

Chaos: How Websites May Disrupt Browser Extensions?

Shubham Agarwal

Max Planck Institute for Security and Privacy
shubham.agarwal@mpi-sp.org

Ben Stock

CISPA Helmholtz Center for Information Security
stock@cispa.de

Abstract—Browser extensions are third-party client side add-ons that have evolved from being an optional means to enhance the Web browsing experience into an everyday essential for users worldwide. Unfortunately, the privileged nature and the vast array of essential features these extensions offer may also pose a *conflict of interest* for the websites on which they operate. Thus, adversaries may attempt to block the conflicting functionalities of these extensions, influence their runtime behavior, or cause security and privacy-related consequences.

In this study, we investigate the runtime behavior of browser extensions and measure their susceptibility to being influenced by malicious websites, resulting in altered code execution and effectively turning off the intended functionality central to the extension. Specifically, we discover two distinct classes of attacks that affect different execution contexts of browser extensions – *Same* and *Isolated Namespace Attacks* – and build an automated pipeline to measure their influence. We report 2,410 distinct extensions, with over 76M+ installs, to be consistently affected by our tests. 677 extensions were affected through the *Same Namespace Attacks*. More concerningly, we observed vulnerable cross-context data flows in 1,752 extensions through the *Isolated Namespace Attacks*, ascertaining that context-isolated components are also susceptible to Web attackers’ influence. Ultimately, we discuss several concerning code patterns and potential measures to mitigate these behaviors.

1. Introduction

Browser extensions are popular third-party client-side add-ons that provide auxiliary non-native browser features for their users on the Web. Today, browser extensions are not just touted as an optional component for Web users. Instead, they have evolved into a crucial part of the Web browsing experience, enabling a myriad of everyday functionalities, such as password management, ad-blocking, and shopping assistance, to name a few. While they offer a plethora of *fancy* functionalities, they are also extremely popular among Web users, with some extensions even accumulating over million+ install counts [1, 2].

From a security and privacy standpoint, some browser extensions may be more important than others, or have greater significance while browsing on the Web. For instance, the [Avast Online Security & Privacy](#) and [BitDefender Web Sage](#) extensions must ensure that they are always

functional and perform the configured set of operations (e.g., suspicious page detection and alerting), expected by their users for safe browsing on the Web. Otherwise, the very point of installing them would be pointless. However, to offer the functionalities they promise to their users, they often perform privileged operations, such as interactions with the Web page JavaScript to handle consent banners or script-injection to scan the served content, respectively. Unfortunately, these privileged and potentially conflicting operations on specific Web applications also make them a particularly interesting target, as they could be exploited by extension-defying websites on the client side. That is, while these extensions operate on both specific and generic sets of websites, they may conflict with those websites’ interests. Hence, websites may attempt to block these conflicting operations. Ad-blocking add-ons are a prominent example of cases in which websites have strong incentives to deny these extensions’ operations at runtime.

With the unprecedented rise in popularity of browser extensions in recent years, both anecdotal and empirical evidence by researchers and practitioners strongly suggest that malicious web applications also find them enticing. They may exploit any *direct data-flow* vulnerabilities within browser extensions, such as through `postMessages`, to carry out security-critical actions on the client side (e.g., XSS, cookie theft, and session fixation) [3, 4, 5, 6]. Orthogonally, a long tail of research also reflects upon the potential privacy risks associated with these extensions and their users (e.g., sensitive data exfiltration, extension fingerprinting, and over-privileged permissions) [7, 8, 9, 10, 11, 12, 13]. While these studies highlight apparent confidentiality and integrity issues with browser extensions, they do not consider another basic yet critical pillar of the CIA triad of application security – *availability* [14]. That is, a nefarious website may influence the runtime execution environment of browser extensions to disrupt any critical functionality, central to users’ requirements. In some instances, this *disruption-of-service* may even potentially incur security and privacy-related consequences for users.

In this study, we investigate the runtime behavior of browser extensions and their susceptibility to being influenced by the websites on which they operate, eventually leading to their reduced or disrupted execution at runtime. We analyze the execution of browser extensions based on two possible classes of attack vectors – *Same Namespace*

Attacks that affects the execution of any extension-injected script into the JavaScript context of the Web page, and the more powerful, *Isolated Namespace Attacks*, where an adversary influences the behavior of a more privileged and *context-isolated* component of browser extensions (e.g., content scripts). To this end, we built an automated pipeline, *Chaos*, and identified over 2,410 Chrome extensions as susceptible to the above classes of attacks. We extend our analysis to identify and report cases which are already influenceable through existing data flows between clobbered payloads and potentially influential or even sensitive sink APIs (e.g., `innerHTML`, `script.src`, and `setTimeout`).

To summarize, our main contributions are as follows:

- We show that malicious Web applications may compromise the runtime integrity of browser extensions by influencing the JavaScript execution environment, thereby altering their expected behavior.
- We perform a large-scale analysis of Chrome browser extensions through our automated pipeline, to investigate the impact of two separate classes of attacks - *Same Namespace Attacks*, and a more powerful *Isolated Namespace Attacks*, incurring potential *Disruption-of-Service* among extensions at runtime.
- We show that the isolated client-facing component of extensions, i.e., content scripts, are susceptible to traditional Web vulnerabilities - *DOM clobbering*, and we identified 1,437 extensions that may be indirectly influenced by our adversary.
- We discuss several code patterns and cases identified through our extended hybrid taint analysis and suggest recommendations for developers to thwart these issues.

We open-source our end-to-end analysis pipeline to help extension developers test their extensions during development and before publishing them on the store.

2. Technical Background

We now discuss the browser extension architecture along with its components and the corresponding execution contexts they operate on. We also detail on the attack vectors extensions are susceptible to, relevant to our study.

2.1. Browser Extensions

Browser extensions typically comprise three distinct components: a *service worker/background script*, the *content scripts*, and a *manifest.json* file [15]. The *service worker* [16] of an extension is a process-isolated privileged execution context that has access to browser APIs, but cannot directly interact with any Web page that the user visits. The *content scripts* [17], on the contrary, are less-privileged JavaScript components and can directly interact with the visited Web page. However, they can only access a small set of browser APIs in addition to the JavaScript APIs. In addition to these two script components, extensions may include *web_accessible_resources* [18] and leverage dedicated APIs to inject additional JavaScript at runtime (e.g., through the `scripting.executeScript` API [19] from the service

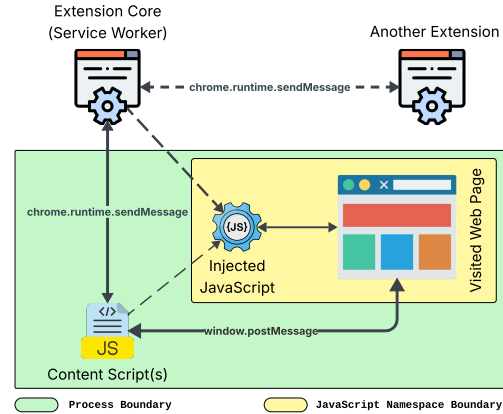


Figure 1: Different Components of Browser Extensions, their Access Boundaries, and Communication Channels.

worker, or through `document.createElement` [20] or similar APIs from content scripts). The *manifest* [21] contains the blueprint of the extension, describing the extensive set of API and host permissions, and the script that they include within different execution contexts. Optionally, browser extensions may also include HTML pop-ups [22] that act as the extension’s UI and allow users to configure their runtime behavior. Notably, each of the above JavaScript components can communicate with one another through dedicated communication APIs (e.g., `postMessage`, `chrome.runtime.sendMessage`, etc.) [23]. Figure 1 depicts the high-level architecture of a browser extensions.

2.2. JavaScript Contexts & Namespaces

Different JavaScript components of browser extensions enjoy varying privileges, owing to the different execution environments they operate on and the set of APIs to which they have access. Unlike other JavaScript components, the highly privileged service worker is both context- and process-isolated. That is, a service worker runs in its own JavaScript context and system process, and thus, is not affected by page JavaScript and the underlying process is not shared [16]. This also applies to the JavaScript associated with the UI popups of browser extensions.

The content script runs in the same browser process and context as the Web page; however, in a different JavaScript namespace, or in an *isolated* world [24]. The shared context allows content scripts to access the DOM and associated data (e.g., cookies, storage data, etc.) of the Web pages the user visits to allow page-specific operations and related features. At the same time, an isolated JavaScript namespace for content scripts and for each browser extension prevents JavaScript namespace pollution that may arise between Web page JavaScript and the extension JavaScript. As described above in Section 2.1, an extension can inject JavaScript directly into the *main* world or into the same JavaScript context where the website’s JavaScript executes. Thus, the extension-injected code can read and observe the execu-

tion traces of the Web page script. More importantly, the converse is also true, where the Web page JavaScript can observe the side effects of extension-injected code in the shared JavaScript namespace at runtime.

2.3. DOM Clobbering

While much of the client side threats and vulnerabilities associated with Web applications in the past have stemmed from script-injection attacks [25, 26], *scriptless* attacks have proven to be just as dangerous and effective. A scriptless attack is one where the attacker leverages non-script entities within the DOM, such as HTML tags themselves or their constituent elements and attributes, to ultimately execute malicious code. *Dangling markup injection*, *script-gadgets*, and *DOM clobbering* are some of the most prominent examples of scriptless or *code-reuse* attacks on the Web [27]. *DOM clobbering* [28] vulnerability surfaces when the Web page includes *named* HTML elements that conflicts with the global entities of the JavaScript it embeds, essentially interfering with the expected course of execution of the JavaScript. That is, an HTML tag with a set *id* or *name* attribute is also accessible on the global *window* or *document* object as a global variable. Hence, if the page HTML includes a named element (say, *foo*), which is then also read by the embedded JavaScript (e.g., through `window.foo`), this may lead to erroneous or unexpected execution. This vulnerability stems from the named property accesses of the W3C HTML parsing standards [29], where the named HTML properties overshadow any JavaScript variables in the global scope. Since this behavior is purely a consequence of script-less entities, any standard script-specific preventive measure (e.g., Content-Security-Policy [30] or HTML Sanitizers [31]) falls short, and unfortunately, does not prevent comprehensively against varied forms of DOM clobbering.

