` which store the previous data and attribute values. These options and configurations allow developers to customize the observer objects for detecting and logging all the changes that occur on a specific target node. As documented by developers [13], the usage of mutation events is highly efficient when expecting element or attribute changes under a specific time threshold, since this approach does not introduce the additional overhead that occurs with continuous DOM querying. Additionally, in certain cases they can even employ the mutation observer and omit using `eventListeners` when they are expecting a specific attribute or value change.

A `Mutation Record` represents an individual DOM mutation with specific properties, and inherits the properties from the DOM's `Node` object. Specifically, a Mutation Record stores the `type` and `target` of each node that the mutation affected. For the `childList` type, it also stores two separate lists of added and removed nodes for the defined target. The `attributes` type has additional entries that store among others the `attributeName` and `oldValue` of the altered attribute. Finally, each record also stores other DOM-related data including the `parent node`, first and last child node, `nextSibling` and `previousSibling` nodes and other node metadata including the `width/height` and `position`.

Motivation. `FingerprintJS` [24] is one of the most popular browser fingerprinting libraries that website vendors employ for multiple purposes, including bot detection and user identification. The library employs advanced browser fingerprinting APIs, such as fonts, canvas, and WebGL fingerprinting, to create the device's identifiers. In April 2021 the company announced that they incorporated extension fingerprinting, focusing on ad-blocker extensions as part of their fingerprinting vector [40]. As they specifically mentioned *"Our goal is to get as much information from ad blockers as possible to generate a fingerprint."* The library leverages DOM fingerprinting to detect the presence of one or multiple ad-blockers. For these extensions, they extract unique CSS selector elements and generate a set of keywords that could exist on the page due to the extension execution. Consequently, the library creates a new

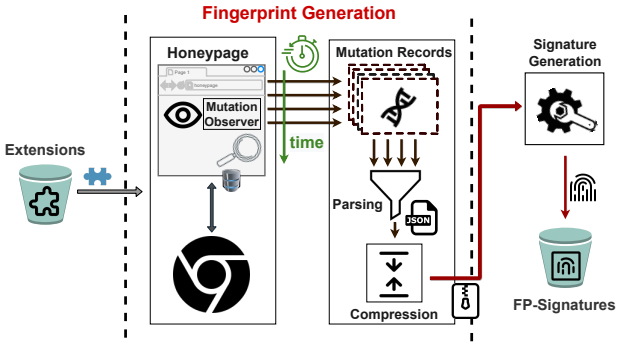


Figure 1: High level representation of our system Chronos.

HTML element (i.e., div tag) with the same values of the potentially blocked CSS selector and injects it into the page. After injection, it logs whether this element is present or blocked. This approach is a textbook example of DOM fingerprinting as introduced by prior research [22, 46], wherein a tracker attempts to fingerprint a user's extensions based on the modifications they introduce to the page.

Threat model. We follow the established threat model of prior research on extension fingerprinting and 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. The attacker's page leverages a mutation observer object that monitors the DOM tree for all types of modifications that occur during extensions' life cycles, and generates behavioral signatures for fingerprinting extensions.

3 SYSTEM DESIGN AND IMPLEMENTATION

In this section we provide details about our system's design and implementation. Figure 1 shows a high-level overview of our Chronos system. In the following, we provide more details about its building blocks, and argue for various design decisions taken while implementing our continuous fingerprinting mechanisms. We first present our approach that utilizes a `MutationObserver` for monitoring the page and capturing the changes that occur. We then discuss the characteristics of `MutationObserver` records and highlight the differences of this approach with the *state of the art* that relies on analyzing a snapshot of the page's DOM. Subsequently, we detail our methodology for constructing extension fingerprinting signatures based on the observed modifications.

3.1 Detecting DOM-based Modifications

To detect DOM-based modifications existing approaches [22, 46] capture a snapshot of the page's DOM, which contains the modifications made by extensions, and compare it with a baseline snapshot taken when visiting the website without any installed extensions (i.e., the original DOM). Based on the modifications that are present in the later snapshot, one can detect extensions the are installed.

This approach has two significant drawbacks that affect its effectiveness and accuracy in detecting installed extensions and distinguishing between their modifications. First, it detects that a new element is added to the page or that an existing one is removed, but it does not provide information about the existing elements' modified properties (e.g., height, width, style, and position). The state-of-the-art framework for DOM-based extension fingerprinting, `Carnus` [22], compares the document's `outerHTML` in the two snapshots and identifies text keywords that appear or disappear from the later snapshot, corresponding to the added and removed DOM elements respectively. However, while this allows it to detect that, for example, a `<div>` element was added to the page, it does not record its properties or whether it was modified multiple times. While a subset of this information could potentially be retrieved by repeatedly polling the DOM, such an approach would introduce considerable overhead while also being unable to capture all of the asynchronous modifications performed.

Second, the most significant drawback of snapshot-based approaches is that they can only observe the *cumulative* result of the modifications that took place prior to capturing the snapshot. As such, they miss extensions that alter the page (or the same page elements) in a similar way, since the snapshot will only include evidence of the last modifications that "overwrote" previously committed ones. Moreover, such approaches also miss extensions that perform *ephemeral* modifications (i.e., changes that reverse the effects of previously committed actions). For instance, we have observed extensions that add an element to the page and soon after remove it, or extensions that inject a `<script>` that removes itself after execution. Existing approaches that take a snapshot at a specific point in time will fail to detect such extensions, unless a snapshot happens to be taken at the exact moment in time where the modifications' side-effects are still present on the page. When considering the fact that the execution life cycle of different extensions will vary, and that browsers execute extensions sequentially when multiple extensions are present, it becomes obvious that approaches that rely on snapshots suffer from fundamental drawbacks.

To overcome these limitations and generate accurate fingerprinting signatures, we leverage the `MutationObserver` interface for *continuously* monitoring the page and collecting information about *all* the alterations that take place. While an approach that relies on capturing multiple snapshots of the page's DOM at specific time intervals (e.g., every few tens of milliseconds) is conceptually simpler, such a DOM-polling approach will impose prohibitively high overheads on both the page and the detection system. Furthermore, even with multiple frequent snapshots, there is no way to ensure that a single modification is captured in each snapshot. This motivates our design and necessitates utilizing the `MutationObserver` mechanism for detecting the modifications.

Honeypage. We follow the methodology of prior work [22] for exercising extensions and making them reveal their presence. Specifically, we use a website under our control (dubbed as *honeypage*) for identifying which modifications each extension introduces. To construct the fingerprinting signatures we visit the honeypage with a browser that has a single extension installed and wait for the honeypage to complete loading and the extension to run its functionality, while collecting all the information about modifications that occur which are recorded by the `MutationObserver`. For each extension we visit the honeypage three times so as to identify extensions that perform *different* modifications each time, and extensions' modifications that include *dynamic content* that changes in a *predictable* way across visits (e.g., including a timestamp).

To ensure a fair comparison of Chronos to the state of the art, we will use the honeypage and dataset from `Carnus` [22]. The honeypage contains a variety of textual and visual elements, media resources, and ad-fetching scripts used for triggering extensions and revealing their functionality. The only change in the honeypage used by our system is that we employ a `MutationObserver` to detect and record the modifications instead of capturing a single snapshot of the page's DOM after a predetermined amount of time.

Extension filtering. For our analysis we focus on extensions that run on all domains (i.e., they include the `<all_urls>` entries in their manifest), since they are activated and executed on any page without domain restrictions. This selection strategy allows us to accurately quantify the risk that extension fingerprinting poses to all users, as any website they visit can employ these techniques and fingerprint those extensions. Compared to the *state of the art*, although `Carnus` [22] did not perform such a filtering on their dataset but exercised all extensions, we consider our results directly comparable to `Carnus`. This is because `Carnus`' honeysites were situated on custom domain names thereby triggering only the extensions that are allowed to execute on arbitrary websites (i.e. the extensions that execute only on specific domains and websites would never be triggered by `Carnus`, even if the authors chose not to filter their dataset).

