TriChord: Extension-Oriented Browser Fuzzing for Chromium

Huinian Yang
China Telecom Cloud Technology Co., Ltd.
Beijing, China
yanghn1@chinatelecom.cn

Daoyuan Wu*
Lingnan University
Hong Kong SAR, China
daoyuanwu@ln.edu.hk

Qingyu Li
China Telecom Cloud Technology Co., Ltd.
Beijing, China
liqy43@chinatelecom.cn

Yiming Liu
China Telecom Cloud Technology Co., Ltd.
Beijing, China
liuym53@chinatelecom.cn

Abstract

Browser extensions are widely used for extending browser functionality, but they also introduce a security-critical attack surface. To support capabilities such as content modification and ad blocking, browsers expose a rich set of privileged APIs to extensions. These APIs form a bridge between potentially untrusted extension code and privileged browser subsystems, so implementation flaws can allow malicious extensions to violate browser security boundaries and escalate privileges. However, whereas prior research has primarily focused on detecting malicious extensions or analyzing flaws in extension architectures, *vulnerabilities in the browser's own implementation of extension APIs remain largely unexplored.*

In this paper, we present TriChord, a fuzzing framework specifically designed to detect browser vulnerabilities in extension API implementations. Our key insight is that, although extension APIs are routinely exercised for benign use cases, subtle vulnerabilities can emerge when they interact with core browser subsystems, coexisting extensions, and user-driven events. TriChord systematically explores this surface by generating subsystem-oriented extension API sequences and coordinating their execution across isolated extensions, privileged browser subsystems, and simulated user actions. This extension-oriented workflow constructs browser states that the existing browser fuzzers do not exercise. By evaluating on multiple versions of Chromium, the open-source base of Chrome, TriChord identified 28 previously unknown security bugs, of which 10 were confirmed by Google and 8 were assigned CVEs, resulting in $73,000 in bug bounty rewards. Moreover, TriChord outperforms existing browser fuzzers in vulnerability discovery, and its code coverage subsumes the baselines' coverage while additionally reaching extension-driven code they do not cover, highlighting the need for specialized security testing of privileged extension APIs.

CCS Concepts

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

*Corresponding author.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
CCS '26, The Hague, Netherlands.
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846622

Keywords

Software Security, Fuzzing, Browser Security, Browser Extension, API Testing, Vulnerability Discovery

ACM Reference Format:
Huinian Yang, Daoyuan Wu, Qingyu Li, and Yiming Liu. 2026. TriChord: Extension-Oriented Browser Fuzzing for Chromium. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3846622

1 Introduction

Extensions are widely adopted for extending the functionality of modern browsers [9, 32, 34, 52]. The Chrome Web Store [20] alone hosts more than 170,000 extensions [10], serving hundreds of millions of users [57]. These extensions enable powerful features such as ad blocking, password management, automation, and privacy protection. To support such functionality, browsers expose privileged extension APIs [8] that provide access to internal browser features unavailable to ordinary web applications. As shown in Figure 1 in §2, these APIs form a bridge between isolated extension processes and core browser subsystems, including tab management, task management [38], networking, and storage.

To ensure security while enabling these capabilities, modern browsers adopt a strict multi-process architecture [5] and sandboxing techniques [11]. Extensions execute in dedicated processes and interact with privileged browser subsystems only through well-defined APIs [8]. This design helps preserve security boundaries, but it also turns extension APIs into a security-critical interface between potentially untrusted extension code and privileged browser logic. Consequently, flaws in extension API implementations can allow malicious extensions to violate browser security boundaries, escalate privileges, or even contribute to sandbox escapes.

The security of browser extensions has long attracted research attention. Most prior work falls into two categories. The first studies malicious or privacy-invasive extensions [6, 26, 39, 53], focusing on malicious code detection, privacy violations, and mitigation strategies. The second analyzes architectural flaws in extension systems [7, 29, 56], such as permission abuse, privilege escalation, and isolation failures. However, both lines of work primarily examine extensions' own logic or manifest settings, rather than vulnerabilities in the browser's own implementation of extension APIs and their interactions with browser subsystems.

In parallel, browser fuzzing has made substantial progress. State-of-the-art fuzzers [19, 55, 60] are effective at finding bugs in web

APIs [33], JavaScript engines, and other browser-facing components. Yet these tools are not designed to exercise extension APIs effectively. Testing extension APIs requires valid extension contexts, correctly structured CRX packages [16, 48], appropriate manifest configurations, and interactions with privileged browser components. As a result, the attack surface exposed by extension APIs, especially the boundary between extensions and browser subsystems, remains insufficiently explored.