3. Threat Model

For our study, we define our attacker as an entity that intends to disrupt the desired functionalities and the associated execution flow of browser extensions on arbitrary websites. More specifically, we assume the attacker can control any specific website and, subsequently, *influence* the execution of browser extensions on those websites. That is, an attacker could tailor the Web page to influence the runtime actions of one or more target browser extensions.

A browser extension may directly interact with a page visited by its user through various means: by modifying the DOM (adding new or removing existing nodes), injecting stylesheets, or even, JavaScript directly into the context of the visited page, or through passively executing the JavaScript in an isolated namespace (through content scripts). For instance, let us consider a privacy-aware *ad-blocker* as a motivating example. An ad-blocker may remove DOM nodes associated with privacy-invasive third-party scripts or sensitive advertisements, while injecting new, yet allowlisted ad sources through new DOM nodes and supporting stylesheets simultaneously. Furthermore, they

```
1 // DOM Clobbering Vulnerability in Isolated Extension
   ↳ Contexts
2 const blockListsConfig =
   ↳ JSON.parse(window.blockingConfig) || {data: [],
   ↳ enabled: true};
3 if (blockListsConfig.enabled) {
4     loadConfigData(blockListsConfig.data);
5 }
6 // attacker-webpage.js -> induces runtime error
7 document.body.innerHTML += "<div
   ↳ id='blockingConfig'></div>"
```

Listing 1: An Example of Exploitable DOM Clobbering Vulnerability among Isolated Contexts of Browser Extensions

may also need to read or write the global properties of the page *window*, and thus, inject JavaScript that executes in the namespace of the Web page. In addition to these, they may also observe and manipulate the DOM throughout the page lifecycle, all through their content scripts.

An attacker website, which is not fond of ad blockers interfering with its ad-driven revenue, may be able to influence or disrupt the behavior of such an extension in various ways, as follows. The Web page JavaScript may overwrite the native definition of JavaScript APIs and other global properties accessed by a specific extension or a group of extensions, ultimately subverting their normal execution. For instance, as shown in Listing 4, the page JavaScript can influence the outcome of the execution of the `document.getElementById` API, invoked by the extension-injected code. This is possible because extension-injected code executes in the same namespace as the page JavaScript and is, therefore, susceptible to influence. Importantly, the attacker does not necessarily need to fingerprint individual browser extensions and may even readily employ a blanket attack strategy to affect a group of ad-blocking extensions with similar runtime behavior.

A more potent attacker may also penetrate through the isolation boundaries between the two JavaScript namespaces – the page JavaScript namespace, and the extension-specific execution namespace (i.e., for content scripts), without even modifying the JavaScript APIs. That is, an attacker could resort to code-reuse attacks to influence the runtime execution of the content scripts of an extension vulnerable to DOM clobbering [32, 33]. Specifically, the attacker can tailor the HTML content of the Web page to induce logical errors or alter the execution flow of the extension script, as desired. For example, Listing 1 shows that an extension JavaScript vulnerable to DOM clobbering and throws an error, as the attacker-supplied HTML pollutes the JavaScript namespace across contexts and changes the expected format of the *blockingConfig* variable. Naturally, the threat also extends to expected behavior of these scripts where the isolated content scripts read attacker-controlled data from the DOM which may lead to altered execution flow downstream. This way, an attacker could even influence the execution of the extension JavaScript across the isolation contexts originally enforced to prevent these context-related conflicts.

We believe it is feasible for the attackers to monitor and conduct an offline analysis on all the newly published

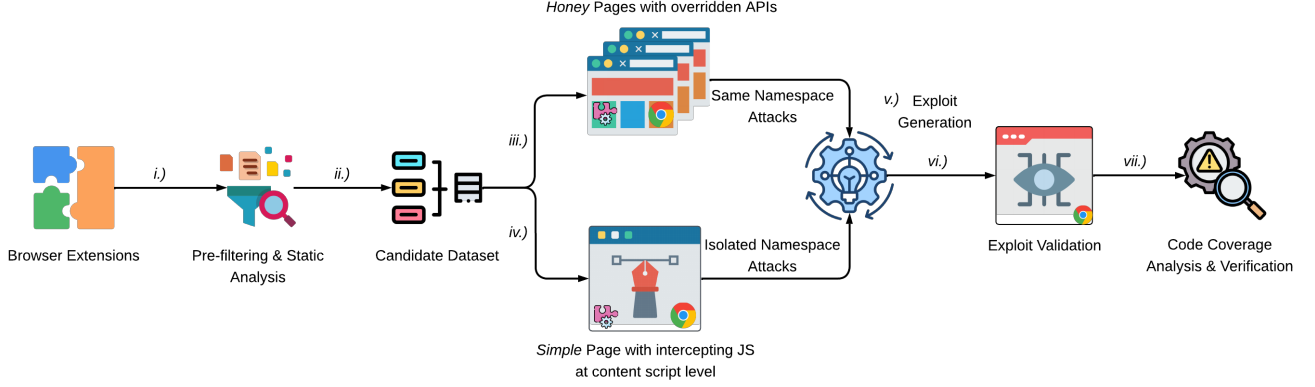


Figure 2: **Summary of our analytical pipeline:** *i)* Extract and identify extensions that contain JavaScript components running on arbitrary URLs. *ii)* Select extensions that pass the preliminary filter for further tests. *iii)* Capture execution traces and coverage data of extension-injected JavaScript. *iv)* Capture extensions’ interactions with the DOM and coverage data through content scripts. *v)* Generate HTML pages to validate the exploit for each attack. *vi)* Capture extensions’ code coverage under exploit environment. *vii)* Store and analyze the overall delta in execution and behavior.

and updated browser extensions available on the extensions stores to identify those extensions that exhibit any of the following interactions: modifying the DOM, injecting JavaScript, accessing global properties, storing data through Web Storage APIs, or having a content script vulnerable to DOM clobbering. Orthogonally, an attacker may instrument the browser [34] or forcefully execute and resolve any runtime constraints [33]. to capture more fine-grained runtime traces during offline analysis. However, they still need to instrument the JS APIs and inject clobbering HTML payloads to carry out attacks in practice. Hence, we resorted to having a singular approach both for monitoring and actual vulnerability analysis here. Importantly, we only consider those extensions for our analysis that exhibit such behavior on any arbitrary websites. Naturally, an attacker with control over a limited set of websites may also jeopardize the extensions’ runtime behavior; however, we do not model them here and only showcase a conservative lower bound of the discussed threat surface. Nevertheless, the threat model and attack vectors here are still directly transferable and applicable to attackers with limited capabilities.

4. Research Methodology

In this study, we focus on our broader research question: *How can potentially malicious Web applications disrupt the desired execution flow of browser extensions to cause functional or security-critical issues on the client side?* To that end, we first identify extensions that interact with arbitrary Web pages through its JavaScript components that users may visit. We then analyze individual extensions to identify different JavaScript components and execution contexts that may directly interact with the visited Web page, as detailed in Section 3. Based on the corresponding execution context, we mount two different *Disruption-of-Service* attacks: the *Same Namespace Attack* and the *Isolated Namespace Attack*.

We measure the impact of the two individual attacks on the overall execution of these extensions by running them against honey pages that we generate for corresponding vectors and extensions for our tests. To analyze the impact of our attacks, we first capture and assess the DOM interactions and code coverage of the extensions’ JavaScript under organic conditions before mounting the attacks as baseline. We then use this baseline to compare and contrast the extensions’ execution and interactions under attack constraints.

4.1. Chaos: Overview

To answer the above research question, we built *Chaos*, an automated end-to-end analysis tool to detect browser extensions susceptible to *Disruption-of-Service* attacks by potentially nefarious Web applications. Figure 2 illustrates our overall research methodology and the analytical steps involved. More specifically, *i.)* we first download a snapshot of all the browser extensions from the Chrome Web Store [35]. We extract individual extensions and identify those that include content scripts or web accessible resources and operate on arbitrary URLs, or have the privilege to inject JavaScript at runtime. That is, we parse the extension manifest to identify the scripts and their corresponding hosts on which they execute. We additionally check for requested API permissions to identify those extensions that inject JavaScript at runtime. *ii.)* Based on the detected JavaScript component, we categorize the extension for testing under either or both attack constraints. Web accessible resources are subject to *same namespace attacks* while content scripts are subject to *isolated namespace attacks*.

For extension-injected JavaScript, *iii.)* we subject the corresponding extension to *same namespace attack* and capture the execution traces of the extension-injected code, along with their baseline code coverage metrics. *iv.)* Similarly, we then capture the execution of content scripts

within extensions, selected for *isolated namespace attack*, along with their DOM interactions and the code coverage statistics. v.) Next, we analyze the collected DOM interaction data and execution traces to generate HTML test pages for individual attacks in an automated fashion. The generated HTML varies with individual extensions in most cases due to the varied interaction patterns they exhibit at runtime. vi.) Through these generated HTML pages, we reload the extension afresh and validate the exploitability of the browser extensions under exploit conditions that mimic the adversarial environment. vii.) Lastly, we analyze the code coverage and DOM-based mutations from the exploit environment to assess the overall impact of the two attacks.

4.2. Pre-filtering & Dataset Construction

As discussed in Section 3, we focus our attention towards browser extensions that operate on arbitrary URLs for this study. However, we stress that our threat model also extends to the remaining extensions that operate on a specific set of hosts, which we omit here. Specifically, we download all the browser extensions available on the Chrome Web Store, extract them, and parse their manifest. Through the information provided in the corresponding manifest, we identify extensions which a.) includes at least one *web accessible resources* that is JavaScript and operates on `<all_urls>`, b.) includes at least one *content scripts* that is JavaScript and operates on `<all_urls>`, or c.) includes a *service worker* (or *background scripts*) in conjunction with the *scripting* (or *tabs*) API permission and `<all_urls>` as host permissions. Here, extensions may inject JavaScript, declared statically as *web accessible resources*, into the *main* world at runtime. Whereas, the declared content scripts, by definition, always execute in an isolated environment, unless explicitly specified to `run_at` in the *main* world. Orthogonally, extensions may inject JavaScript dynamically (i.e., not declared in the manifest). They may inject JavaScript into the *main* world, similar to *web accessible resources*, or may inject a content script at runtime in the *isolated* world, through the *scripting* (or its precursor *tabs*) API. Thus, we consider all the above cases in our filter strategy. Following prior works [12, 36], we also normalize the wildcard host expressions (`*://*`, `https://*/*`, `*://*.example.org/*`, etc.) and accurately identify extensions running on arbitrary hosts.