3.2 Recording Mutation Information

A behavioral modification can be either (i) the addition or removal of DOM elements and (ii) the alteration of existing elements' attributes. The included `MutationObserver` starts checking for changes as soon as the page is loaded and the JavaScript code starts executing. It detects changes as they occur, in an asynchronous fashion, and for each modification it returns a `MutationRecord`. To monitor all the behavioral modifications that occur in the DOM tree, we configure the mutation observer to target the DOM's root node (i.e., `document.documentElement` and `document.body`) and capture all the `childList` and `attribute` mutation types.

Since we are interested in the entire DOM, by monitoring these two mutation types we are able to capture fine-grained information about extension-originating modifications, without the need to monitor and capture mutation types and properties such as `characterData` and `subTree`. Even though this information is available to the mutation observer, we found that the knowledge about the addition/removal of nodes and attribute changes is sufficient for detecting extensions, and that we do not need to further collect information about the DOM structure and the nodes' hierarchy.

In general, the `mutationRecords` returned by the mutation observer contain a plethora of information, and a large number of entries, relating to each observed modification. The majority of this information, however, consists of properties shared among multiple mutations' nodes and thus not helpful for uniquely characterizing a mutation. We do not include those entries since they provide supplementary information to the `childList` and attribute types that we already leverage. In practice, we verified that if a mutation event is present, it either belongs to the `childList` or

attributes type mutation record. The mutation object's additional entries store redundant information and structural related data that do not enhance our signature extraction.

To that end, we parse the mutationRecords and consider only a few specific entries when constructing the fingerprinting signatures. In the case of mutations of the attributes type, we store the target node identifier and its outerHTML attribute (which includes the altered values) and ignore the attributes' individual entries that can also be found in the mutationRecord.

3.3 Fingerprint Generation

In the remainder of this section we describe our methodology for constructing the extensions' fingerprinting signatures and extracting the user's fingerprint when visiting the attacker's website.

To construct the fingerprinting signatures, we visit our honeypage with a browser that has one extension installed at a time, and the mutationObserver in the honeypage records information about the mutations that take place. We then parse the obtained mutationRecords and identify specific entries, based on each mutation's type, that we use to form our fingerprinting signature. A similar process is followed when a user visits the attacker's website (i.e., our honeypage in this instance). The mutationObserver in the page collects information about the modifications introduced by the installed extensions, which we process similarly to the signature generation and use as the user's fingerprint. Finally, in order to determine which extensions the user has installed, we compare and try to match the extensions' signatures from our database with the user's collected fingerprint.

Dynamic modifications. During the signature generation process we visit the honeypage three times for each extension, to identify those that perform (i) different modifications in each visit and (ii) modifications that include dynamic content. After an initial analysis of the obtained mutationRecords we identified four classes of dynamic behaviors that alter one or multiple elements in a mutation. Such mutations need to be identified and handled with caution, as their dynamic parts can result in the extensions' signatures not matching the user's fingerprint. The four classes of dynamic behaviors that we identified are: (i) jQuery injected elements, (ii) extension WAR URLs, (iii) URL query parameters, and (iv) timestamps.

For the first class, we observe that the jQuery library uses an internal library called Sizzle to perform CSS queries and modifications [32]. When the library is called, it injects a element into the DOM, which has a name of the form sizzle-ID, where the ID is a unique numeric string reflecting the type of the selection. When the library finishes its querying process, it then removes the element from the DOM. This identifier is dynamic and changes every time that the extension runs. To handle this case of dynamic elements we replace the identifier in the element's name with a value representing its size. For example the name sizzle-1649704082959 is transformed to sizzle-13. This approach allows us to remove the dynamic parts of a signature effectively without altering the mutation's static or immutable parts.

We follow a similar approach for handling the case of WAR URLs that are included in the mutations. Since an extension's UUID changes when multiple extensions are installed in the browser, we replace the UUID with the keyword ID when storing the mutation in the signature, without actually altering the path that the extension requests (i.e., chrome-extensions<ID>/<path>/<resources>). With regards to requested resources' URLs that are found in the DOM modifications, we observed that if an extension requests a resource from an external URL, either the URL or the query parameters might change across runs. This may be part of their intended behavior, or it might be affected by the state of the DOM (e.g., if an element is present, the extension fetches a different resource). If we observe that a URL is stable but its parameters vary, we replace these parameters with the keyword ID. For example, in the case of https://s3.amazon.com/content.js?rand=1234, we will store the URL https://s3.amazon.com/content.js?rand=**ID** in the signature. On the other hand, if the resources are dynamic and their URLs change in an unpredictable way, we replace the resources name with the keyword Resource without altering their paths. Finally, we apply the same approach for handling the case of dynamic timestamps and dates. Specifically, we omit the dynamic values and only store the mutation's static part in the fingerprinting signature.

We followed a continuous testing approach for developing this strategy, by verifying that all dynamic values are detected and handled accordingly. As our evaluation shows (§4), it is uncommon for extensions to introduce additional dynamic elements that alter signatures' structure and content and, thus, our heuristics are comprehensive. This process is straightforward and easily applicable, while also effectively handling the dynamic behavior of fingerprintable extensions. Chronos follows the same approach when replacing the dynamic values in the signature-generation phase and during the extraction of users' fingerprints.

Attribute types. The attributes type of mutation record stores the outerHTML of a style or element modification that was triggered by the extension. Depending on its target, this entry (i.e., body or page's element) either stores a specific modification or extracts the whole HTML object of the page. For this type we compare the outerHTML with the original page's DOM (i.e., when an extension is not installed) and we extract the specific attribute changes. This filtering optimizes the content of the mutation record since it only stores the required information and enhances the fine-grained fingerprint extraction process.

Fingerprint collection. Since mutation records are triggered asynchronously, we wait until all the mutation events fire before storing them in a JSON object. The key in the JSON object is the mutation record's identifier, which stores the order in which each event was fired. Each key's entries are a serialized nested object that stores the required information for each mutation type. For optimization purposes and reducing the network overhead, we also compress the JSON object before sending it back to the server for further analysis. For compression we use JavaScript's popular Pako [33] library.

Another crucial dimension for our system functionality pertains to *when* we collect the signature trace. Previous work [22, 46] reported using a hard threshold of 10-15 seconds, which was empirically measured as sufficient time for an extension to reveal itself. Since Chronos *continuously* collects DOM modifications that are triggered while being performed, we can pinpoint a more accurate threshold. After extensive experimentation we found that eight seconds are adequate for extensions to load and perform their intended functionality even when multiple extensions are present. Out of all the evaluated extensions, only 0.05% performed modifications after

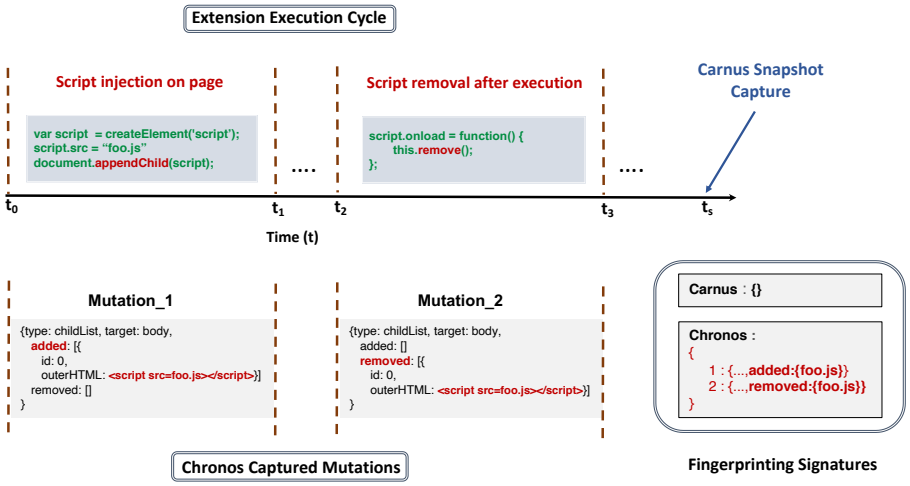


Figure 2: Extension execution timeline and continuous fingerprint generation.

the eight-second mark. Those are uncommon extensions that modify the DOM and interact with dynamic elements (e.g., animations, live data illustration) while active on the page. However, due to their continuous modification of the DOM we are still able to collect a representative set of mutations in under eight seconds. For the remainder of the paper we will use this threshold, unless stated otherwise.