Vulnerabilities in extension APIs or privileged browser processes can have serious consequences, including privilege escalation, browser compromise, and sandbox escapes. Despite the severity of this threat, security issues arising at the interface between extensions and browser internals remain under-studied. Even recent work such as Moreno et al. [35] focuses primarily on API misuse in debugging and developer-facing settings, rather than on implementation vulnerabilities in browser processes themselves. This leaves a clear gap: *the lack of specialized security testing that targets extension APIs together with their deeper interactions with browser internals.* In particular, fuzzing extension APIs imposes a combination of target-specific requirements that existing browser fuzzers do not handle [19, 55, 60]:

- *Extension-specific reachability:* Extension API handlers cannot be reached from ordinary web pages alone. Tests must run as installed extensions with valid manifests and execution contexts before their calls are dispatched to privileged browser-process handlers [16, 48].
- *Subsystem-oriented state construction:* Related state mutations span API namespaces and auxiliary operations. Exercising one privileged subsystem therefore requires combining APIs according to the browser state they jointly affect, rather than merely producing type-correct calls within one namespace.
- *Multi-principal execution:* Vulnerability-triggering states can depend on several isolated extensions acting on a shared browser subsystem while user-driven lifecycle events occur. A useful harness must coordinate these actors rather than execute a call sequence in a single page or process.

We do not claim that multi-file packaging, permission declaration, API dependencies, or event-driven execution are individually new fuzzing problems. The distinguishing challenge is their integration into an extension-oriented workflow that constructs privileged browser states jointly influenced by multiple isolated extensions and user actions. Existing browser fuzzers evaluated [19, 55, 60] later in this work do not instantiate these states: our coverage analysis shows that TriChord subsumes 89.7% of their combined coverage on shared browser-process code while reaching 6,053 extension-handler lines that none of them exercises (§5.2).

In this paper, we present TriChord, the first dedicated fuzzing framework for discovering vulnerabilities in browser extension API implementations and their interactions with core browser subsystems. TriChord addresses the above challenges by: ① generating subsystem-oriented API sequences that combine cross-namespace operations affecting the same privileged browser state; ② coordinating multiple isolated test extensions with simulated user and lifecycle events to expose state interactions that single-context fuzzing misses; and ③ automating the end-to-end workflow from valid test generation and installation through execution and crash collection.

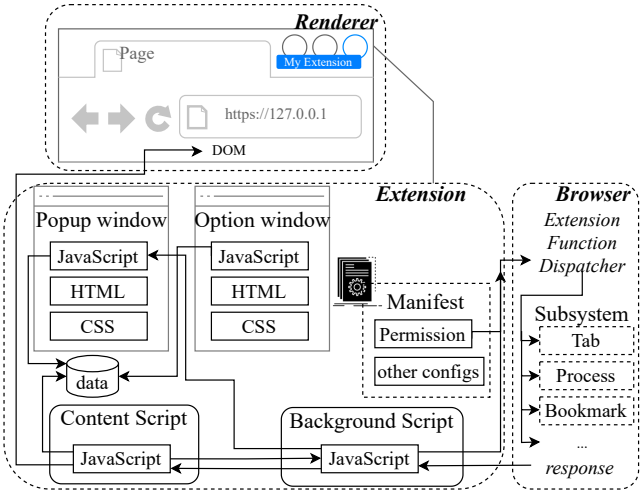

Figure 1: Extension Architecture in Chromium.

Our evaluation demonstrates the practical impact of TriChord. Across four major Chromium versions, TriChord consistently outperforms state-of-the-art browser fuzzers, including Domato [19], FreeDom [55], and Minerva [60], in both vulnerability discovery and code coverage. Compared to these baselines, TriChord achieves up to 36.24% higher region coverage and 28.09% higher function coverage on the privileged browser codebase, and uncovers 28 previously unknown security bugs across multiple browser subsystems and extension interfaces. Our ablation study further shows that coordinated multi-extension orchestration and event-driven user interaction simulation are essential for triggering complex vulnerabilities missed by traditional fuzzers. These results underscore the need for specialized, extension-oriented security testing of privileged browser APIs.

Contributions. To summarize, we make the following contributions in this paper:

- We identify the browser-side implementation of privileged extension APIs as an under-tested attack surface and conduct the first systematic fuzzing study of this surface.
- We design, implement, and release TriChord, a framework that combines subsystem-guided state construction with coordinated multi-extension and user-event orchestration.
- We show empirically that this design subsumes 89.7% of the baselines' combined coverage on shared browser-process code, reaches 6,053 extension-handler lines absent from all baselines, and uncovers 28 previously unknown Chromium vulnerabilities, including 10 confirmed by Google, 8 assigned CVEs, and $73,000 in bug bounty rewards.

2 Background

2.1 Browser Extensions

Browser extensions are software modules that customize and extend browser functionality beyond the original browser distribution. Within the browser's multi-process architecture, they occupy a privileged position between ordinary web content and the browser's privileged process. As a result, a single extension can

influence multiple renderer processes; for example, an ad blocker may apply filtering rules across all open tabs. Conversely, a single webpage may be affected by multiple concurrently running extensions that inject scripts, intercept network requests, or manipulate the DOM.