Through the above filtering process, we construct two individual datasets of extensions. The first dataset corresponds to those extensions that include *web accessible resources* operable on arbitrary hosts. Whereas, the second dataset consists of extensions that include a content script and also run on `<all_urls>` or its semantically equivalent wildcards. Naturally, an extension may include both of these JavaScript components, and thus, we include them in both dataset separately. Since we do not rely on static API signature detection, we also consider extensions with script injection privileges *scripting* within both the dataset for the sake of completeness. This is because the *scripting* API enables

```

1 function _overwriteAPI(obj, prop, isBaseline) {
2   let _native = obj[prop];
3   function _hook() {
4     _logInvocation(obj, prop, arguments);
5     if (isBaseline) return _native.apply(this,
    ↪ arguments);
6     else throw new Error('Mallory');
7   }
8   obj[prop] = _hook;
9 }
10
11 function _overwriteProp(obj, prop, isBaseline) {
12   let _getter =
    ↪ obj["__lookupGetter__"](prop).bind(obj);
13   let _setter =
    ↪ obj["__lookupSetter__"](prop).bind(obj);
14
15   function _modGetter() {
16     _logInvocation(obj, prop, arguments);
17     if (isBaseline) return _getter(...arguments);
18     else throw new Error('Mallory');
19   }
20   function _modSetter() {
21     _logInvocation(obj, prop, arguments);
22     if (isBaseline) return _setter(...arguments);
23     else throw new Error('Mallory');
24   }
25   Object.defineProperty(obj, prop, {
26     get: _modGetter, set: _modSetter,
27   });
28 }

```

Listing 2: Overwriting Global JavaScript APIs & Object Properties to Capture their Invocation at Runtime.

JavaScript injection in both *main* and *isolated* JavaScript namespace, configured through the *world* property [37].

In this step, we also extract additional information relevant from extension code to complement runtime traces and prepare attack templates accordingly. Concretely, we extract all the variables registers by the WARs on the *window* and *document*. Similarly, we also extract keys and properties accessed on prototypes of popular JavaScript objects and DOM queries. We utilize these data to supplement the honey page construction step for corresponding attacks. Next, we conduct tests for the extension datasets, corresponding to *same* and *isolated* namespace attacks, respectively, as per our threat model. Since the underlying methodology and attack vectors differ between the two classes of attacks, we now discuss them individually in the following.

4.3. Same Namespace Attacks

Browser extensions often inject JavaScript directly into the namespace of the visited Web page to read or overwrite global variables, or execute code as first-party entities. They may achieve this either through injecting the JavaScript from their content scripts (e.g., by using a combination of `document.createElement` and `document.appendChild` APIs [20, 38]) or indirectly from their service worker by leveraging the *scripting* APIs [19]). The extension-injected script that executes in the same context and JavaScript namespace of the Web application is prone to being directly influenced by the Web page JavaScript through various means. Thus, we broadly term this as *Same Namespace Attacks* in our study.

Prior work in this direction by Agarwal et al. [36] focused on the side effects of these observable execution traces of browser extensions, showed that this could be unique in nature lead to fingerprinting. However, we argue that a malicious Web application could very well advance beyond merely fingerprinting the presence of an extension, to disrupt its client side functionality through manipulating the shared execution environment, or even cause potential security and privacy implications. Furthermore, in this case, our attacker may not care about runtime behavior unique to individual extensions. For instance, multiple ad blockers may share a similar runtime execution pattern which an adversary may utilize to disable them all at once without the need to fingerprint them apriori. Thus, we investigate the means through which Web adversaries may disrupt extensions and their functionalities at runtime. The first dataset of extensions that may statically or dynamically inject JavaScript in the *main* world forms the candidate dataset here.

4.3.1. Data Collection. Since our adversary is sufficiently able to listen to and observe the execution traces of extension-injected JavaScript, we mimic their behavior and build *honey* pages to capture them at runtime. Thus, we employ the methodology adopted by Agarwal et al. [36] and Lekies et al. [25] to collect the execution traces of extension JavaScript in the shared namespace. That is, we overwrite the native definitions of the global JavaScript APIs and properties that are shared between the Web page and the injected code to capture the corresponding APIs it invokes and properties it accesses. As shown in Listing 2, the `_overwriteAPI` method overwrites the native definition of a given API (line 8) by assigning it to a custom function we define (line 3) to capture their invocations, while also preserving their intended behavior to avoid any side effects while capture their natural course of execution (line 5). Similarly, we also overwrite the object properties (line 11), through redefining the corresponding *getters* and *setters* of the object properties. This way, an adversary can observe and identify the APIs used by the injected code at runtime, and primarily serves as a potential target for disruptions.

In addition to observing global API invocations and object property accesses, we also capture other runtime behavior that may be abused to influence the extensions’ runtime execution. We collect all the variables set by extensions as well as the data stored by them on the client side at runtime. The data collected here complements to the variable data that we extracted during pre-filtering stage. We reiterate that while prior work [36] leveraged the uniqueness of these data points to measure their fingerprinting characteristics, our goal here is orthogonal. In addition to these, we also monitor registration of custom properties on the prototype of JavaScript objects which is then inherited by each of the corresponding instances [39, 40], such as `Function.prototype`, as shown in Listing 3 (line 8).

While we monitor the execution traces of the extension-injected code, we also explicitly capture two additional data – code coverage metrics of the extension-injected code, and any potential DOM-related interactions that these scripts

may have had during our tests. These two runtime metrics serve as the baseline indicators of an individual extensions’ runtime behavior in its natural environment, and thus, are crucial to our data collection strategy. The `Profiler` API, exposed through the Chrome Devtools Protocol [41], allows capturing precise and detailed block-level code coverage for individual scripts. We use this API with our *Playwright*-controlled browser instances [42] to extract code coverage data for individual extensions. Further, we utilize the `MutationObserver` API [43] that logs all extension-induced DOM-related modifications at runtime.

4.3.2. Attack Validation. Once we collect the ground truth data points (e.g., execution traces, global variables, DOM mutations, etc.) for each candidate extension, as described above, we analyze them to generate corresponding test pages and confirm the exploit. Here, we consider our exploit to be successful when the adversary is able to significantly influence the target extension, resulting in a substantial difference in the execution of extension code. We relaunch a fresh browser instance with the browser extensions, albeit now under attack conditions. That is, we modify the *honey* page to disrupt the extensions’ runtime execution. Hence, our attack strategy is straightforward. For each extension, we overwrite the corresponding API that now throws an error and instantly halts the execution of the extension-injected code (as in line 6 of Listing 2). Similarly, we overwrite all object properties, known custom properties of different JavaScript object prototypes, as well as pre-define all the global variables we collected through the `Object.defineProperty` API [44], now with `enumerable` and `configurable` properties set to *false*, as shown in Listing 3 (line 9–17). This way, when the injected code attempts to register or access a global variable or custom property in the same namespace, here pre-defined by us, the registration fails silently while its further usage may incur errors. Lastly, we also clear the client-side data stored by extensions. As before, we collect the code coverage metrics and any DOM-related mutations through extensions to assess the overall impact.

4.4. Isolated Namespace Attacks

Isolated script components of browser extensions, content or background scripts, can interact with the DOM of the visited Web pages through direct or indirect channels. However, the JavaScript within executes in an isolated JavaScript namespace. That is, the Web page JavaScript cannot directly observe the execution traces from the content script of extensions, since they have an isolated JavaScript namespace. For instance, a top-level variable `foo` (or `window.foo`) declared by both the Web page JavaScript and the extension script here refers to two different variable instances, as the underlying `window` object is no longer shared. Naturally, the two namespaces also have separate copies of JavaScript APIs, and overwriting the definition in one namespace does not affect the other. However, the same does not apply to the property accesses which conflicts with *named* HTML elements. More specifically, any HTML element in the Web

```

1  const KEYS = ['config', 'settings', ...];
2  const PROTOTYPES = [
3    [Object.prototype], ...];
4  for (const [proto] of PROTOTYPES) {
5    for (const key of KEYS) {
6      if (!_hasOwnProperty.call(proto, key)) continue;
7      __logCustomProp(proto, key);
8      Object.defineProperty(Object, proto, key, {
9        get() { throw new Error('Mallory!'); },
10       set() { throw new Error('Mallory!'); },
11       configurable: false, enumerable: true,
12     });
13   }
14 }

```

Listing 3: Monitoring & Poisoning Known Keys of Different *Object Prototypes* to Influence their Accesses at Runtime.

page, which includes the *id* or *name* property, is globally accessible on the *window* object, as per the HTML specifications [29]. This is also true for the namespace-isolated content scripts. That is, these named HTML entities may lead to a DOM clobbering vulnerability in content scripts by registering these named properties as global variables, thereby polluting the isolated JavaScript namespace.

Prior studies related to DOM clobbering [32, 33] investigated its impact on traditional Web applications. In this study, we instead show that DOM Clobbering also affects browser extensions, and even across isolated JavaScript boundaries, due to the underlying HTML parsing specifications. Thus, a malicious Web application with prior knowledge of a target extension may include additional HTML entities to meddle with the execution of the target content scripts. In fact, extension scripts that are bundled or include third-party libraries often register variables in the global scope and are thus susceptible to DOM clobbering, as shown by Liu et al. [33]. Hence, during the static pre-filtering stage, we analyze individual content scripts to identify code bundlers or DOM interactions that we then complement with runtime traces to generate custom HTML payloads for runtime tests. Orthogonally, extension content scripts may also access data from DOM nodes which may then dictate further functionalities and code execution downstream. While this is expected behavior among extensions, unfortunately, our adversary can still influence the script execution by injecting clobbering payloads. This is analogous to the insecure data flow from script- and DOM-gadget sources [45, 46, 47] into security-critical sinks on the Web.