Continuous fingerprinting timeline. To further detail the continuous fingerprint process, we illustrate the fingerprint generation timeline in Figure 2. In this scenario, an extension performs an ephemeral modification by injecting a script (`foo.js`), and removing it after its execution. At time t_0 the extension's content-script is running on the page and injects a `script` tag including the external script. This addition triggers the mutation observer since it detects the added script element on the page. At a later time t_2 the injected script removes itself since it finished its functionality, and the mutation observer is triggered again due to the removal of the previously added script. If we were following the traditional DOM fingerprinting snapshots like Carnus, we would not have taken a DOM snapshot during this time. Even if we were instructing Carnus to take a snapshot every few milliseconds (with all the noise and overhead that this entails), there would be no guarantee that our snapshots would have interleaved these addition/removal actions in a way that would have uncovered the ephemeral script. When Carnus takes its snapshot at a later time the extension has already finished its intended functionality and its fingerprint is empty, as if the extension had no visible side-effects on the DOM. On the other hand, Chronos' fingerprint includes one addition and one removal. This example illustrates the power of continuous fingerprinting compared to prior snapshot-based approaches in effectively uncovering extensions with ephemeral modifications.

3.4 Fingerprint Matching

Conceptually, detecting a single extension that is present in a user's browser is straightforward when leveraging the signatures generated by Chronos. In practice, users may have multiple extensions installed and several may be triggered on a given page. As such, we need to follow a different process for effectively fingerprinting multiple extensions. When a user visits the honeypage, we extract the fingerprint trace that we have to match with multiple entries accordingly. For this task, we design a matching algorithm that distinguishes the fingerprints of a larger fingerprint trace. A high level overview of our proposed algorithm is given in Algorithm 1.

Before applying any detection technique, we perform a set of preliminary processing and filtering. Our algorithm's input includes only the fingerprint trace and the set of all fingerprinted extensions. The first step is to select the subset of extensions with signature sizes smaller than the fingerprint trace's. This filtering is essential since the fingerprint trace's maximum size is the accumulated size of the target signatures while it also reduces the number of unnecessary comparisons by removing extensions with larger signatures, as they cannot be part of the fingerprint. Another characteristic of the signatures is that they can share identical mutation records, and thus, smaller signatures can form perfect subsets of larger ones. To avoid this type of mismatch when comparing signatures, we also sort the set of signatures in descending order based on their size. Moreover, signatures can have additional *unique* mutation records that are sufficient to detect the presence of each extension. We divide our target signature set into two smaller subsets to optimize our search process. The first one includes extensions with unique mutations, and the second one stores signatures formed by extensions with non-unique mutations. After applying this final filtering, we handle each subset of extensions individually.

In the first iteration (lines 2-8), we attempt to detect the signatures formed by unique mutations. For each signature, we compute the common set between its unique records and the mutations of the fingerprint trace. We successfully detect the extension if at least one mutation record matches the signature and the fingerprint trace. We then remove all of the signature's mutation records from the fingerprint trace, and we also store the extension's ID. We proceed with this process until the fingerprint trace is empty or until there are no other signatures left for comparison.

We follow a similar approach for the second part of our algorithm (lines 12-17). Since the search space is the set of signatures that *do not* have unique mutations, and thus we are not able to

Algorithm 1: Fingerprint matching process for detecting multiple extensions

Input: Fingerprint Trace, Fingerprint DB.
Output: List of detected extension IDs

```
ID_Vector =[]
signatures_set =Fingerprint DB {length ≤ size(fp_trace)}
unique_signatures = sorted{signatures_set{unique}}
non-unique_signatures = sorted{signatures_set{non-unique}}
foreach signature in unique_signatures do
    common= {signature.unique_records ∪ fp_trace}
    if common then
        fp_trace.remove(signature.mutations)
        ID_Vector.insert(signature.ID)
    end
end
foreach signature in non-unique_signatures do
    found = Boyer-Moore {signature.mutations, fp_trace }
    if found==True then
        fp_trace.remove(signature.mutations)
        ID_Vector.insert(signature.ID)
    end
end
return ID_Vector
```

distinguish them by just retrieving the shared mutations, we apply a different matching algorithm. Specifically, we build off of the Boyer-Moore string search algorithm [47]. The main idea behind this algorithm is the following: at first, it processes the target string and creates indexes to store the position of each character. Then it compares each pattern's character (starting from the end) to find a word or the same characters in the target string. When there is a mismatch, the search slides to the next matching position in the pattern using the precomputed index value. In our case the target string is the fingerprint trace, while the pattern is the signature we are looking for. We iterate through the set of signatures with common mutations, and for each signature we apply the algorithm until there is a match. For the matching signatures we remove their mutations from the fingerprint trace and store them accordingly. As we show in Section 4, this algorithm is efficient and effective at matching and detecting fingerprints.

In general, the aforementioned matching process that leverages both the "direct" and Boyer-Moore algorithms is effective in distinguishing unique signatures. While the Boyer-Moore algorithm can also be used in a standalone fashion with similar accuracy, our proposed solution that combines them both remains highly accurate while reducing overhead, as we show in our evaluation.

4 EXPERIMENTAL EVALUATION

In this section we experimentally evaluate Chronos' extension fingerprinting capabilities. For our analysis we use two datasets:

- Ext_1: This dataset contains the extensions used by Carnus [22]; it was collected in March 2018 and it contains 102,482 extensions. After applying the domain filtering rules and omitting the extensions that do not run on every domain, we are left with 27,342 extensions.
- Ext_2: To perform a fine-grained analysis of extension behavior across time, we collected a fresh snapshot of the Chrome Webstore in December 2021. To avoid any bias from the same extensions being in two different datasets, we omit any extensions already present in Ext_1 and only collect new versions of those extensions as well as new extensions. This dataset contains 11,140 extensions, with 3,144 being new versions of extensions from Ext_1.

Table 1: Number of extensions detected in each dataset.

Dataset	Extensions	Fingerprintable (%)
Ext_1	27,342	8,385 (30.66%)
Ext_2	11,140	3,865 (34.69%)
Total (all extension versions)		12,251
Total (unique extensions)		11,219

Experimental setup. We first deploy our honeysite in a popular, widely-used web hosting service to perform our experiments. We leverage Selenium [39] for orchestrating and controlling the browsers that act as desktop users that visit the honeysite with a specific extension installed. To increase the efficiency of our experiments, we build our framework into a Docker Container [9], that allows us to run multiple browsers with different sets of installed extensions in parallel. For all the experiments, we use an off-the-shelf desktop machine with a 6-core Intel Core i7-8700, 32GB of RAM, connected to our university's network. To reduce potential extension failures due to mismatched browser environments, for each dataset, we used the latest version of Google Chrome as well as one that was contemporary to the time period of each dataset [7] (i.e., versions: 73.0.3683.68 and 96.0.4664).

Overview. In Table 1 we present a summary of the detected extensions in our datasets. For the first and oldest dataset, we are able to uniquely fingerprint ≈31% of the extensions. Similarly, the percentage for the most recent dataset is ≈35%. In general, our average detection percentage is strictly better than any prior DOM-fingerprinting mechanism, while also providing a lower bound of the fingerprintability of the extension ecosystem. These detections reflect the inherent behavior of extensions interacting and modifying the DOM. However, extensions are complex and multi-dimensional components that provide various capabilities that may not always result in DOM modifications (e.g., they may employ the browsers' popup windows), which results in them not being fingerprintable by *any* DOM-based detection system. Moreover, extensions may also expect different *input* values, such as specific content in the honeypage, or user interactions [44], in order to trigger their DOM-modification logic.