An extension is itself a composite application composed of several component types with distinct responsibilities and security contexts, as illustrated in Figure 1:

- **Manifest** [24]: A JSON configuration file that defines the extension's permissions, scripts, and other metadata such as icons and background behavior.
- **Content scripts**: JavaScript files that execute in web pages and can observe or manipulate the DOM. They have limited access to extension APIs and typically rely on explicit message passing to request privileged operations from the background context.
- **Background scripts**: The core logic of the extension, implemented as either a persistent background page (Manifest V2) or a service worker (Manifest V3). It can access the full set of extension APIs permitted by the manifest and coordinates extension-wide activities such as handling browser events, maintaining state, and communicating with other components, while remaining isolated from direct DOM access.
- **UI components**: Interactive surfaces such as popup windows, options pages, and sidebars through which users configure or control the extension.

Major browsers, including Chrome[1], Firefox, Edge, and Safari, follow the WebExtensions [49] model maintained by the W3C Community Group. This model originated from Chrome's extension system and now enables developers to port extensions across browsers with relatively minor modifications. Table 1 compares the major extension ecosystems and shows Chrome's dominance in both market share and ecosystem size [10]. Chrome accounts for over 67% [46] of the browser market and hosts more than 170,000 extensions, far surpassing Firefox and Edge in both metrics. This scale is a key reason why we focus our study on Chrome-compatible extensions and their security implications.

2.2 Subsystems and Extension APIs

As shown in Figure 1, the browser process grants extensions advanced capabilities through a rich set of *extension APIs*, which provide controlled access to internal **subsystems**. These subsystems are organized by functionality and partitioned for permission isolation. Examples include the *tab subsystem*, which manages tab creation and rearrangement; the *web request subsystem*, which handles network observation and modification; and the *DevTools subsystem*, which supports debugging and analysis. To preserve isolation, each extension typically runs in its own renderer process, which appears in the browser task manager as "Extension: Name".

[1]Chromium is an open-source browser project that serves as the foundation for Google Chrome and numerous other browsers [12]. The research targets examined in this work, extensions and privileged browser processes, are directly applicable to Chrome. Although Google Chrome incorporates proprietary features [12] in addition to the open-source Chromium base, both platforms share the same extension API architecture.

Table 1: Comparing Major Browser Extension Ecosystems.

	Chrome	Firefox	Edge
Market Share	67.94%	2.45%	5.07%
# Extensions	176,465	50,954	26,150
# Extension APIs	≈ 560	≈ 370	≈ 470

This process separation improves both security and lifecycle management.

Extension APIs are browser-exposed functions and events that allow extensions to access and manipulate browser features through JavaScript. They are organized into namespaces such as `chrome.tabs` and `chrome.bookmarks`, each containing related methods and events. When an extension script invokes an API such as `chrome.tabs.create`, the browser first checks whether the required permission is declared in the extension's manifest. The invocation is then translated into an inter-process communication (IPC) message carrying the API name and parameters from the extension process to the browser process. Inside the browser process, the extension host receives this IPC message and dispatches it to the appropriate subsystem, such as the tab manager. After the requested operation completes, the browser process returns the result via IPC, and the extension script receives it through a callback or event handler.

Example. A simple example is opening a new browser tab. The extension code might call:

```
chrome.tabs.create(
    {url: "https://www.example.com"},
    function(tab) {
        console.log("Created tab:", tab.id);
    }
);
```

This call triggers an IPC message to the browser process, which creates the new tab and returns its details to the extension via the callback. In this way, extensions interact with browser internals through permission-controlled and isolated interfaces, while sensitive operations remain confined to the privileged browser process.

Nevertheless, extensions hold substantially higher privileges than ordinary web content. To enable advanced functionality, browsers must expose sensitive capabilities such as tab management, network interception, and access to user data through extension APIs. Even with permission checks and process isolation, these APIs still form a controlled bridge from extension code to privileged browser internals. This necessary exposure enlarges the attack surface: implementation bugs, isolation failures, or subtle permission-enforcement flaws may be exploitable by malicious or compromised extensions to escalate privileges or escape browser sandboxes. *Extensions therefore constitute an important attack vector against core browser security guarantees.*

3 Design

Building on the extension architecture and attack surface described in §2, we now present the design of TriChord. TriChord is a subsystem-oriented fuzzing framework that targets vulnerabilities in browser-side extension API implementations and their interactions with privileged browser subsystems. Its design is guided by

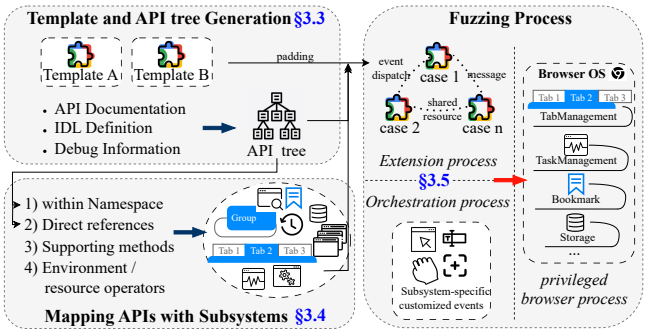

Figure 2: An overview of the subsystem-oriented extension generation and fuzzing processes in TriChord.

three requirements: generating valid extension contexts, steering exploration toward security-critical browser subsystems, and exercising behaviors that arise only under realistic multi-extension and user-driven scenarios.