4.4.1. Extension Instrumentation & Data Collections. To mount isolated namespace attack against extensions, an attacker needs to know in advance about the execution pattern of content scripts, their interactions with the DOM and the client side storage APIs. Thus, we first capture execution patterns for all our candidate extensions. However, this is relatively non-trivial as compared to monitoring execution traces in the shared JavaScript namespace. Thus, in the pre-attack phase, we instrument extensions to enable runtime monitoring. Kapravelos et al. [48] and later Starov, Oleksii and Nikiforakis, Nick [9] instrumented the content script-facing JavaScript APIs to monitor their DOM-interactions

in depth. That is, they instrumented the DOM-facing APIs (e.g., `getElementById`, `getElementsByTagName`, `querySelector`, etc.) which allowed them to monitor the accessed elements as well as generate *on the fly* and embed them within the DOM if they did not exist at runtime. (Unfortunately, the scripts and the codebase associated with these studies are not publicly available). Inspired from these methods, we resorted to implementing these features along with recently introduced APIs that we include for monitor.

Concretely, we first identify all the DOM APIs that allow access to DOM elements based on their characteristics (as listed in Table 4). We then instrument these APIs to record their invocations through extension (content) scripts. Listing 5 provides an overview of our instrumentation strategy that we now discuss in detail. In case the element(s) does not exist, we generate the element with supplied arguments and inject them as a child node precisely into the parent node where the API was invoked by the extension code (line 2–5) (falling back to `document.body` when invoked on the top-level document node). Through this, we also handle any subsequent queries to the same element to avoid duplicate injections. Further, many APIs we instrument returns *HTMLCollection* [49], instead of a singular element (e.g., `getElementByClassName`, `querySelector`, etc.). The extension code may access the a target element through known indices or through additional runtime checks. For instance, a content script may look for a certain named `iframe`, and invokes `document.getElementsByTagName("iframe")` - `[3].name`. We model these cases by wrapping the accessed object - both existing and newly generated ones, into a *proxy* object (line 8), through utilizing the *Proxy* JavaScript API [50]. The now *proxi-fied* object enables capturing any fine-grained operations on the accessed object, including index-based accesses, attribute reads and writes, etc. Importantly, the DOM clobbering vulnerability necessitates that the conflicting variable was read by the vulnerable component before being explicitly set. Hence, we also capture the order of variable reads and writes by content scripts on the global scope that we later use to build clobbering exploits.

We implement the above monitoring and capturing logic in a separate JavaScript file that our candidate extension loads and executes before its own content scripts at runtime. To achieve this, we rewrite the *manifest* of each candidate extensions, such that our monitoring content script is the first entry and the first script to execute before other listed content scripts. Notably, we empirically tested and confirmed that content scripts execute in the order of their definition in the manifest.

4.4.2. Attack Validation. Through our extension instrumentation strategy above, we mimic our attacker who is capable of performing offline analysis of browser extensions and collecting DOM-related interactions through content scripts. In the following step, we now build exploit templates for individual extensions based on the advanced runtime monitoring of DOM-facing JavaScript APIs combined with complementary information obtained from static analysis.

TABLE 1: Summary of the Extensions Dataset for this study.

Total downloadable extensions (as on 14.04.2026)	265,271
Candidate extensions for:	
↪ SNA test candidates	30,385
↪ INA tests candidates	54,058
↪ Overlapping Extensions b/w SNA & INA	30,014

That is, we utilize the DOM queries that we collected (i.e., elements, tags, attributes, etc.) to build corresponding HTML pages, that when queried, will result in disruption of content script logic and extension execution at runtime. For instance, we craft the HTML page with a dummy element with a dummy `div` element with `id` set to `foo`, if we observed a corresponding API invocation (e.g., `document.getElementById('foo')`) or a variable access (i.e., `document.foo`) for the corresponding extension. Essentially, we reuse the HTML structure from the data collection stage, where our monitoring script already created all queried elements that the extensions query at runtime. However, we now modify the associated values (e.g., `innerText`, `outerText`, etc.) for the queried nodes accessed by extensions through content scripts. While we do not extensively capture the expected data type of the value, we test different values based on the accesses we observed through our node proxies in the data collection strategy. For example, if we observe `node.src`, we expect the datatype to be string, and thus, supply corresponding values. We set the value to `undefined` as a fallback when we cannot infer the expected datatype.

Through the above approach, our attacker aims to disrupt or halt the organic execution of content scripts. We reload individual browser extensions and run them on corresponding custom HTML pages. We observe the runtime execution of these extensions by capturing their code coverage and DOM interactions. As with *same namespace attacks*, we collect the code coverage and DOM interactions at this stage as well for further comparative analysis.

4.5. Impact Analysis

Our attackers’ goal is to meddle with the runtime execution of extensions and disrupt their functionality. Thus, we consider two crucial signals that serve as indicators to assess *direct* impact of our two attack classes on these extensions – *fraction of code executed* and *DOM-based interactions*.

Code coverage analysis enables us to compare and contrast the fine-grained execution of individual script files in different execution context, as well as injected JavaScript, at the block level. We gather these metrics through the `Profiler.startPreciseCoverage` API [41], accessible through the Chrome Devtools Protocol. More precisely, the `CoverageRange` object, returned by the API, reports the *start* and *end offsets* along with the count of JavaScript instruction that executed within each block. We use this block-level execution analytics to determine the differences under baseline and attacker constraints. Further, we also capture any transient or persistent DOM-based mutations caused by extension code. This allows us to assess the

attacker’s ability and impact at the page or UI level. The attacker, here, may monitor and adapt to the DOM structure the extension expects to influence their runtime behavior, for instance, by denying them node injections or removal. Importantly, we only consider coverage delta beyond a minimum threshold of *1KB*, ruling out any non-deterministic jitters, and report strict lower-bounds.

5. Evaluation & Results

Based on our attacker model and research methodology, we built an automated analysis pipeline, *Chaos*, and analyzed live Chrome extensions to identify already vulnerable extensions in the wild. We downloaded all publicly listed Chrome browser extensions [35] as of April 14th, 2026, and identified those extensions that operated on arbitrary URLs and were included in relevant scripts and/or corresponding permissions, as detailed in Section 4.2. Table 1 summarizes the extension dataset we used for our tests. Consistent with prior research, we repurposed the HTML honey pages used in prior studies [12, 13]. Notably, non-deterministic execution pattern among browser extensions have been reported in the past [51], due to extensions’ reliance on page events and the race conditions that affect measurement tools. To negate these effects, we ran our tests *three* times on individual extensions to identify any inconsistencies during data collection (e.g., race conditions, random code injection or execution patterns, etc.) and only considered extensions which showed *consistent* and *unidirectional* behavior across all *three* tests. Nevertheless, we emphasize since the attacker is not bound by the limitations and indeterminism caused by our measurement tools, they can deterministically carry out the attack through merely controlling the page that users visit. Overall, we report 2,410 extensions to be consistently *disrupted* by either of the two classes of attacks we discovered. We now discuss our evaluation and the impact of each attacks in the following.

5.1. Same Namespace Attacks (SNA)

We tested 30,014 extensions for same namespace attacks that included code-level signatures or permissions to inject and execute JavaScript into the context of the visited page. We first monitored these extensions and calculated their execution traces through three distinct baseline visits. Next, we then observed their execution traces and runtime behavior under attack constraints, i.e., on a specially crafted honey page designed to influence their execution. We summarize our findings observed for SNA in Table 2. 677 extensions showed that our attacker had a consistent influence across all three visits. However, it is worth noting that almost 4.5x more extensions showed altered execution in at least one visit, and similarly, 3x in 2/3 visits. Upon investigating this further, we found this behavior to be an inherent limitation of the tools available for extension-related data collection and measurements. That is, we observed race conditions among the execution threads of the extension-injected script, our own API-overwriting script, and the CDP APIs used

TABLE 2: Impact of *Same Namespace Attacks* on Extensions: 677 extensions were *consistently* impacted across multiple iterations of our tests, 1,864 extensions showed disruptions in 2/3 visits, and 2,728 in at least 1 visit.

Attack	Consistent (all 3 visits)	Majority ($\geq 2/3$)	At least 1 visit
API Overwrites	406	584	1,256
Variable Pollution	151	350	1,059
Event Poisoning	160	305	1,021
State Cleanup †	62	250	979
Prototype Poisoning †	81	254	1,031
Total Distinct Extensions	677	1,864	2,728

to capture code coverage information for individual scripts. This, in fact, also directly translates to the randomness among individual vectors that we monitored. That is, our attacker model assumes that the attacker JavaScript and code coverage API must run before an extension injects its own JavaScript. Unfortunately, we observed this to be non-deterministic in our automated pipeline across visits for the same set of extensions, directly affecting vectors that relied on execution order (e.g., variable assignment before registration by extension code, client-side state cleanup, etc.). Importantly, although our automated pipeline may fail to capture these behaviors across visits consistently, our adversary does not impose these requirements and thus is not restricted by them. Additionally, we observed non-deterministic, random execution traces induced by extensions on each page load, which we ruled out by explicitly relying on consistent signals. Further, an honest-but-curious extension may modify its runtime behavior due to the tampered JavaScript execution environment. However, this is common for both baseline as well as attack test pages that we use in our pipeline, thus, ruling out any intentional runtime deviations by extensions across baseline and attack conditions. Hence, we report a strict lower bound of all the extensions affected through SNA.