Comparison to prior work. To gain insights regarding the effectiveness of our approach that relies on *continuously* fingerprinting browser extensions, we compare our findings with the state-of-the-art DOM-based extension fingerprinting system, Carnus [22]. In general, we differentiate the extensions that generate a non-unique signature and the extensions that have unique signature fingerprints and can be directly detected by our system. For the remainder

Table 2: Detected extensions and signature collisions of Carnus [22] and Chronos.

Dataset	Carnus		Chronos			Fingerprintable Extensions	
	Detections	Collisions	Detections	Collisions	Resolved Collisions (Carnus)*	Carnus	Chronos
Ext_1	6,965	1,521 (21.84%)	11,036	2,651 (24.02%)	1,481	5,444	8,385
Ext_2	2,184	287 (13.14%)	4,369	504 (11.54%)	204	1,897	3,865

*Collisions between Carnus' signatures that are resolved with Chronos' signatures.

of our analysis, we refer to the extracted signatures as `detections` and the unique signatures as `fingerprintable extensions`.

For a fair and straightforward comparison with our system, we run Carnus on both datasets. Since Carnus incorporates additional detection mechanisms (e.g., WAR, intra/inter communication) we only report the DOM-based behavioral detections. A breakdown of the results is provided in Table 2 with Carnus detecting 6,965 extensions in the Ext_1 dataset and 2,184 in the Ext_2 dataset. However, these detections are not unique due to signature collisions, i.e., signatures with the same functionality modifying the same DOM elements and, thus, we omit those collisions from the resulting fingerprint signatures. We find that 5,444 extensions are uniquely fingerprintable in the Ext_1 dataset and 1,897 in the Ext_2 dataset. The number we report for the oldest dataset is different than the number reported in [22] since the authors followed an approach that allowed mismatches for fingerprints of large size, which included 349 additional extensions with colliding signatures in their final set of fingerprintable extensions. Following the same principle, we apply our collision approach so as to directly compare the number of unique fingerprintable extensions. As we detail later, since our signatures are fine-grained and contain different information when compared to traditional DOM signatures, we do not include colliding signatures in the unique fingerprintable set of extensions.

Regarding Chronos' capabilities, we find that out of the 11,036 detections of Ext_1, the 8,385 are *uniquely* fingerprintable, while from the 4,369 detections of Ext_2, 3,865 are also uniquely fingerprintable. As expected, our system fingerprints *all* of the extensions that Carnus can fingerprint in both datasets. When comparing directly to Carnus and its original dataset, we are able to fingerprint 2,941 additional extensions that Carnus misses due to the snapshot-based observation of the DOM. Similarly, 1,968 extensions (50%) from Ext_2 can only be fingerprinted by Chronos, due to its ability to capture all modifications that occur within the DOM, even if those modifications are ephemeral and their traces are erased by subsequent modifications. In total, we are able to uniquely fingerprint 4,546 extensions in Ext_1 and Ext_2 that the state-of-the-art approach would miss. In general, we verify the efficacy of our continuous fingerprinting approach since we detect ephemeral behaviors of extensions (e.g., an injection and removal of a script or short-lived modifications) that are not detectable by the previous approaches.

Ephemeral modifications. To better understand the behavior of the extensions that can only be fingerprinted by our system, we further analyze their signatures. We find that all the 4,546 extensions include *at least* one addition *and* one removal of a page element. This specific behavior is not something one might expect from extensions; instead one would expect extensions to simply perform a consistent or immutable set of modifications on the page. This expectation is what drove prior fingerprinting strategies. However, based on our analysis, we find that this behavior is common for extensions that require information about the DOM's state or the browser's state. We discovered that extensions often inject an element, which could be a simple `div` tag, or a standalone script in order to access those properties. In fact, we find that the majority of the extensions (≈85%) inject a `div` tag which is required for the functionality of the `jQuery` library. Specifically, when the extension is present, the library injects a `sizzle` identifier, which is the internal component that allows the library to activate its CSS selectors and perform queries on the DOM [32, 34]. Once the querying is finished, the library removes the tag element and any other modification and may proceed with additional functionality. This behavior is consistent across different jQuery library versions that employ this type of environment-testing mechanisms.

Our analysis reveals similar behaviors for the remaining extensions (≈15%). Extensions tend to inject a script directly in the page or under an `iframe` that accesses the browser's values and variables. In the simplest scenario an extension injects a script that verifies that JavaScript is enabled. We also found cases of extensions reading the local storage and looking for a specific type of stored variables (e.g., whether the page contains a type of resource). When they find the required queried elements, they send a message to the extension's components (e.g., background script) and remove the injected script. This behavior is exemplified by the popular "Adobe Reader" extension (over 10M users), as it injects a script for identifying whether a PDF document is present in the browser. Moreover, similar behaviors include script injections for detecting if another ad-blocking extension is present. Specifically the "SpeedTest" extension, with over 2M users, injects a script that queries the DOM for the existence of an ad blocker since this information is required for its intended functionality. The aforementioned diversity of extension behaviors was only uncovered due to our continuous fingerprinting methodology, and was overlooked by traditional DOM-fingerprinting systems. Our approach is uniquely suitable for detecting such unpredictable behaviors and generating the appropriate signatures containing all of the ephemeral modifications.

4.1 Signature Stability, Size & Characteristics

Signature collisions. Next we analyze the identical signatures that Chronos detected and compare our results to Carnus. In Carnus-style snapshots, multiple extensions may generate the same signature due to similar behaviors. By leveraging mutation observers, we are able to resolve most collisions as our signatures are formed by continuous modifications and contain comprehensive fine-grained execution information. Specifically, using our technique our system resolves ≈97% and ≈71% of the collisions that affect Carnus in each

dataset, respectively. This highlights that the signatures generated using mutation records overcome the limitations of traditional DOM signatures. Nonetheless, in our approach, there are also extensions with identical behaviors that generate the same signature. We note that our dynamic identifier-replacement heuristics can result in collisions, but they are necessary for enabling Chronos to uniquely fingerprint extensions, since 5,050 fingerprints contain at least one mutation record with dynamic content which we would miss if we did not apply the dynamic heuristics. Regarding the collisions, out of 2,651 in Ext_1, 89% exist due to shared `jQuery` libraries. Also, 6% of the collisions are generated from extensions that perform *identical* functionality and are published under different identifiers by the *same* developer (e.g.,"One-Click Summarizer" and "One Click Reader"). For the remaining ≈5% of the extensions, we observe that the collision occurs due to extensions offering similar functionality and employing the same JavaScript interfaces and public libraries (e.g., for VoIP and Remote Control functionality).For the Ext_2 dataset, the distribution is similar, with 90% of the collisions generated by the jQuery library, ≈6.5% and ≈3.5% due to the same developer and functionality respectively.

Signature stability. To understand the stability of Chronos-derived signatures between different runs of the same extension, we calculate the number of mutations contained in each of the three fingerprints generated for each extension (i.e., from the three executions of each extension). Interestingly, we find that 99.5% of the signatures have the same number of mutations across runs. The remaining 0.5% represents highly dynamic extensions whose modifications are not deterministic and, thus, the fingerprint's size varies.

Apart from the size, signatures can also be volatile since mutation signatures can have a dynamic mutation or dynamic parts in a mutation record, as described in §3.3. Even if a signature has identical records across runs, parts of the signature might differ due to this dynamic behavior. Also, different signatures may share the same mutations due to common libraries. To quantify how these behaviors affect the fingerprinting process, we measure the number of signatures with at least one unique mutation. We find that from the total 11,219 unique extensions that Chronos detects across datasets, 10,555 (94%) have at least one unique mutation record. Since a mutation record stores the outerHTML of each modification, it has the potential of being unique. Even if the actions of adding/removing nodes are common across extensions, the type and the content of the modification itself can be unique based on the extension's purpose and functionality. As we discuss later in this section, this characteristic has significant implications, as our system is able to distinguish an extension's signature by just identifying a single unique mutation.