3.1 Overview

The overall architecture of TriChord is shown in Figure 2. At a high level, TriChord combines subsystem-oriented extension generation with coordinated runtime orchestration to drive privileged browser code through extension APIs. Its workflow consists of three main components, followed by an assembly step that turns these components into executable test extensions.

The first component (§3.3) uses two manually designed extension templates and a hierarchical API tree automatically extracted from Chromium schemas, providing the structural foundation for test generation.

The second component (§3.4) consumes a one-time, manually curated API-to-subsystem map. During fuzzing, TriChord automatically uses this map to group related APIs and target specific browser functionality.

The third component (§3.5) drives runtime exploration. It coordinates multiple generated extensions through event dispatching, message passing, and shared-resource contention, while also simulating user interactions to trigger behaviors that depend on realistic execution contexts inside the privileged browser process.

These components come together in the subsystem-oriented extension generation process described in §3.2, which assembles subsystem-relevant API sequences into concrete extensions ready for fuzzing.

The distinguishing role of this design is not API dependency handling alone. The subsystem organization directs independently generated extensions toward the same privileged browser state, while orchestration combines their actions with browser and user-driven lifecycle events. The resulting unit of exploration is therefore a multi-principal browser state, rather than an API sequence executed within a single page or harness.

The remainder of this section details each component in turn.

3.2 Subsystem-Oriented Extension Generation

This subsection introduces the overall process of how TriChord assembles concrete test extensions from subsystem-relevant APIs.

Algorithm 1: Subsystem-Oriented Extension Generation

Input: API tree T, Subsystem Classification S, Extension Templates {TemplateA, TemplateB}
Output: Set of test extensions E

1 **Procedure** *GenerateExtensions*(T, S)
2 Select target subsystem s from S;
3 Identify related APIs $R = \{api \mid api \in S$ and api relates to subsystem $s\}$;
4 Initialize $C \leftarrow \emptyset$; `// Empty set of API calls`
5 **for** $i = 1$ **to** k **do** ; `// Generate k different API calls`
6 Select $api \in R$;
7 Parse parameter types from T for selected api;
8 Generate valid parameter values according to T;
9 $call \leftarrow$ Instantiate api with generated parameters;
10 Add $call$ to C;
11 **end**
12 Initialize $E \leftarrow \emptyset$; `// Empty set of extensions`
13 Select appropriate template from {TemplateA, TemplateB};
14 Place API calls C into placeholder interfaces in the selected template;
15 Add assembled extension to E;
16 **return** E;

The key idea is to combine the structural constraints encoded in the extension templates and the API tree (§3.3) with the subsystem-level API relationships identified in §3.4. By doing so, TriChord generates extensions that are both syntactically valid and semantically focused on a target browser subsystem.

Algorithm 1 summarizes the extension assembly process in TriChord. The procedure first selects a target subsystem and retrieves the set of APIs associated with that subsystem according to the mapping described in §3.4 (lines 1–3). This step narrows generation to interfaces that are likely to exercise the same privileged browser functionality.

Next, TriChord samples k APIs from this subsystem-specific set and instantiates them into executable calls (lines 4–11). For each selected API, TriChord consults the API tree to resolve parameter types, required fields, value constraints, and dependency information, and then generates concrete arguments that satisfy these constraints. This use of the API tree ensures that the resulting calls are not merely random strings, but valid extension API invocations that can pass basic structural checks and reach deeper browser logic.

Finally, TriChord inserts the instantiated calls into placeholder locations in a suitable extension template (lines 12–16). The choice between Template A and Template B depends on whether the target APIs require richer extension contexts or can be exercised within a minimal setup. The output is a set of concrete, subsystem-oriented test extensions that are ready for execution and further orchestration during fuzzing.

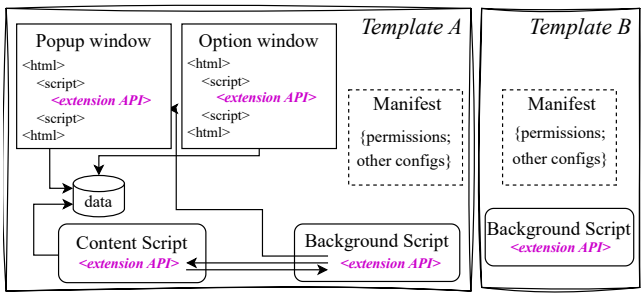

Figure 3: Two template variants used by TriChord.

3.3 Extension Templates and API Tree

To generate executable extension test cases, TriChord requires both valid extension scaffolds and a structured representation of the extension API surface. It therefore relies on two complementary configuration artifacts: extension templates, which provide executable contexts, and an extension API tree, which provides the structural information needed to instantiate valid API calls.