Table 3 provides an overview of the impact of the altered shared execution environment on extension-injected code. We measure functional disruption among individual extensions through calculating *the fraction of code killed* for each of them as: $frac_code_kill = (\sum_s exec_s^{base} - exec_s^{atk}) / \sum_s src_s$, where $exec_s$ and src_s are V8-covered and total source bytes of script s . This way, we measure the impact on all the scripts affected by our tests at once, and normalize the execution delta across three distinct visits. Through $frac_code_kill$, we identify the *functional disruptions* due to our attack on individual extensions. While over 62.7% of extensions showed reductions of 5–20% in code execution, we also observed cases in which extensions executed only half or less of their code compared to their baseline behavior. For instance, [Petra Aptos Wallet](#) loses its entire page-facing functionality due to API overwrites, which affects its *inpage.js* script. Similarly, [Eternl](#) also injects *dom.js* into the Web page, and thus, executes 60% less code due to event poisoning, state cleanups, and prototype poisoning. Orthogonally, [Leap Wallet](#) was solely affected by

TABLE 3: Kill Severity Distribution and Functional Disruptions among Browser Extensions Consistently Affected by SNA. Note that a single extension may be affected by multiple vectors, and thus, fewer column-wise aggregates.

Attack Vector	#	>80% killed	50–80% killed	20–50% killed	5–20% killed
API Overwrites	406	4	15	161	226
Variable Pollution	151	3	10	59	79
Event Poisoning	160	1	3	17	139
State Cleanup	62	3	2	11	46
Prototype Poisoning	81	3	5	14	59
Overall (distinct)	677	12	25	216	424

variable poisoning, relying on the `window.leap`, while attacker code simply initialized it to `{}`, thus, disrupting the entire script execution. Note that we only consider extensions that were consistently affected during all three visits and within similar thresholds to eliminate any noise.

5.2. Isolated Namespace Attacks (INA)

As with SNA, we ran baseline tests for all 54K extensions subject to INA tests in the first phase to capture their runtime traces and DOM-interactions over three distinct visits to our honey pages. Recall that we instrumented the candidate extensions by including additional content script to capture variable accesses within the isolated JavaScript context as well as DOM-level interactions. This way, we observed a total of 19,552 extensions to interact with the DOM or register variables susceptible to clobbering (i.e., variable read-before-explicit-set) in the global scope. We complemented this data with the DOM interactions we statically captured through content scripts during pre-filtering stage. Overall, based on the collected traces, we selected 40,213 extensions for which we could generate clobbering payloads, used to test the extension under attack constraints (as summarized in Table 8). As summarized in Figure 3, 1,437 extensions incurred functional disruptions or exhibited potentially data flows through our attack in the isolated context. We now discuss the two attack sub-classes and their impact separately.

DOM Clobbering through global variables: We observed a total of 1,323 extensions that accessed any variable in the global scope before explicitly setting them at runtime. We reiterate that while our honey page includes varied HTML content to trigger as much extension-associated behavior as possible, we may have missed capturing many variable accesses behind unfulfilled conditional logic. Of these 1,323 extensions, we observed 389 of these consistently accessing the corresponding clobbered payload under the attack environment. Interestingly, 112 / 389 extensions not only read the clobbered data but also demonstrated its utility and the data flows that ended up in potentially sensitive sinks (e.g., `fetch`, `setAttribute`, and `href`). 230 extensions accessed the clobbered payload and read the data without any explicit signs of sanity checks before using the data, while 47 of them did check the `nodeType`

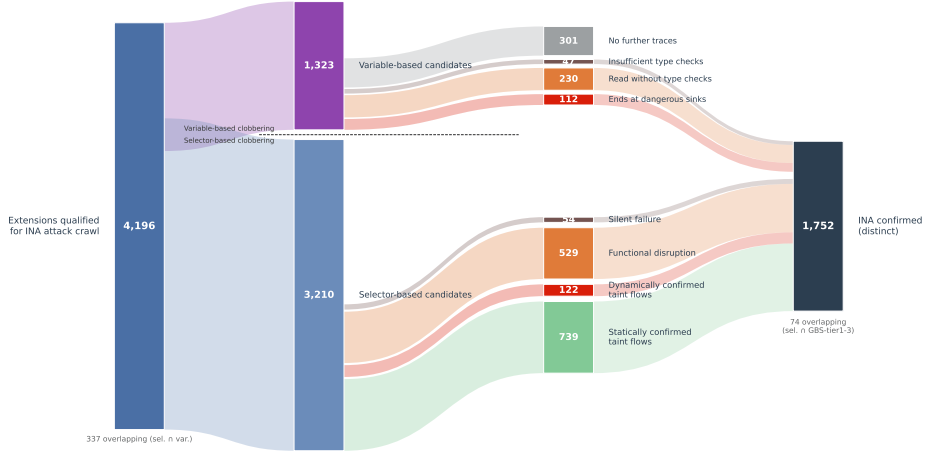


Figure 3: Breakdown of Impact on Browser Extensions Affected by INA. The isolated scripts (e.g., content scripts) were vulnerable to two different classes of data flows: the DOM clobbering vulnerability induced through *get-before-set* variables, and attacker-controlled DOM-gadgets. We omit the 301 extensions with no further traces from our final reporting.

and `nodeName`, yet showed further data flow. The remaining 301 extensions accessed the clobbered payload and read functional and structural properties, such as `length` and `forEach`. [Verified Wallet](#), for instance, showed total disruption or silent failure under clobbered conditions (`window.process` and `window.Buffer`). Similarly, [AutomatR Extension](#) showed significant disruption through clobbering `window.jQuery`, a pattern shared with numerous other extensions, albeit with varied end impact.

DOM reads through Selector APIs: Of all the 40,213 extensions that we tested under attack environment, 16,453 extensions interacted with clobbered payloads through selector-related APIs (e.g., `querySelector` and `getElementById`). We then explicitly focused on the specific set of source properties on the clobbered payload that extensions accessed (as listed in Table 7). Notably, we relied on prior empirical and anecdotal findings on DOM clobbering and their functional or security-critical repercussions to curate the subset of source properties [28, 32, 33]. Through this approach, we narrowed our investigation to a total of 3,210, recording explicit property reads on these payloads consistently across visits. Through measuring *frac_code_kill* (as for SNA), we found 529 / 3,210 of these to have experienced functional disruptions (>50%: 86, 20–50%: 142, 5–20%: 301) while 54 others even failed silently, when combined with the DOM-interaction patterns we captured. For instance, we observed silent failure for [Passei Direto](#) as relying on `textarea` with `sb_form_q` id on the DOM to perform 268 further DOM-level mutations which was all suppressed by our attack payload (detailed in Listing 6). Whereas, while our attacks only disrupted 5% of total execution for [Avast Online Security & Privacy](#), the injected payload disrupted the compliance notification component of the extension, since this component solely relied on the rendered DOM elements of the visited page –

`document.getElementsByName("aosp-compliance")` (see affected LoC in Listing 7 for more details).

A large fraction of extensions that interacted with our injected payloads during any visit and accessed properties did not exhibit consistent, substantial functional disruption. For these cases, we performed a lightweight hybrid taint analysis to identify flows that originate from the injected payload when the extension code reads associated properties. We ran dynamic tests and confirmed existing data flow to potentially sensitive sinks for 122 extensions. [Cast-Buddy](#), for example, read the `src` property of `video` element through `document.getElementsByTagName` which it then relayed to its service worker through `chrome.runtime.sendMessage` API, flagged by our dynamic taint analysis. Furthermore, our static taint analysis (detailed in Appendix A) revealed another 739 extensions with data flows from clobbered sources to sensitive sinks, missed by our dynamic monitor due to potential runtime constraints. For example, [Pop up blocker for Chrome](#), we found a data flow originating from anchor `hrefs` to service workers (in `content.js`) through the messaging API, however, only when the user clicks on the link. We detail on additional manually-identified case studies in Appendix B.

6. Discussion

We now discuss the limitations of our study methodology and suggest actionable steps and strategies for extension developers to build safer and robust extensions.

6.1. Limitations

In this study, we identify ways an attacker Web application can disrupt the runtime execution of browser extensions. However, as discussed in Section 3, we report here only a conservative lower bound on the number of browser extensions affected by our attacker, due to the inherent limitations

of our experimental design and the tools used for data collection. First, we only consider extensions that execute on arbitrary hosts. However, extensions running on a specific set of hosts are equally susceptible to the underlying threat. In fact, websites such as e-commerce platforms may even have added incentives to disarm and be better positioned to adapt to certain extensions (e.g., Honey). Secondly, we only run extensions on our own test origin, which may not be the target host for corresponding extensions, despite including host permissions for arbitrary origins. Third, in our tests, we rely on a baseline HTML (honey) page also used in prior research. However, extension logic may have different preconditions for execution that our test pages do not model, and thus may go undetected by our pipeline. Then, we rely on the *Profiler* API, exposed via the Chrome DevTools Protocol, to capture code coverage of individual JavaScript at runtime. While our attacker model also applies to Mozilla extensions, unfortunately, there is no direct alternative to the *Profiler* API in the Mozilla ecosystem. Thus, we did not analyze the Mozilla extensions in this study. Lastly, we report a conservative lower bound figure for extensions affected by *same namespace tests*. This is because we observed substantial race conditions between the CDP thread (that we rely on for code coverage metrics) and the execution of the extension-injected scripts across visits, particularly for those that operate on *document.start*. To rule out any discrepancies, we only reported extensions consistently affected across *all three* visits. However, other extensions that exhibit inconsistent influence during our tests remain vulnerable to the underlying threat.

6.2. Recommendations

Contrary to our understanding and the popular belief, the *isolated* content scripts could be influenced and susceptible to DOM clobbering vulnerabilities. To curb this issue, extension developers shall only *get* variables and properties after explicitly *setting* them or verifying the return data. Reading specific DOM content (and optionally verifying its parent) by specifying the full path of the target node through the *querySelector* API may eliminate reads from intentional out-of-order markups on the page. Furthermore, they must perform additional sanitization and checks on data read from the DOM against allowlisted values to prevent them from falling into security-critical sinks (e.g., *script.src*, *location.href*, *element.innerHTML*, etc.). We also suggest that the maintainers of JavaScript bundling libraries incorporate extension-associated APIs and appropriately scope global variables into their DOM clobbering defense suite.