Signature statistics. In Figure 3 we present statistics regarding the signatures' sizes and structure. Figure 3a depicts the number of mutation records per signature. We find that 50% of the signatures contain less than 10 records, while only 5% of the signatures contain at least 50 mutation records, with the larger signatures storing up to 1,000 records. This trend captures the overall extension ecosystem's behavior, where the majority of the extensions perform a specific set of deterministic modifications. In contrast, only a few extensions have highly dynamic and elaborate behavior that triggers multiple mutation events. For instance, the popular weather forecast extension "Forecast Fox" modifies the DOM to create numerous elements with a graphical UI and real-time information.

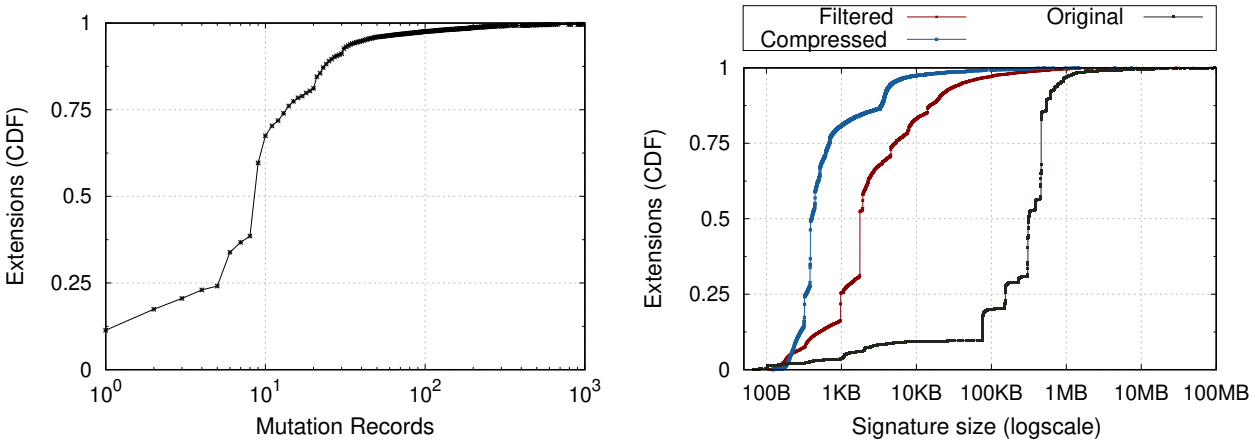
In Figure 3b we compare the original size of the mutation observer's object (i.e., the total number of entries included in the object), the filtered signature entries that Chronos uses (i.e., outerHTML and target entries), and the compressed (and filtered) signature size that is sent by the client's device back to the server for storage. Here we present the original mutation observer's object size only for completeness; in practice we never collect this object since the website directly applies our filtering strategy during the fingerprint's generation. As can be seen, our strategy of selecting only the mutation that holds crucial information is highly efficient since the final signature's size is 99.5% smaller than the original object. This confirms that our signature-generation strategy is efficient while still retaining precise information regarding the modifications that occur over time, which would be missed if we followed a snapshot-based approach.

Moreover, for the filtered signatures that form our datasets, we find that more than half require less than 1.5 KB of storage while less than 3% require 100 KB or more. Similar to Figure 3a, the extensions that perform multiple dynamic non-deterministic modifications are those that are more demanding in terms of storage. Since we compress signatures in our experimental setup, we observe an additional ≈75% size reduction for half of the extensions. These numbers indicate that our signature generation and collection is highly efficient since less than 75 KB of compressed data is transferred over the web during signature collection. Considering that users may be using smartphones and connecting over limited data plans, a naive approach of collecting and transferring all mutation observer records as they occur would have been prohibitive.

4.2 Longitudinal Analysis & Categorization

Extension types & popularity. We classify the fingerprintable extensions based on their type, as provided by the Chrome extension store. The most prevalent category is that of "Productivity" with ≈35% of Ext_1 and ≈45% of Ext_2 belonging to this category. Moreover, we also compute the relative popularity of the detected extensions, based on the number of downloads on Chrome's Webstore. Almost half of the extensions fingerprinted in both datasets have more than 100 users, while ≈10% of the extensions have more than 10,000 users.

Longitudinal analysis. As we mentioned earlier, Ext_2 contains 3,144 extensions that are newer versions of extensions from Ext_1. Out of those, 1,032 (32.8%) remain detectable across datasets (i.e., they were fingerprintable in the older dataset and remained fingerprintable in the new one). This indicates that certain extensions remain fingerprintable over time (≈4 years) even if they change their intended functionality or aspects of their behavior. We also find that from the initially fingerprintable extensions, 144 (12.2%) stop being fingerprintable. While not as prevalent, this also reflects how the extension ecosystem can evolve, since certain extensions may significantly change their functionality over time or offer the same features using different browser mechanisms (e.g., browser-popup windows). While this is an interesting trend, we have no means of assessing whether developers made these changes for the express purpose of making their extensions non-fingerprintable. Finally, 12.5%



(a) Signature's number of records per extension.

(b) Extensions' fingerprint sizes.

Figure 3: Number of mutation records per signature and signature size for the total number of detected extensions.

of the extensions that were not fingerprintable in the older dataset are indeed fingerprintable in the new dataset. In general, this experiment highlights that whenever an extension is updated, where developers might perform multiple modifications to the code and the functionality, fingerprinting systems should repeat their analysis.

To further quantify these behaviors and better understand the underlying trends, we compute the signature sizes of the two fingerprints for the 1,032 extensions that are fingerprintable across versions. We find that 697 extensions (67.5%) did not alter their behavior and their fingerprinting signature was identical. Moreover, for 127 extensions (12.7%), the most recent signature is shorter and stores fewer mutation records, while the size is greater for 20% of the signatures. We also explore if the updates alter the signature's uniqueness and compute the number of extensions that include at least one unique mutation record. In the older dataset, 1,014 extensions (≈98.2%) of the common fingerprintable extensions contain unique mutations, while in the most recent dataset that drops to 97.33%. Despite this slight decline, the rate is relatively stable over time. We conclude that, apart from the expected shift in the signatures due to updates, this trend also reflects that JavaScript libraries and APIs are constantly updated. Accordingly, developers adapt to these new changes by altering the extension's behavioral signatures, which Chronos is still able to detect.

4.3 Multi-Extension Fingerprinting

Here we evaluate Chronos' capabilities under a realistic deployment scenario, where multiple extensions are simultaneously installed in a user's browser. First, we perform a set of preliminary experiments to understand the behavior of extensions when they run in a multi-extension environment, and then perform a large-scale experiment to quantify our system's accuracy in a multi-extension setup.

Execution timeline. At first, we investigate how multiple extensions interact with each other, the resulting order of their execution, and how signatures are affected. The *exact* execution order is not a significant factor in our system since we are mainly interested in extracting the extension's fingerprint and identifying potential changes due to co-interference (i.e., one extension's changes affecting or prohibiting another extension's actions). For this experiment, we randomly choose extensions and form different subsets containing up to ten extensions. For each set, we install the extensions, visit the honeysite and collect the fingerprint traces. To generate an adequate number of measurements, we run this experiment 100 times for each independent set. We observe that the execution order is always identical (i.e., extension A runs before extension B), and the execution is deterministic. This is in line with prior observations by Picazo-Sanchez et al. [35] who reported that browsers execute extensions sequentially and that the execution order is based on the installation date of each extension (i.e., the extension installed first is executed first). However, we find that currently the execution order is based on the alphanumeric order of extensions' UUID (e.g., the extension with UUID "abc" will run before the extension with UUID "xyz"), which could potentially be attributed to a change in Chrome's handling of extensions. The finding that extensions' execution is sequential is crucial for our analysis, as it results in extensions' signatures that contain the same number of mutation records.

We also observe that execution interruption between extensions is not frequent, and it only occurs when extensions are requesting and fetching external resources. In practice, this type of execution "interleaving" occurs when extension A starts its life cycle and then requests an external resource; during this idle time the browser will start the execution of the following extension. This scenario creates inconsistencies when multiple highly-dynamic and network-related events of different extensions run in parallel and introduce overhead over the browser. However, as we already reported in §4.1, the majority of signatures are relatively short and stable, and thus even if interleaving occurs it does not alter the signatures' trace.