Templates. Browser extensions are heterogeneous packages composed of multiple file types and execution contexts, including HTML, CSS, and JavaScript components. As a result, extension fuzzing must generate not only API calls, but also executable extension structures in which those calls can run correctly. To address this challenge, we manually designed two reusable variants: a comprehensive template (Template A) and a minimal template (Template B), as illustrated in Figure 3.

Template A includes the architectural components commonly found in modern extensions, including `manifest.json`, background scripts, content scripts, popup pages, and options pages. This richer structure supports APIs whose behavior depends on communication across multiple contexts, such as message passing, storage access, and UI-triggered interactions. By preserving these components, Template A enables fuzzing of behaviors that span multiple execution contexts and would be missed by a minimal setup.

In contrast, Template B contains only the essential components for extension functionality: `manifest.json` and background scripts. This streamlined template targets the large portion of extension APIs that can be exercised without auxiliary UI or content-script contexts. Its smaller execution footprint improves fuzzing throughput and makes it better suited for API-focused testing when richer extension structure is unnecessary.

Template Construction Process. We derived these templates from Chrome extension documentation and official test samples, and manually added the full set of permissions to their `manifest.json`. At runtime, TriChord automatically selects the Manifest V2 or V3 template and inserts generated API sequences into the background context placeholders.

API Tree. The extension API tree is the second foundation of TriChord. It provides a hierarchical and semantically enriched representation of the extension API surface, allowing TriChord to generate calls that respect API structure, parameter constraints, and inter-API dependencies. TriChord automatically constructs the API tree by parsing Chromium's Interface Definition Language

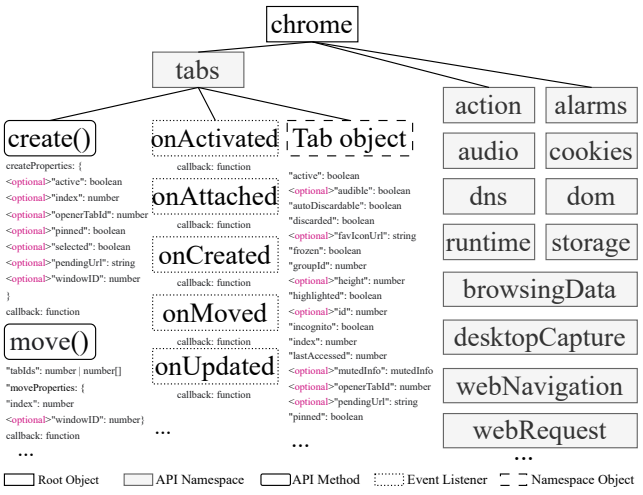

Figure 4: The automatically extracted extension API tree used by TriChord.

(IDL) files and JSON specifications; API documentation and runtime debug information are used only for the subsequent manual refinement of grammar rules described in §4.

As shown in Figure 4, nodes in the API tree represent extension API components of different types. The root corresponds to the global extension object (`chrome`), while the first-level children correspond to namespaces such as `tabs`, `storage`, and `runtime`. Each namespace node further contains method, event, and property nodes.

Each node stores metadata used to guide test generation:

- **Method nodes** store parameter types, required and optional fields, valid value ranges, and dependency constraints. For example, the `chrome.tabs.query` node records the structure and valid properties of the `queryInfo` object.
- **Event nodes** maintain listener registration patterns, callback signatures, and event dispatch conditions. For instance, the `chrome.runtime.onMessage` node captures the expected message format and response-handling pattern.
- **Property nodes** record access modes (read, write, or both) and data types.

This structured representation enables TriChord to generate syntactically correct and semantically meaningful API invocations for the script generator (§4). It also helps preserve data dependencies across calls and systematically exercise edge cases in parameters and execution contexts during fuzzing.

3.4 Mapping APIs with Their Subsystems

Although the API tree captures the structure of the extension API surface, it does not by itself indicate which APIs jointly exercise the same privileged browser functionality. Effective fuzzing therefore requires a second layer of organization that maps APIs to the browser subsystems they affect. We manually construct this mapping once; during fuzzing, TriChord automatically consumes it so that generated tests concentrate on subsystem-relevant behaviors rather than individual APIs in isolation.

Table 2: API classification statistics for major subsystems.

Subsystem	Number of APIs
Action	22
Alarms	15
Bookmarks	28
BrowsingData	21
ContextMenus	12
Extension	36
FontSettings	27
History	19
Notifications	10
Omnibox	17
Process	18
Runtime	74
Search	12
Sessions	16
Status	18
Storage	15
TabCapture	11
TabGroups	19
Tabs	83
WebNavigation	39
Windows	38

API Relationship Classification. TriChord adopts a method-level classification strategy based on four types of relationships:

(1) *Namespace:* methods, events, and properties declared directly under the subsystem's namespace.
(2) *Direct references:* APIs that reference subsystem objects in return values, parameters, or listener signatures according to the API tree.
(3) *Supporting methods:* APIs that prepare, create, or manipulate the execution context required for core subsystem operations.
(4) *Environment and resource operators:* APIs that manage resources or environmental state surrounding the subsystem.