On the other hand, it is challenging for developers to fully prevent *same namespace attacks* due to inherent flaws in the extension architecture, as also highlighted by prior works. However, for the same namespace, JavaScript running at *document.start*, developers may resort to alternate script-injection techniques: declare WARs as content scripts running in the *main* world in the *manifest*, wherever feasible. This forces scripts to execute before any page JavaScript, allowing to retain the native definitions of APIs.

7. Related Work

In this section, we outline prior studies on the threats and vulnerabilities related to the browser extensions. We first highlight the dangers associated with *benign-but-buggy* extensions directly related to our study, and then, also briefly shed light on the threats corresponding to malicious browser extensions that have been reported so far.

Benign-but-Buggy Extensions: Seminal works on browser extensions, Barth et al. [52] showed that extensions often have elevated API privileges, and thus argued for the *principle of least privilege* for extensions. Similarly, Carlini et al. [53] also argued for firm isolation boundaries between different extension components to avoid any privilege escalation vulnerabilities. More recently, Somé [3] and Fass et al. [4], through independent studies, showed that extensions are susceptible to attacker-controlled data flows solely through messaging channels, resulting in severe client side threats. Kim and Lee [54] reported that, despite the isolation boundaries in place, extensions are still vulnerable to universal cross-site scripting and privilege escalation, among other threats on the client side. Yu et al. [6] developed and proposed an advanced static analysis tool to detect vulnerabilities in browser extensions, aiming to mitigate these issues in the wild. Orthogonally, a long tail of research showed that extensions exhibit a multitude of observable fingerprintable characteristics at runtime [7, 8, 9, 10, 11, 12, 13, 36, 55, 56, 57, 58]. Sjösten et al. [59] and Karami et al. [60], in their respective follow-up works, also proposed strategies extension developers could use to avoid extension fingerprinting through static runtime identifiers and DOM interactions, respectively. Agarwal et al. [61] recently conducted a qualitative study with browser extension developers to understand their security and privacy mindset, and reported technical and eco-systemic obstacles behind insecure development practices.

Notably, while the above research studies provide ample evidence of *confidentiality* and *integrity* issues among browser extensions, none of these studies highlight the issues associated with their (*un-*)*availability* or *disruption*. In this study, we aim to bridge this gap and discover potential means to cause *Disruption-of-Service* among extensions.

Malicious Browser Extensions: Kapravelos et al. [48] analyzed the runtime behavior of malicious extensions and discovered a wide array of attacks, including malicious code injection and affiliate fraud. Similarly, Jagpal et al. [62] conducted a large-scale longitudinal analysis of Chrome extensions using hybrid analysis techniques and identified similar abuse vectors, in addition to extensions involved in Facebook hijacking and other social-engineering campaigns. Aggarwal et al. [63] demonstrated that extensions often acted as spies, exfiltrating potentially sensitive information. Perrotta and Hao [64] showed that extensions may also act as botnets, contacting an external command & control server to adapt to its runtime behavior. Xing et al. [65] and Thomas et al. [66] also showed that extensions inject unwanted ads at runtime. Pantelaio et al. [67] performed a longitudinal study on the update patterns among browser extensions and

discovered that browser extensions may turn malicious after receiving updates. While malicious extensions continue to exist, we instead focus on their benign counterparts that are susceptible to being influenced by malicious applications.

DOM Clobbering: Researchers and practitioners highlighted the threats associated with scriptless attacks that affects Web applications, including DOM clobbering, as early as a decade ago. Heiderich et al. [27] discussed various scriptless attacks, carried out through CSS files, SVG images, font files, and dangling HTML markups, and suggested high-level countermeasures to mitigate them. Khodayari and Pellegrino [32] conducted a comprehensive literature survey through analyzing Tranco Top 5,000 websites to identify five distinct classes of DOM clobbering issues in the wild. More recently, Liu et al. [33] built a dynamic analysis framework through utilizing Symbolic DOM to model attacker-controlled HTML markups. They discovered popular DOM-clobbering gadgets in JavaScript libraries and bundlers.

The above DOM clobbering studies solely focused on Web applications. Whereas, in this study, we uncover the presence and prevalence of DOM clobbering vulnerabilities among browser extension, showing that the underlying threat is unfortunately transferable to the extension ecosystem.

8. Conclusion

In this study, we discover two different classes of vectors among browser extensions, the *Shared & Isolated Namespace Attacks*, that a malicious Web application may exploit to influence their execution and behavior at runtime. Through our automated detection pipeline, *Chaos*, we identified 2,410 live Chrome extensions already susceptible to either of the two vectors we discovered. Our study underpins the fact that the effects of *isolated* attacks are particularly concerning, as they may lead to unexpected disruptions and potentially security-critical side effects across isolated JavaScript contexts. Unfortunately, the underlying issues stem from current architectural design and inherent limitations in modern browsers, and thus, require attention from developers and browser vendors alike.

Acknowledgements

We thank our reviewers for their valuable feedback. We also thank Tristan Hermanns who conducted preliminary experiments for this study as part of his Master Thesis. This work was conducted in the scope of a dissertation at the Saarbrücken Graduate School of Computer Science.

References

- [1] PixieBrix, “The State of Enterprise Extensions,” 2023. [Online]. Available: <https://www.pixiebrix.com/reports/state-of-browser-extensions>
- [2] Forbes, “Extending E-Commerce Revenue With Browser Extensions,” 2025. [Online]. Available: <https://www.forbes.com/sites/douglaslaney/2025/09/30/extending-e-commerce-revenue-with-browser-extensions/>
- [3] D. F. Somé, “Empoweb: Empowering Web Applications with Browser Extensions,” in *IEEE S&P*, 2019.
- [4] A. Fass, D. F. Somé, M. Backes, and B. Stock, “DoubleX: Statically Detecting Vulnerable Data Flows in Browser Extensions at Scale,” in *ACM CCS*, 2021.
- [5] S. Agarwal, “Helping or Hindering? How Browser Extensions Undermine Security,” in *ACM CCS*, 2022.
- [6] J. Yu, S. Li, J. Zhu, and Y. Cao, “CoCo: Efficient Browser Extension Vulnerability Detection via Coverage-guided, Concurrent Abstract Interpretation,” in *ACM CCS*, 2023.
- [7] A. Sjösten, S. Van Acker, and A. Sabelfeld, “Discovering browser extensions via web accessible resources,” in *CODASPY*, 2017.
- [8] E. Trickel, O. Starov, A. Kapravelos, N. Nikiforakis, and A. Doupé, “Everyone is different: Client-side diversification for defending against extension fingerprinting,” in *USENIX Security*, 2019.
- [9] Starov, Oleksii and Nikiforakis, Nick, “Xhound: Quantifying the fingerprintability of browser extensions,” in *IEEE S&P*, 2017.
- [10] P. Laperdrix, O. Starov, Q. Chen, A. Kapravelos, and N. Nikiforakis, “Fingerprinting in style: Detecting browser extensions via injected style sheets,” in *USENIX Security*, 2021.
- [11] K. Solomos, P. Ilia, S. Karami, N. Nikiforakis, and J. Polakis, “The Dangers of Human Touch: Fingerprinting Browser Extensions through User Actions,” in *USENIX Security*, 2022.
- [12] K. Solomos, P. Ilia, N. Nikiforakis, and J. Polakis, “Escaping the Confines of Time: Continuous Browser Extension Fingerprinting Through Ephemeral Modifications,” in *ACM CCS*, 2022.
- [13] S. Karami, P. Ilia, K. Solomos, and J. Polakis, “Carnus: Exploring the Privacy Threats of Browser Extension Fingerprinting,” in *NDSS*, 2020.
- [14] Splunk, “What’s The CIA Triad? Confidentiality, Integrity, & Availability, Explained,” 2024. [Online]. Available: https://www.splunk.com/en_us/blog/learn/cia-triad-confidentiality-integrity-availability.html
- [15] W3C, “Web Extensions,” 2025. [Online]. Available: <https://w3c.github.io/webextensions/specification/>
- [16] Chrome for Developers, “Service worker overview,” 2025. [Online]. Available: <https://developer.chrome.com/docs/workbox/service-worker-overview>
- [17] Chrome for Developers, “Content scripts,” 2025. [Online]. Available: <https://developer.chrome.com/docs/extensions/develop/concepts/content-scripts>
- [18] Chrome Developers, “Manifest - Web Accessible Resources,” 2025. [Online]. Available: https://developer.chrome.com/docs/extensions/mv3/manifest/web_accessible_resources/
- [19] Chrome Developers, “chrome.scripting.executeScript,” 2025. [Online]. Available: <https://developer.chrome.com/docs/extensions/reference/scripting/#method-executeScript>
- [20] Mozilla Developer Network, “Node: appendChild() method,” 2025. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/Node/appendChild>