Identical modifications. Another experimental aspect that is crucial for matching fingerprints is whether different extensions trigger identical modifications in parallel. In this scenario, multiple extensions will attempt to alter the same page element or resource, resulting in a form of "race condition". This interference may affect the value stored in the mutation record as there are three potential

Table 3: Chronos' accuracy in multi-extension settings.

	2	3	4	5	6	7	8	9	10
TP (%)	98.33	97.4	96.7	98.21	96.5	98.32	95.84	97.48	98.25
FN (%)	1.67	2.6	3.3	1.79	3.5	1.68	4.16	2.52	1.75
F1 (%)	99.16	98.68	98.32	99.10	98.22	99.15	97.88	98.72	99.12

outcomes: (i) the last extension will perform the same type of modification and "overwrite" the previous modification, (ii) the extension will not perform any modification, or (iii) the extension will perform a *different* and non-fingerprinted modification according to its functionality. In the case where no modification or different modification occurs, the `childList` type mutation records will be affected since they store the information of added and removed nodes, and thus potential mismatches will be created. However, this behavior will not affect the `attribute` type of mutations since we hold the information of each attribute change individually, per mutation record. If an attribute modification is not present or additional attributes have changed, we can still identify the mutation record uniquely.

In order to handle these mismatches in the attribute types, and to further quantify this behavior, we run an experiment with extensions that include at least one attribute mutation record and capture how the mutation record is altered due to co-interference. Similar to the previous setup we install sets of up to ten extensions, selecting random and different extensions every time, and run each combination 500 times. We find that even when co-interference alters the content of the mutation records, the attribute mutation record remains at least *80%* identical to the original signature's mutation record. This result implies that even if the extension does not perform one of the attribute modifications since another extension has already completed it, the rest of the attribute modifications will occur normally. We use this threshold for the rest of our analysis when we refer to the matching algorithm process

Multi-extension fingerprint. Using the aforementioned insights, we setup a large-scale experiment for evaluating Chronos' accuracy and efficacy in detecting multiple extensions present in the same browser. We randomly select N fingerprintable extensions (sets of 2 up to 10) and install them in the same browser. We focus on fingerprintable extensions, since non-fingerprintable extensions do not perform page modifications and the execution trace would not change. If we included non-fingerprintable extensions, it would artificially inflate our reported accuracy, due to the lower likelihood of extension co-interference. We visit the honeypage 100 times for each set and compute each run's scores independently. For the matching task, we attempt to match a given fingerprint trace with multiple signatures, using the matching algorithm that we introduced in §3.4. When comparing attribute mutation records we use the 80% similarity threshold. Table 3 presents the results of our evaluation. For our analysis, False Positives reflect those extension signatures that are not actually in the fingerprint trace but are misclassified as detected by our system. Conversely, False Negatives are those signatures that are present in the fingerprint trace but our system fails to detect them. Importantly we note that we *do not have any* False Positives since our signature's structure and the matching algorithm do not generate mismatches. Contrastingly, Carnus suffered from 0.5-7.25% false positive rates [22]. Our system's accuracy is a direct result of our finding that 94% of the signatures contain unique mutations, as outlined in §4.1.

Overall, Chronos is highly accurate as it identifies 96–99% of the installed extensions. For the remaining cases, we have identified two different behaviors that lead to mismatches (FN ≈ 1-4%). In the first case, the modification that the extension attempts to execute has already been performed by another extension, and thus it does not perform it. For example, suppose extension A changes the page's background color from white to red, and extension B accesses the background color's value to inject a new element. When extension B reads the page's background variable, it is different from the predefined (white), and in that case it will not perform this specific modification. The second type of mismatch includes those extensions that perform a `dynamic` modification that has not been captured during the extension fingerprinting phase. Interestingly, we found cases of extensions explicitly injecting debugging messages (e.g., *"Something is not right at this moment! Please try again after some time."*) instead of completing their intended modification. This behavior could also potentially be used as part of an extension's fingerprint since the absence of a specific modification and a "debugging" modification could reveal the presence of an extension. While these mismatches are part of the extension's capabilities, since we cannot currently predict how or when the extension will perform a different modification due to the interference from a different extension, we consider this an interesting future direction. Nonetheless, despite the potential for co-interference, our experiments demonstrate that Chronos is highly accurate in a realistic deployment scenario and always achieves an F1-score higher than 98%.

System performance. As we have detailed previously, we use a threshold of eight seconds during the extension fingerprinting phase and when fingerprinting users' extensions in real-time. This design decision is crucial since it minimizes the time that the user is required to stay on the website. By design, we offload all the processing of the mutations and detection to the server. In practice, the website will generate the fingerprinting trace after eight seconds, compresses it, and send it back to the server. The server is then responsible for decompressing the fingerprint and extracting the signature. For each signature, it processes the attributes elements, dispatches the modifications, and finally, handles the removal and replacement of the dynamic parts of the signatures.

To test the server-side computation overhead, we employ the same experimental setup for matching fingerprints from multiple extensions; we measure the time required for the server to process the fingerprint trace and the algorithm to match the trace to the stored signatures. We run this experiment 100 times for N=10 installed extensions to collect a representative number of measurements that provide an upper bound (i.e., the worst-case scenario). We find that, on average, the server requires 1.5 seconds to detect 10 installed extensions (`stdev 0.8`) with a median of 0.25 seconds. This result is expected if we take into consideration the signatures sizes (§4.1) and the fact that under this experimental scenario, the average decompressed fingerprint trace size is only 0.6 MB. These numbers represent the `upper-bound` of computational overhead since, in a realistic scenario where the user has a smaller number of fingerprintable extensions generating mutations, the signatures will be significantly smaller. In general, while we find that Chronos is highly efficient, it can be further optimized if an attacker targets

a specific subset of extension or they employ high-end machines with multi-threading components for processing and decompressing, and dedicated GPUs for more efficient pattern matching.

4.4 Preventing DOM-based Fingerprinting

DOM-based fingerprinting is particularly robust against defenses since it essentially captures execution artifacts that are inherent to a given extension's functionality and, thus, not trivial to prevent. Nonetheless, certain recent studies have proposed DOM-fingerprinting countermeasures and defenses to protect user's privacy and mitigate such attacks [23, 48]. In more detail, Trickel et al. [48] proposed CloakX, a system that aims to broadly defend against extension fingerprinting. Their system employs a set of heuristics that randomize the `ID` and `class` HTML attributes. Since extensions can inject uniquely identifiable elements, this strategy attempts to hide the existence of such elements and, thus, prevent behavior-based detection. CloakX also randomizes the Web Accessible Resources paths, and also injects random tags, attributes, and custom elements to make fingerprints noisy and non-uniquely distinguishable. Since the randomization of `ID` and `Class` elements as well as the WAR paths when included in a WAR URL (see §3) can potentially affect our signatures, we quantify the effect of this countermeasure against our system.

To replicate the countermeasure's effect, we follow an approach similar to our dynamic identifier replacement. Specifically, we replace all `ID` and `Class` elements with the keyword "`Random`" and also replace all references to the `chrome-extension://UID/PATH` with `<Random-Path>`. We apply the aforementioned randomization heuristics on the entirety of 11,219 unique detections across the datasets. The countermeasure results in 124 ($\approx$ 1%) fingerprints becoming non-unique and generating collisions, with no impact on the remaining fingerprints. Moreover, 92% of the fingerprints still have at least one unique mutation, hinting at a minor decline (2%) compared to the original signatures. Regarding the injection of random elements and tags, Chronos is effective at identifying dynamic signatures and filtering out those mutations that are not present or stable. Even if CloakX was able to inject random elements in *every* extension signature, we would still be able to fingerprint the majority of extensions based on their unique mutations.