Together, these relationships let TriChord capture not only obvious subsystem APIs, but also auxiliary interfaces that are often necessary to drive deeper browser states.

Example: The tabGroups Subsystem. To illustrate the mapping process, consider the *tabGroups* subsystem, which allows users to group and manage related tabs. Its associated APIs span all four relationship types:

- Namespace: `tabGroups.*` APIs provide direct operations for creating, querying, updating, and moving tab groups, as well as related events such as *onCreated*, *onUpdated*, and *onRemoved*.
- Direct references: APIs that reference *groupId*, such as `tab.groupId`, `chrome.tabs.group`, `chrome.tabs.query`, and `chrome.tabs.onUpdated.addListener`, also participate in tab-group behavior.
- Supporting methods: APIs such as `chrome.tabs.create`, `chrome.tabs.get`, and `chrome.tabs.duplicate` create or manipulate tabs that may later be grouped.
- Environment and resource operators: APIs such as `chrome.windows.remove` and `chrome.processes.terminate` affect the surrounding execution environment in which tab groups exist.

This example shows that meaningful subsystem testing must include both primary interfaces and auxiliary APIs that shape subsystem state before, during, or after core operations.

Subsystem Coverage and Classification Statistics. Applying this strategy across the Chromium extension API surface yields a broad set of subsystem-specific API groups. Table 2 summarizes the resulting coverage. For example, the `Tabs` subsystem contains 83 relevant APIs, followed by `Runtime` (74), `WebNavigation` (39), and `Windows` (38). Other well-covered subsystems include `Bookmarks`, `FontSettings`, `TabGroups`, and `Extension`. This mapping gives TriChord a principled basis for targeting complex subsystem interactions and overlooked behaviors that would be difficult to expose through API-by-API fuzzing alone.

3.5 Orchestrating Testing of Extensions

The third component of TriChord orchestrates the testing workflow by coordinating the execution of multiple extension test cases and simulating realistic user interactions. This design enables deeper and more comprehensive exploration of browser extension security boundaries compared to conventional single-extension testing.

Multi-Extension Coordination. TriChord generates multiple extension variants that operate within the same target subsystem. Executing these variants together allows the fuzzer to exercise interactions that naturally arise when extensions coexist in a real browser environment. The coordination between extensions—such as which events one can trigger for another—are derived from the inter-API dependencies encoded in the API tree (§3.3). This coordination is particularly useful for exposing race conditions, ordering bugs, and resource contention vulnerabilities, and it improves exploration in three ways:

(1) *Event handling:* one extension instance can register listeners for browser events while another actively triggers those events, creating realistic interleavings across extension instances.
(2) *Runtime communication:* communication through APIs such as `runtime.message` allows TriChord to test behaviors involving inter-extension data exchange and callback-driven execution.
(3) *Shared resource concurrency:* simultaneous access to shared browser resources across isolated extensions helps uncover concurrency bugs and edge cases that single-extension fuzzing is unlikely to trigger.

User Interaction Simulation. Many extension-triggered behaviors also depend on user actions rather than API sequences alone. Extensions expose functionality through popup interfaces, options pages, context menus, and browser UI elements, making user interaction an important part of the attack surface. To capture these behaviors, TriChord augments API fuzzing with simulation of subsystem-relevant user actions, such as mouse clicks in designated regions, drag-and-drop operations, keyboard shortcuts, and navigation to specific URLs. For example, when targeting the `tab`

subsystem, TriChord may click and drag tab elements in the browser window; when targeting bookmarks, it may perform rapid and repeated clicks on bookmark entries. These actions help trigger UI-dependent state transitions and event sequences that would otherwise remain unexplored.

Orchestration. During fuzzing, each round is paired with a dedicated orchestration process that combines subsystem-oriented extensions with the corresponding user-interaction schedule. This process controls when extensions execute, when events are triggered, and when user actions are simulated, thereby creating richer interaction patterns inside the privileged browser process. In effect, orchestration turns individually valid extension test cases into coordinated executions that expose deeper subsystem states and vulnerability-triggering conditions.

4 Implementation

We implement TriChord as a Python prototype of about 4,000 lines of code. The implementation follows the three-component design described in §3: the extension script generator, the API-to-subsystem mapping, and the orchestration and automation.

Configuration Workflow. Before fuzzing, TriChord uses five configuration artifacts with explicit automation boundaries. ① *API tree.* For each browser version, TriChord automatically constructs the tree by parsing Chromium's JSON schemas and IDL files; this step requires no manual correction. ② *Templates.* We manually designed Templates A/B based on Chrome's documentation and official test samples and added the full permission set to `manifest.json`. At runtime, TriChord automatically selects a template and inserts generated API calls into its placeholders.