- [mozilla.org/en-US/docs/Web/API/Node/appendChild](https://developer.mozilla.org/en-US/docs/Web/API/Node/appendChild)
- [21] Chrome for Developers, “Manifest,” 2025. [Online]. Available: <https://developer.chrome.com/docs/extensions/reference/manifest>
 - [22] Chrome Developers, “Add a popup,” 2025. [Online]. Available: <https://developer.chrome.com/docs/extensions/develop/ui/add-popup>
 - [23] Chrome Developers, “Message Passing,” 2025. [Online]. Available: <https://developer.chrome.com/docs/extensions/mv3/messaging/>
 - [24] Chrome Developers, “Isolated World,” 2025. [Online]. Available: https://developer.chrome.com/docs/extensions/develop/concepts/content-scripts#isolated_world
 - [25] S. Lekies, B. Stock, M. Wentzel, and M. Johns, “The unexpected dangers of dynamic javascript,” in *USENIX Security*, 2015.
 - [26] B. Stock, M. Johns, M. Steffens, and M. Backes, “How the Web Tangled Itself: Uncovering the History of {Client-Side} Web ({In} Security),” in *USENIX Security*, 2017.
 - [27] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk, “Scriptless attacks: stealing the pie without touching the sill,” in *ACM CCS*, 2012.
 - [28] PortSwigger, “DOM clobbering,” 2025. [Online]. Available: <https://portswigger.net/web-security/dom-based/dom-clobbering>
 - [29] W3C, “Web IDL Standard,” 2025. [Online]. Available: <https://webidl.spec.whatwg.org/#legacy-platform-object-abstract-ops>
 - [30] M. West, “Content Security Policy Level 3,” 2025. [Online]. Available: <https://w3c.github.io/webappsec-csp/#directive-script-src>
 - [31] O. C. S. Series, “HTML Sanitization,” 2025. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#html-sanitization
 - [32] S. Khodayari and G. Pellegrino, “It’s (DOM) clobbering time: Attack techniques, prevalence, and defenses,” in *IEEE S&P*, 2023.
 - [33] Z. Liu, T. Lee, J. Yu, Z. Kang, and Y. Cao, “The DOMino Effect: Detecting and Exploiting DOM Clobbering Gadgets via Concolic Execution with Symbolic DOM,” in *USENIX Security*, 2025.
 - [34] J. Jueckstock and A. Kapravelos, “Visiblev8: In-browser monitoring of javascript in the wild,” in *ACM IMC*, 2019.
 - [35] Chrome Developers, “Chrome Extensions Sitemap,” 2025. [Online]. Available: <https://chromewebstore.google.com/sitemap>
 - [36] S. Agarwal, A. Fass, and B. Stock, “Peeking through the window: Fingerprinting Browser Extensions through Page-Visible Execution Traces and Interactions,” in *ACM CCS*, 2024.
 - [37] Chrome Developers, “ExecutionWorld,” 2025. [Online]. Available: <https://developer.chrome.com/docs/extensions/reference/api/scripting#type-ExecutionWorld>
 - [38] Mozilla Developer Network, “Document: createElement() method,” 2025. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/Document/createElement>
 - [39] Mozilla Developer Network, “Object prototypes,” 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Advanced_JavaScript_objects/Object_prototypes
 - [40] Z. Kang, S. Li, and Y. Cao, “Probe the Proto: Measuring Client-Side Prototype Pollution Vulnerabilities of One Million Real-world Websites,” in *NDSS*, 2022.
 - [41] C. D. Protocol, “Profiler Domain,” 2025. [Online]. Available: <https://chromedevtools.github.io/devtools-protocol/tot/Profiler/>
 - [42] Playwright, “Playwright enables reliable end-to-end testing for modern web apps,” 2025. [Online]. Available: <https://playwright.dev>
 - [43] Mozilla Developer Network, “MutationObserver,” 2015. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver>
 - [44] Mozilla Developer Network, “Object.defineProperty(),” 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty
 - [45] S. Lekies, K. Kotowicz, S. Groß, E. A. Vela Nava, and M. Johns, “Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets,” in *ACM CCS*, 2017.
 - [46] S. Roth, M. Backes, and B. Stock, “Assessing the impact of script gadgets on csp at scale,” in *ACM CCS*, 2020.
 - [47] J. Drescher, S. Mirzaei, S. Khodayari, D. Klein, T. Barber, M. Johns, and G. Pellegrino, “In the DOM We Trust: Exploring the Hidden Dangers of Reading from the DOM on the Web,” in *ACM CCS*, 2025.
 - [48] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson, “Hulk: Eliciting malicious behavior in browser extensions,” in *USENIX Security*, 2014.
 - [49] Mozilla Developer Networks, “HTMLCollection,” 2025. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/HTMLCollection>
 - [50] Mozilla Developer Networks, “Proxy,” 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy
 - [51] S. Agarwal and B. Stock, “First, Do No Harm: Studying the manipulation of security headers in browser extensions,” in *MADWeb*, 2021.
 - [52] A. Barth, A. P. Felt, P. Saxena, and A. Boodman, “Protecting browsers from extension vulnerabilities,” in *NDSS*, 2010.
 - [53] N. Carlini, A. P. Felt, and D. Wagner, “An Evaluation of the Google Chrome Extension Security Architecture,” in *USENIX Security*, 2012.
 - [54] Y. M. Kim and B. Lee, “Extending a Hand to Attackers: Browser Privilege Escalation Attacks via Extensions,” in *USENIX Security*, 2023.
 - [55] I. Sanchez-Rola, I. Santos, and D. Balzarotti, “Exten-

sion breakdown: Security analysis of browsers extension resources control policies,” in *USENIX Security*, 2017.

- [56] O. Starov and N. Nikiforakis, “Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions,” in *WWW*, 2017.
- [57] O. Starov, P. Laperdrix, A. Kapravelos, and N. Nikiforakis, “Unnecessarily Identifiable: Quantifying the fingerprintability of browser extensions due to bloat,” in *WWW*, 2019.
- [58] K. Solomos, N. Nikiforakis, and J. Polakis, “Harnessing Multiplicity: Granular Browser Extension Fingerprinting through User Configurations,” in *ACSAC*, 2024.
- [59] A. Sjösten, S. Van Acker, P. Picazo-Sanchez, and A. Sabelfeld, “Latex Gloves: Protecting Browser Extensions from Probing and Revelation Attacks,” in *NDSS*, 2019.
- [60] S. Karami, F. Kalantari, M. Zaeifi, X. J. Maso, E. Trickle, P. Ilia, Y. Shoshitaishvili, A. Doupe, and J. Polakis, “Unleash the Simulacrum: Shifting Browser Realities for Robust {Extension-Fingerprinting} Prevention,” in *USENIX Security*, 2022.
- [61] S. Agarwal, R. Mrowczynski, M. Hellenthal, and B. Stock, “I have no idea how to make it safer: Studying Security and Privacy Mindsets of Browser Extension Developers,” in *USENIX Security*, 2025.
- [62] N. Jagpal, E. Dingle, J.-P. Gravel, P. Mavrommatis, N. Provos, M. A. Rajab, and K. Thomas, “Trends and lessons from three years fighting malicious extensions,” in *USENIX Security*, 2015.
- [63] A. Aggarwal, B. Viswanath, L. Zhang, S. Kumar, A. Shah, and P. Kumaraguru, “I spy with my little eye: Analysis and detection of spying browser extensions,” in *IEEE Euro S&P*, 2018.
- [64] R. Perrotta and F. Hao, “Botnet in the browser: Understanding threats caused by malicious browser extensions,” in *IEEE S&P*, 2018.
- [65] X. Xing, W. Meng, B. Lee, U. Weinsberg, A. Sheth, R. Perdisci, and W. Lee, “Understanding Malvertising Through Ad-Injecting Browser Extensions,” in *WWW*, 2015.
- [66] K. Thomas, E. Bursztein, C. Grier, G. Ho, N. Jagpal, A. Kapravelos, D. McCoy, A. Nappa, V. Paxson, P. Pearce, N. Provos, and M. A. Rajab, “Ad Injection at Scale: Assessing Deceptive Advertisement Modifications,” in *IEEE S&P*, 2015.
- [67] N. Pantelaios, N. Nikiforakis, and A. Kapravelos, “You’ve Changed: Detecting Malicious Browser Extensions through Their Update Deltas,” in *ACM CCS*, 2020.
- [68] E. Kenneally and D. Dittrich, “The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research,” 2012.
- [69] T. Kohno, Y. Acar, and W. Loh, “Ethical Frameworks and Computer Security Trolley Problems: Foundations for Conversations,” in *USENIX Security*, 2023.

Ethical Considerations

We followed best practices and accepted strategies for our study, in line with the established standards of our community [68, 69], and extensively evaluated the design, methodology, and impact of our study before we began our experiments. We detail our assessment in the following:

Dataset Construction: We leverage the official endpoint exposed by the Chrome Web Store to collect all the publicly listed extensions for our analysis. Thus, we do not collect any commercial extension that we are not authorized to collect. Further, since data collection is also a one-time exercise, we do not place a substantial load on the servers. Lastly, we use the downloaded extension solely for our analysis and do not distribute it for commercial purposes.

Data Collection Strategies: We make modifications within the local copies of the collected extensions that we analyze in our study. Naturally, none of these changes affect the published extension itself. Further, we run extensions only against our own test pages, which we designed for this study. Thus, we do not incur any unnecessary traffic to live Web applications at all, or that could be attributed to the extension under study. We believe our experiment caused negligible side effects to these live applications.

Disclosure: We already notified the developers of 2,019 affected extensions about the underlying threat for whom we could find the contact information on the Web store.

LLM Usage Statement

We did *not* use any Generative AI / LLM-related tools to develop or refine the research questions or the methodology of our study. Similarly, we did not use these tools to implement the code for our data collection pipeline. We also did *not* use any AI-related tools while writing, phrasing, or rephrasing the content of this paper, beyond using the text-correction tools to fix grammatical errors. We state that we used AI tools to write scripts that generate plots by supplying placeholder data, to restructure the analysis scripts already written by the authors and supplement them with comments provided by the authors themselves for readability purposes. Importantly, the authors verified each generated artifact through extensive manual checks.

Appendix A. Static Taint-Tracking Analysis

By instrumenting the JavaScript APIs that enable DOM interactions in the isolated context, we found 3,210 extensions that read or accessed potentially attacker-controllable data during our baseline INA tests, as listed in Table 7. Merely accessing DOM content, however, does not imply downstream flow to security-critical sinks, and our dynamic taint-tracking analysis missed many such flows due to runtime constraints. We therefore extended our analysis with static JavaScript taint tracking in the content scripts of these extensions, performing a CodeQL-driven inter-procedural analysis from the previously captured DOM-read sources

to our predefined API sinks (as in Table 5). We augmented the TaintTracking framework with constraints preserving taint through DOM-specific operations (e.g., *href*, *src*, *innerHTML*, and *value*) directly or via `getAttribute()`, DOM-traversal properties, *shadowRoot*, and *XPath* evaluation and its resulting accessors, compiling each extension’s content scripts into an individual database for flow identification. Through manual inspection, we found false positives from ambiguous single-letter identifiers in *jQuery* and similar libraries; we filtered these by discarding flows with a bare single-letter sink identifier and five or more distinct converging sources, as well as flows matching *jQuery*’s internal `ready`, `fx.interval`, and `fx.tick` constants. This yielded 739 additional extensions with potentially attacker-controlled data flow in the isolated namespace.