In a different direction, the recent work by Karami et al. [23] proposes a solution specifically for DOM-fingerprinting. Their approach separates the DOM that the extensions interact with and the DOM that the page's scripts access. Their implementation intercepts various JavaScript APIs and function calls, in order to control which information is available to the original and the "parallel" DOM. One of the APIs they target is the `Mutation Observer` API, which is integral to our system. Their countermeasure affects our attack as our system would be unable to perform continuous fingerprinting through the Mutation Observer API.

Overall, the first proposed solution of CloakX does not affect the efficacy and efficiency of Chronos, while the second approach of Simulacrum impacts our system. Both of these defense mechanisms are significant contributions to the extension fingerprinting ecosystem since they provide solutions for better protecting users. As neither one has so far been adopted by browsers, we argue that our work highlights the importance of browsers adopting these defenses and further exploring this space. We expand this discussion of countermeasures and other complementary defenses in §6.

5 DISCUSSION

Our work is the first one to observe that DOM-based extension-fingerprinting is not inherently limited to before/after DOM snapshots, that prior work has relied upon [22, 28, 45, 46]. Instead, using modern browser APIs such as the *mutationObserver* [10], trackers can be alerted of any change in a page's DOM and match these changes against offline-curated, extension-signature databases. In this way, trackers can fingerprint extensions that present ephemeral DOM changes, as well as those with colliding signatures under more coarse-grained fingerprinting schemes. Using this notion of continuous fingerprinting in our Chronos system, we show that trackers can uncover thousands of additional extensions that were invisible to prior state-of-the-art DOM fingerprinting techniques. These findings underline the privacy risks of extension fingerprinting, which can be used not only as an additional source of user-identifying entropy (such as the users' screen properties and how their graphics cards renders complex images) but also as a means of uncovering sensitive socioeconomic information about users based on the extensions that they *chose* to install [22].

If we take a step back, we can observe that privacy is becoming a mainstream concern for browser vendors. Browsers like Brave and Mozilla Firefox are constantly adding privacy-enhancing mechanisms in their browsers, ranging from built-in blocklists for stopping advertising and tracking scripts, to randomizing the values of fingerprintable APIs (e.g., Canvas) and making certain mechanisms entirely unavailable to scripts (e.g., APIs related to battery status) [5, 16]. Even Google Chrome (which traditionally did not implement extra privacy mechanisms) has been evaluating novel privacy-preserving user-targeting techniques for advertising, through its Privacy Sandbox program [18].

Despite all this progress, extension fingerprinting has not attracted the attention of browser vendors in a way that would protect extension users by default. With the recent addition of extension-fingerprinting logic to the web's most popular browser-fingerprinting library [24], extension fingerprinting is now becoming available at a global scale, in the same way that canvas fingerprinting is currently available. Moreover, the results of this paper show that the threat of extension fingerprinting was understated in prior work, with more extensions being fingerprintable than was originally thought. We therefore argue that it is imperative that browser vendors include extension fingerprinting in their threat modeling and start evaluating possible anti-fingerprinting techniques already proposed by academia, ranging from the further randomization of extensions [48], to strict access control where a user's decisions about the context in which extensions should and should not run, supersede those of extension authors [41, 45]. We hope that this paper serves as additional motivation to kickstart this process.

6 RELATED WORK

Since the work of Peter Eckersley [12], who was the first to demonstrate in 2010 with the Panopticlick experiment that fingerprints can be used to uniquely identify a user's device and that they

can be easily collected at scale, browser fingerprinting has become notoriously prevalent on the web [1, 8, 21]. It is mainly used for user identification and tracking [1, 14, 51], but lately this technique has also been deployed for other purposes, such as bot detection [11, 20, 52] and augmenting authentication mechanisms [3, 11, 25, 29]. Many past works have proposed techniques for expanding the fingerprinting surface, designed countermeasures and mechanisms for detecting fingerprinters and preventing them from tracking users, as well as conducted studies in order to measure the prevalence and effectiveness of such techniques [2, 6, 8, 14, 15, 17, 21, 26, 27, 30, 31, 37, 43, 50, 51].

A fingerprinting vector that has become prominent in recent years is the detection of users' installed browser extensions. Early studies in this area relied on detecting the presence of specific web-accessible resources (WAR) that extensions expose [19, 42]. In another line of work, Sanchez-Rola et al. [38] and Van Goethem and Joosen [49] proposed a timing side-channel attack that exploits browsers' access control mechanism for extensions' resources in order to infer their presence. However, the countermeasures deployed by certain browsers or proposed by the research community [41, 48] have rendered these techniques ineffective.

More recently the research community proposed techniques that infer the presence of extensions by identifying the side-effects of their executed functionality, such as detecting the modifications to the page [22, 46] or changes to its stylistic properties [28], as well as monitoring the exchanged messages and the resources they fetch [22, 45]. The work of Starov and Nikiforakis [46] was the first to demonstrate that extension fingerprinting is feasible through the detection of their DOM-based modifications. They built `XHound`, a framework that patches the extensions' source code to place hooks into the functions that extensions use to query the DOM elements, and dynamically create these elements on-the-fly in their honeypages, aiming to trigger extensions' functionality. Karami et al. [22] built the `Carnus` framework and used it to conduct a large scale analysis on extension behavioral-based fingerprinting. This framework exercises extensions and generates signatures in an automated way, in order to detect extensions' presence based on their DOM modifications, the messages they exchange and the resources they fetch. However, both of these detection systems take a snapshot of the honeypage's DOM some seconds after the page has loaded and compare it to the original version of the DOM, neglecting the aspect of time and thus missing extensions that perform ephemeral modifications. In a recent work, Solomos et al. [44] investigated how user interactions affect the fingerprintability of extensions. Specifically, they designed a system that incorporates user interactions, and detected a large number of extensions that are triggered only through user interactions. This line of work revealed a new dimension of extension fingerprinting, and their system could be used in conjunction with Chronos for detecting extensions.

With respect to extension fingerprinting countermeasures, Sjösten et al. [41] explored how extensions can reveal their presence when they inject a WAR in the page and the browser implements UUID randomization (i.e., Firefox). In this case, the injected resource's URL contains the extension's UUID in its path, and since UUIDs are unique (due to the randomization) the detection of just a single extension allows the page to uniquely track the user. As a mitigation, Sjösten et al. proposed `Latex Gloves`, a whitelist-based mechanism that determines which pages are allowed to interact with an extension's WARs and which extensions can interact with each particular website. Starov et al. [45] explored whether the artifacts that extensions leave in the page, which make them fingerprintable, are necessary for the extensions' operation. Similarly to Karami et al.'s [22], this work considers extensions that can be detected based on the messages they exchange. They found 3,320 extensions that perform modifications unnecessary for their operation (2,189 of those extensions inject unnecessary elements into the DOM and 1,526 set unnecessary attributes). However, since this work utilizes `XHound` for their exploration, which is a snapshot-based system and thus blind to ephemeral modifications, it can only observe modifications that remain on the page, and misses any unnecessary temporal artifacts that are inserted to the page and then removed. Recently Laperdrix et al. [28] proposed a technique that fingerprints extensions based on the cascading style sheets (CSS) that certain extensions inject in the page. Their approach relies on including specific elements in the page that extensions expect (i.e., are triggered by) and observe changes to their style properties when particular extensions are installed. We consider this technique orthogonal to DOM-based extension fingerprinting, as we do not expect extensions to perform temporal stylistic changes, nor have we observed any such a behavior during our experiments.

7 CONCLUSION

With browser fingerprinting continuing to proliferate across the web, extension fingerprinting presents a unique threat to users due to the two-pronged privacy loss that it incurs. More critically, as DOM-based extension fingerprinting techniques enter the global stage, accurate assessments and explorations of this fingerprinting vector are crucial for drawing more attention from the research community and incentivizing browser vendors to adopt custom-tailored countermeasures. To that end, in this paper we uncovered a previously-overlooked yet prevalent behavior in the extension ecosystem, wherein extensions perform a series of *ephemeral* modifications. With prior work overlooking the importance of an extension's *life cycle*, these short-lived changes are essentially untraceable using state-of-the-art approaches. As a result, prior studies do not capture the full scale of the threat posed by behavior-based extension fingerprinting. We presented an extensive experimental evaluation of our prototype system Chronos that highlights the importance of employing a *continuous* fingerprinting strategy as we are able to uniquely fingerprint 4,546 *additional* extensions, while also demonstrating how our fine-grained approach is highly accurate in realistic deployment scenarios where multiple extensions are installed and modify the page. We hope that our work will be a catalyst for additional privacy protections being deployed by browsers.