The extension script generator builds on Domato [19] and reuses its variable generation and reference-tracking mechanisms to preserve dependencies across API calls. ③ *Grammar rules.* TriChord automatically generates approximately 600 initial rules across 45 namespaces from the API tree. Adapting these rules to new or changed APIs requires manual refinement based on API documentation and runtime debug output, particularly when discrepancies arise between a browser release and the corresponding documentation. This refinement is repeated whenever we adopt a new Chromium version. ④ *Subsystem mapping.* We manually categorized approximately 200 APIs into 21 subsystems. Namespace and direct-reference associations are derived automatically from the API tree, whereas supporting-method and environment/resource associations are annotated manually. This manual effort covers 40 supporting-method associations and 31 environment/resource associations. Both types of annotation are one-time refinements that improve coverage quality but are not required for TriChord to operate. TriChord consumes the resulting map automatically during test generation. ⑤ *UI interactions.* We manually specified the 12 interaction patterns encoded in 13 distinct `xdotool` grammar rules; their selection, scheduling, and dispatch through `xdotool` [27] are fully automated.

After these one-time artifacts are configured, the approximately 500-line orchestration framework runs without human supervision, automating test generation, extension packaging, browser-instance management, multi-extension execution, user-interaction scheduling, crash detection, and stack-signature collection. Root-cause verification and vulnerability reporting are performed manually after a campaign and are not part of automated fuzzing workflow.

5 Evaluation

In this section, we evaluate TriChord according to the experimental setup described in §5.1, and present results addressing three research questions (RQs), detailed from §5.2 to §5.4.

5.1 Experimental Setup

RQ1: Benchmarks. To assess the effectiveness of TriChord, we compare it against three state-of-the-art browser fuzzers: ① *Domato* [19] (commit `fadff39`): a grammar-based fuzzer tailored for DOM APIs with broad browser compatibility; ② *FreeDom* [45]: a full-fledged fuzzer specializing in DOM API testing; ③ *Minerva* [58]: a browser fuzzer leveraging dynamic mod-ref analysis to target browser APIs. We evaluate all four tools on four major Chromium versions in order to capture the impact of API evolution and the transition from Manifest V2 to V3:

- *Version 1:* Publish DEPS for 85.0.4151.0 [13];
- *Version 2:* Publish DEPS for 93.0.4544.0 [14];
- *Version 3:* Publish DEPS for 114.0.5720.2 [15];
- *Version 4:* Publish DEPS for 137.0.7118.3 [17].

All evaluated fuzzers are generation-based and do not require seed inputs. For RQ1, we report two primary metrics: (1) code coverage on the core browser codebase (i.e., `src/chrome/browser` [1]), and (2) the number of crashes and deduplicated security issues discovered.

RQ2: Ablation Studies. To understand the contribution of TriChord's major design components, we evaluate three reduced variants: ① *Basic Extension API Support:* A baseline variant that randomly executes extension APIs, without subsystem-oriented clustering or orchestration; ② *No Coordinated Extensions:* Disables multi-extension orchestration, restricting execution to a single active extension at a time; ③ *No User Interaction Simulation:* Removes all synthetic user interaction, preventing event-driven process triggers. These variants allow us to isolate the effects of subsystem-oriented clustering, multi-extension orchestration, and user-interaction simulation on coverage and vulnerability discovery.

RQ3: Security Study. To assess TriChord's practical capability to discover previously unknown vulnerabilities, we conduct a longitudinal security study spanning 2020–2025 across more than 20 Chromium versions from both stable and development channels. This setup allows us to evaluate the framework across the evolution of extension APIs, including both Manifest V2 [21] and Manifest V3 [22]. For each version, we run fuzzing campaigns for at least 7 days, configuring TriChord to target all identified subsystems while giving additional attention to newly introduced features.

Environment Configuration. All experiments were performed on two identical machines running 64-bit Ubuntu 20.04 LTS, each equipped with an Intel(R) Core(TM) i5-14400 CPU and 32 GB RAM. Each fuzzing instance uses a 30-second timeout per input. For evaluation, we compile the target browser with the following build configuration:

- *ASan build [30]:* For memory safety detection using AddressSanitizer;

- *Clangcov build [31]:* For code coverage collection using Clang's source-based instrumentation, providing region, function, and line-level execution information.

5.2 RQ1: Comparison with Existing Fuzzers

To answer RQ1, we compare TriChord with the three browser fuzzers introduced in §5.1 across four Chromium versions. We focus on two criteria: coverage of security-critical browser code and bug discovery effectiveness, as summarized in Figure 5 and Table 3.

Code Coverage. Figure 5 compares the code coverage achieved by each fuzzer during a 12-hour campaign over the `src/chrome/browser` directory.

To ensure a fair comparison across tools with different design goals, we restrict the coverage target to shared browser-process code. Specifically, we exclude extension-specific directories such as `src/chrome/browser/extensions` and `src/extensions`, which would unfairly favor TriChord, as well as renderer-centric directories such as `third_party/blink`, where DOM-oriented fuzzers naturally dominate. We therefore evaluate coverage on `src/chrome/browser`, which is accessible to all three baseline fuzzers and TriChord and, architecturally, corresponds to Chromium's privileged browser process.