Appendix B. Additional Case Studies

We discuss three additional case studies among the 2,410 extensions, susceptible to SNA or INA, that each correspond to *a.*) apparent security impact, *b.*) disruption of functionality/usability, and *c.*) unclear impact, respectively.

1. *Security-critical data-flows in WhoMails*: We found a DOM-based XSS vulnerability in this extension, which stemmed from unsanitized DOM read within the extension’s content script (`content.js:719`), that is then dispatched via *sendResponse* to the popup that requested the data (`content.js:42`). The extension popup (`popup.js`) retrieves the content and writes back to its own HTML without any further sanity-checks. Given that the popup JavaScript is the privileged component, an attacker may inject malicious JavaScript to exploit this DOM-based XSS vulnerability.

2. *Visible Functional Breakage in frago.ai*: We found this extension’s reader-mode content-scoring logic to be disruptible by any website. The extension classifies candidate DOM containers using two competing regexes: an *unlikelyCandidates* denylist (e.g., *sidebar*, *ad*, and *popup*) and an *okMaybeltsACandidate* allowlist override (e.g., *article*, *content*, and *main*). A malicious page can wrap its genuine article text in a container whose class matches the denylist but not the override (e.g., `class="sidebar-wrapper"`), causing the extension’s scoring pass to find zero qualifying candidates. This results in the extension’s reader-mode overlay rendering empty or failing outright.

3. *Unclear Disruptive Impact in Dabblr*: We found a case of unexplained, inconsistent internal behavior whose real-world impact remains unclear. The extension injects an in-page notification element (`mint_notif_<hash>`) and schedules its own removal via a `setTimeout` call (`js/content.js:713–720`) that removes the element after a configurable delay (`obj.close_timeoute` seconds). We observed highly inconsistent code-coverage behavior across visits for this particular flow, consistent with the extension’s own auto-dismiss timer racing against our coverage instrumentation. Unlike the previous two cases, we cannot classify the final impact of these cases using automated measures due to unstable execution behavior.

Appendix C. Additional Tables & Figures

TABLE 4: DOM-facing JS APIs monitored for INA.

Prototype	Monitored APIs
Element	<code>getAttribute</code> , <code>getAttributeNS</code> , <code>getAttributeNode</code> , <code>getAttributeNodeNS</code> , <code>getAttributeNames</code> , <code>getHTML</code> , <code>getElementsByTagName</code> , <code>getElementsByTagNameNS</code> , <code>getElementsByClassName</code> , <code>querySelector</code> , <code>querySelectorAll</code>
Document	<code>getElementById</code> , <code>getElementsByName</code> , <code>getElementsByTagName</code> , <code>getElementsByTagNameNS</code> , <code>getElementsByClassName</code> , <code>getSelection</code> , <code>querySelector</code> , <code>querySelectorAll</code>
DocumentFragment	<code>getElementById</code> , <code>querySelector</code> , <code>querySelectorAll</code>
HTMLCollection	<code>namedItem</code>
Node	<code>getRootNode</code>
Range	<code>createContextualFragment</code>

TABLE 5: DOM sources / sinks monitored for taint analysis.

Method	Role	APIs / Properties
Dynamic	Source	<code>src</code> , <code>href</code> , <code>action</code> , <code>value</code> , <code>textContent</code> , <code>innerHTML</code> , <code>outerHTML</code> , <code>data</code> , <code>location</code> , <code>setAttribute</code>
	Sink	<code>fetch</code> , <code>sendMessage</code> , <code>postMessage</code> , <code>sendBeacon</code> , <code>XMLHttpRequest.send</code> , <code>eval</code> , <code>setTimeout</code> , <code>setInterval</code> , <code>chrome.storage.set</code> , <code>localStorage.setItem</code> , <code>innerHTML</code> , <code>outerHTML</code> , <code>src</code> , <code>href</code>
Static	Source	<code>getElementById</code> , <code>querySelector</code> , <code>querySelectorAll</code> , <code>getElementsByTagName</code> , <code>getElementsByClassName</code> , <code>getElementsByName</code>
	Sink	<code>fetch</code> , <code>sendMessage</code> , <code>postMessage</code> , <code>eval</code> , <code>setTimeout</code> , <code>setInterval</code> , <code>chrome.storage.set</code> , <code>location.assign</code> , <code>location.replace</code> , <code>chrome.tabs.create</code> , <code>chrome.tabs.update</code> , <code>innerHTML</code> , <code>outerHTML</code> , <code>src</code> , <code>href</code> , <code>textContent</code>

TABLE 6: Top 10 JS API hooks that affected extensions in the Same Namespace (SNA).

Vector	Affected Extensions
<code>Object.defineProperty</code>	196
<code>EventTarget.addEventListener</code>	179
<code>Object.hasOwnProperty</code>	157
<code>Array.forEach</code>	151
<code>Array.push</code>	149
<code>Function.bind</code>	145
<code>Object.create</code>	138
<code>Object.keys</code>	130
<code>RegExp.test</code>	122
<code>String.replace</code>	118

```

1 // extension-injected JavaScript to remove ads
2 let adElement = document.getElementById('ad-promo');
3 adElement.parentElement.remove()
4
5 // website-injected JavaScript to allow ads
6 let nativeDefinition =
7   ↪ Document.prototype.getElementById;
8 Document.prototype.getElementById = function() {
9   if (Array(...arguments).includes('ad-promo')) {
10     return null;
11   } else {
12     return nativeDefinition.apply(this,
13   ↪ arguments);
14   }
15 }

```

Listing 4: Potential vulnerabilities in extension-injected code due to shared JavaScript namespace with page JavaScript

```

1 function _instrumentAPI(object, property) {
2   let nativeDefinition = object[property];
3   let result = nativeDefinition.apply(this,
4   ↪ arguments);
5   if (result == null) {
6     result = _injectElementOrAttribute.apply(this,
7   ↪ [...arguments]);
8   }
9   logAPIInvocation(object, property, result);
10  return _proxifyResult(result);
11 }

```

Listing 5: An abstract view of our instrumentation strategy for content scripts to capture DOM interactions.

TABLE 7: Potentially sensitive properties accessed by isolated content scripts through clobbering payloads in INA.

Source Properties	Injection type	Extensions
<i>Content & value injection</i>		
↪ value	Form value / data read	483
↪ textContent	Text node overwrite	331
↪ data	CharacterData payload	3
<i>URL & navigation injection</i>		
↪ src	Script/image/iframe URL	440
↪ href	Anchor / link target	2,086
↪ action	Form submission URL	24
↪ location	Page navigation	2
<i>HTML injection</i>		
↪ innerHTML	Arbitrary HTML subtree	102
↪ outerHTML	Replace entire element	18
↪ insertAdjacentHTML	Adjacent HTML injection	38
<i>Attribute injection</i>		
↪ setAttribute	Arbitrary attribute write	280
Extensions accessing any source properties		3,210

```

1 // Feature-Crop.76584aa8.js line 1 (minified)
2 let c = t
3   ?? document.querySelector("textarea")?.value //
4   ↪ col 269137
5   ?? document.getElementById("sb_form_q")?.value; //
6   ↪ col 269196
7 if (c) {
8   S.current.interceptedText = c;
9   // ...
10 }

```

Listing 6: Silent failure in [Passei Direto](#) extension through selector-based payload. The payload allows attackers to disrupt further DOM-interactions.

TABLE 8: Statistics on clobbering payloads per extension for INA. Each payload correspond to either of the 3 sources: runtime traces, AST parsing, or bundler-specific gadgets.

Category	Extensions
<i>Payload data sources (extensions contributing each source, non-exclusive)</i>	
– Runtime hook log (selector-based)	19,114
– Static AST analysis	28,478
– Clobberable window variables (GBS)	1,323
– Document access patterns	995
– Bundler gadgets	60
Total distinct extensions	40,213
<i>Clobber element types (by runtime query call count)</i>	
– CSS selector (querySelector(All))	64.9%
– Attribute (getAttribute)	14.9%
– ID (getElementById)	9.5%
– Tag (getElementsByTagName)	9.1%
– Class (getElementsByClassName)	1.5%
– Name (getElementsByName)	0.1%
<i>Payload size (bytes per extension)</i>	
↪ median / 95 th percentile / max	186 B / 1,920 B / 1,826,077 B
<i>Clobber targets per extension</i>	
↪ median / 95 th percentile / max	3 / 32 / 18,852

TABLE 9: Top 10 JS API vectors for INA that affected extensions leading to functional disruptions and silent failures.

Vector	FD	SF	FD ∪ SF	Hits
Document.querySelectorAll	72	9	80	55,387
Document.querySelector	40	1	40	1,373
Element.getAttribute("href")	36	4	39	3,015
Document.querySelectorAll("style[data-emotion]:not([data-s])")	34	–	34	235
Document.querySelectorAll("style[data-emotion]='\css\']")	34	–	34	127
Document.querySelectorAll("script")	32	–	32	181
Element.getElementsByTagName("**")	25	4	28	196
Element.querySelector	27	1	28	1,644
Element.getAttribute("disabled")	24	4	27	190
Element.getAttribute("className")	24	4	27	190

```

1 // client/main.js
2 var ComplianceHandler = class {
3   ...
4   getCompliancePopup() {
5     // line 37483
6     return
7     ↪ document.getElementsByName("aosp-compliance").length
8     ↪ > 0 ?
9     ↪ document.getElementsByName("aosp-compliance")[0] :
10    ↪ null;
11  }
12  isNotificationAlreadyDisplayed() {
13    // line 37489
14    return this.getCompliancePopup() !== null;
15  }
16  get shouldShow() {
17    // line 37463
18    return ... &&
19    ↪ !this.isNotificationAlreadyDisplayed();
20  }
21  ...
22 }

```

Listing 7: Functional disruption in [Avast Online Security & Privacy](#) extension through DOM-gadgets. The led to disabling the compliance-related notification to the user.