Acknowledgements: We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the National Science Foundation under grants CNS-1934597, CNS-2211574, CNS-2143363, CNS-2211575, CNS-2126654, CNS-1941617 as well as the Office of Naval Research under grant ONR 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] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. 2014. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. 674–689.

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

[3] Furkan Alaca and P. C. van Oorschot. 2016. Device Fingerprinting for Augmenting Web Authentication: Classification and Analysis of Methods *(ACSAC '16)*. 289–301.

[4] Ben Smith. 2019. Google Blog - Update on Project Strobe: New policies for Chrome and Drive. (2019). https://blog.google/technology/safety-security/update-project-strobe-new-policies-chrome-and-drive/.

[5] Brave. 2021. Brave Fingerprinting Protections. (2021). https://github.com/brave/brave-browser/wiki/Fingerprinting-Protections.

[6] Yinzhi Cao, Song Li, and Erik Wijmans. 2017. (Cross-)Browser Fingerprinting via OS and Hardware Level Features. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*.

[7] Chrome. 2022. ChromeDriver - WebDriver for Chrome. (2022). https://chromedriver.chromium.org/downloads.

[8] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. 2018. The Web's Sixth Sense: A Study of Scripts Accessing Smartphone Sensors. In *Proceedings of ACM CCS, October 2018*.

[9] Docker. 2022. Accelerate how you build, share, and run modern applications. (2022). https://www.docker.com/.

[10] MDN Web Docs. 2021. MutationObserver. https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver. (2021).

[11] Antonin Durey, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. 2021. FP-Redemption: Studying Browser Fingerprinting Adoption for the Sake of Web Security. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.

[12] Peter Eckersley. 2010. How Unique is Your Web Browser?. In *Proceedings of the 10th International Conference on Privacy Enhancing Technologies (PETS'10)*.

[13] Emre Erkoca. 2020. MutationObserver and Event Usage. (2020). https://dev.to/emreerkoca/mutationobserver-and-event-usage-35k6.

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

[15] Amin FaizKhademi, Mohammad Zulkernine, and Komminist Weldemariam. 2015. FPGuard: Detection and Prevention of Browser Fingerprinting. In *29th IFIP Annual Conference on Data and Applications Security and Privacy (DBSEC) (Data and Applications Security and Privacy XXIX)*, Vol. LNCS-9149. 293–308.

[16] firefox 2022. Firefox's protection against fingerprinting. (2022). https://support.mozilla.org/en-US/kb/firefox-protection-against-fingerprinting.

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

[18] Google 2022. Chrome Developers: The Privacy Sandbox. (2022). https://developer.chrome.com/docs/privacy-sandbox/.

[19] Gabor Gyorgy Gulyas, Doliere Francis Somé, Nataliia Bielova, and Claude Castelluccia. 2018. 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, 14–27.

[20] Karl Hughes. 2021. Bot Detection: Identifying Bot Traffic with Open-source Browser Fingerprinting Techniques. (2021). https://fingerprintjs.com/blog/bot-detection/.

[21] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. 2021. Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1143–1161.

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

[23] Soroush Karami, Faezeh Kalantari, Mehrnoosh Zaeifi, Xavier J Maso, Erik Trickel, Panagiotis Ilia, Yan Shoshitaishvili, Adam Doupé, and Jason Polakis. 2022. Unleash the Simulacrum: Shifting Browser Realities for Robust Extension-Fingerprinting Prevention. In *31th {USENIX} Security Symposium ({USENIX} Security 22)*.

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

[25] Pierre Laperdrix, Gildas Avoine, Benoit Baudry, and Nick Nikiforakis. 2019. Morellian Analysis for Browsers: Making Web Authentication Stronger with Canvas Fingerprinting. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 43–66.

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

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

[28] Pierre Laperdrix, Oleksii Starov, Quan Chen, Alexandros Kapravelos, and Nick Nikiforakis. 2021. Fingerprinting in Style: Detecting Browser Extensions via Injected Style Sheets. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.

[29] Xu Lin, Panagiotis Ilia, Saumya Solanki, and Jason Polakis. 2022. Phish in Sheep's Clothing: Exploring the Authentication Pitfalls of Browser Fingerprinting. In *31st USENIX Security Symposium (USENIX Security 22)*. 1651–1668.

[30] Keaton Mowery and Hovav Shacham. 2012. Pixel Perfect: Fingerprinting Canvas in HTML5. In *Proceedings of W2SP 2012*.

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

[32] Neeraj Singh. 2010. How jQuery selects elements using Sizzle. (2010). https://www.bigbinary.com/blog/how-jquery-selects-elements-using-sizzle.

[33] NPM JS. 2021. Pako in JS. (2021). https://www.npmjs.com/package/pako.

[34] NPM JS. 2021. Sizzle. A pure-JavaScript CSS selector engine designed to be easily dropped in to a host library. (2021). https://www.npmjs.com/package/sizzle.

[35] Pablo Picazo-Sanchez, Juan Tapiador, and Gerardo Schneider. 2020. After you, please: browser extensions order attacks and countermeasures. *International Journal of Information Security* 19, 6 (2020), 623–638.

[36] Corey Prophitt. 2017. Nefarious LinkedIn. https://github.com/dandrews/nefarious-linkedin. (2017).

[37] Valentino Rizzo, Stefano Traverso, and Marco Mellia. 2021. Unveiling Web Fingerprinting in the Wild Via Code Mining and Machine Learning. *Proceedings on Privacy Enhancing Technologies* 2021, 1 (2021), 43–63.

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

[39] Selenium. 2022. Selenium is a suite of tools for automating web browsers. (2022). https://www.selenium.dev/.

[40] Sergey Mostsevenko. 2021. How ad blockers can be used for browser fingerprinting. (2021). https://fingerprintjs.com/blog/ad-blocker-fingerprinting/.

[41] Alexander Sjösten, Steven Van Acker, Pablo Picazo-Sanchez, and Andrei Sabelfeld. 2019. LATEX GLOVES: Protecting Browser Extensions from Probing and Revelation Attacks. In *26th Annual Network and Distributed System Security Symposium*. The Internet Society.

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

[43] Alexander Sjösten, Daniel Hedin, and Andrei Sabelfeld. 2021. EssentialFP: Exposing the Essence of Browser Fingerprinting. In *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*. 32–48.

[44] Konstantinos Solomos, Panagiotis Ilia, Soroush Karami, Nick Nikiforakis, and Jason Polakis. 2022. The Dangers of Human Touch: Fingerprinting Browser Extensions through User Actions. In *31th {USENIX} Security Symposium ({USENIX} Security 22)*.

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

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

[47] Jorma Tarhio and Esko Ukkonen. 1993. Approximate boyer–moore string matching. *SIAM J. Comput.* 22, 2 (1993), 243–260.

[48] Erik Trickel, Oleksii Starov, Alexandros Kapravelos, Nick Nikiforakis, and Adam Doupé. 2019. Everyone is Different: Client-side Diversification for Defending Against Extension Fingerprinting. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1679–1696. https://www.usenix.org/conference/usenixsecurity19/presentation/trickel

[49] Tom Van Goethem and Wouter Joosen. 2017. 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)*.

[50] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. 2018. Fp-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies. In *27th USENIX Security Symposium (USENIX Security 18)*. 135–150.

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

[52] Antoine Vastel, Walter Rudametkin, Romain Rouvoy, and Xavier Blanc. 2020. FP-Crawlers: Studying the Resilience of Browser Fingerprinting to Block Crawlers. In *MADWeb'20 - NDSS Workshop on Measurements, Attacks, and Defenses for the Web*.

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