The results are reported using three metrics: line, region, and function coverage. In the top row of Figure 5, TriChord consistently achieves the highest line coverage across all four versions. On average, TriChord surpasses Domato, FreeDom, and Minerva by 30.27%, 29.80%, and 23.84%, respectively. The bottom row further shows that TriChord maintains a substantial advantage in region and function coverage, improving region coverage by 21.87%–36.24% and function coverage by 14.63%–28.09% over the baselines.

This coverage advantage reflects TriChord's design. Because it models extension APIs as a privileged bridge into browser subsystems, TriChord steers exploration toward browser-process code that is both reachable and security-critical. In addition, coordinated multi-extension execution and user-interaction simulation allow TriChord to trigger deeper browser states than generic interface fuzzers can typically reach.

Coverage Intersection. Aggregate coverage alone does not establish whether TriChord explores code that prior fuzzers already reach, or a distinct portion of the browser process. We therefore treat each tool's covered lines as a set of stable `path:line` identifiers and report, for each baseline B relative to TriChord's set T, the fraction of the baseline's coverage subsumed by TriChord ($|T \cap B|/|B|$) and the coverage unique to each tool ($|T \setminus B|$, $|B \setminus T|$). Table 4 summarizes the results on the shared browser-process scope, excluding extension-handler code: TriChord covers 71,907 lines, i.e., 119.4% of the combined baseline coverage (60,222 lines), subsuming 89.7% of the baselines' coverage overall and 95.7%, 96.7%, and 90.6% of each baseline individually, while adding 17,902 lines the baselines do not reach. The baselines are not simply underpowered tools: on the renderer-only scope (`third_party/blink`), TriChord reaches 80.8% of the baselines' joint coverage, which includes 98,707 lines TriChord does not reach.

In the extension-handler scope (`src/chrome/browser/extensions` and `src/extensions`), TriChord covers 210.2% of the baseline union. In addition, TriChord covers 13,025 unique lines, compared with 114 unique lines covered by all baselines combined. The Chromium tree contains a single fuzz target in `src/chrome/browser/extensions` and none in `src/extensions`, indicating that this surface was previously unexplored; TriChord reaches 6,053 lines of extension-handler code that none of the baselines exercises. We therefore characterize TriChord as subsuming the baselines' coverage on the shared browser-process scope while additionally reaching previously unexplored extension-handler code. This claim rests on the subsumption-and-complementarity relationship described above rather than on aggregate coverage alone.

Bug Detection. Table 3 reports crash counts and deduplicated security issues from a 24-hour run. For each fuzzer, crashes are grouped by manually verified root cause, with memory errors identified using AddressSanitizer. TriChord substantially outperforms the baselines in overall bug discovery, showing stronger and more consistent results across the evaluated versions.

The contrast is especially clear in Version 2, where TriChord triggers 2,919 crashes corresponding to 9 distinct security issues. Although FreeDom discovers one additional issue in Version 4 (3 versus 2), TriChord remains clearly superior overall, both in total issues found and in crash-triggering capability across versions.

The discovered bug types also differ markedly in severity. The three baselines together uncover 8 security issues, all low severity, such as null dereferences. In contrast, 4 out of 9 issues found by TriChord are memory-corruption vulnerabilities, including use-after-free and container-overflow bugs with potential implications for arbitrary code execution or privilege escalation.

5.3 RQ2: Effectiveness of Major Components

To quantify the contribution of TriChord's major design components, we perform an ablation study using the three reduced variants introduced in §5.1. Figure 6 and Figure 7 show the coverage and bug discovery results on Chromium version 2, respectively. The full version of TriChord, which combines API-subsystem mapping, coordinated multi-extension orchestration, and user interaction simulation, achieves 16.40% region coverage, 24.16% function coverage, 19.07% line coverage, and 9 security issues.

The first variant, TriChord$_1$, emulates conventional API fuzzing by executing random API calls without subsystem mapping or orchestration. It reaches only 14.30% region coverage, 21.54% function coverage, 16.62% line coverage, and 2 security issues. This result shows that extension API support alone is insufficient: without subsystem-aware targeting, the fuzzer explores browser behavior less effectively and misses vulnerabilities hidden behind more complex execution states.

The second variant, TriChord$_2$, disables multi-extension orchestration. Removing this component causes region, function, and line coverage to drop by 10.8%, 7.4%, and 9.6%, respectively, while security issues fall from 9 to 1. This sharp decline highlights the importance of extension coordination for exposing bugs that depend on inter-extension interactions, ordering effects, or shared-resource racing conditions.

The third variant, TriChord$_3$, eliminates user interaction simulation. This variant suffers the largest coverage degradation, with reductions of 17.1% in region coverage, 13.4% in function coverage,

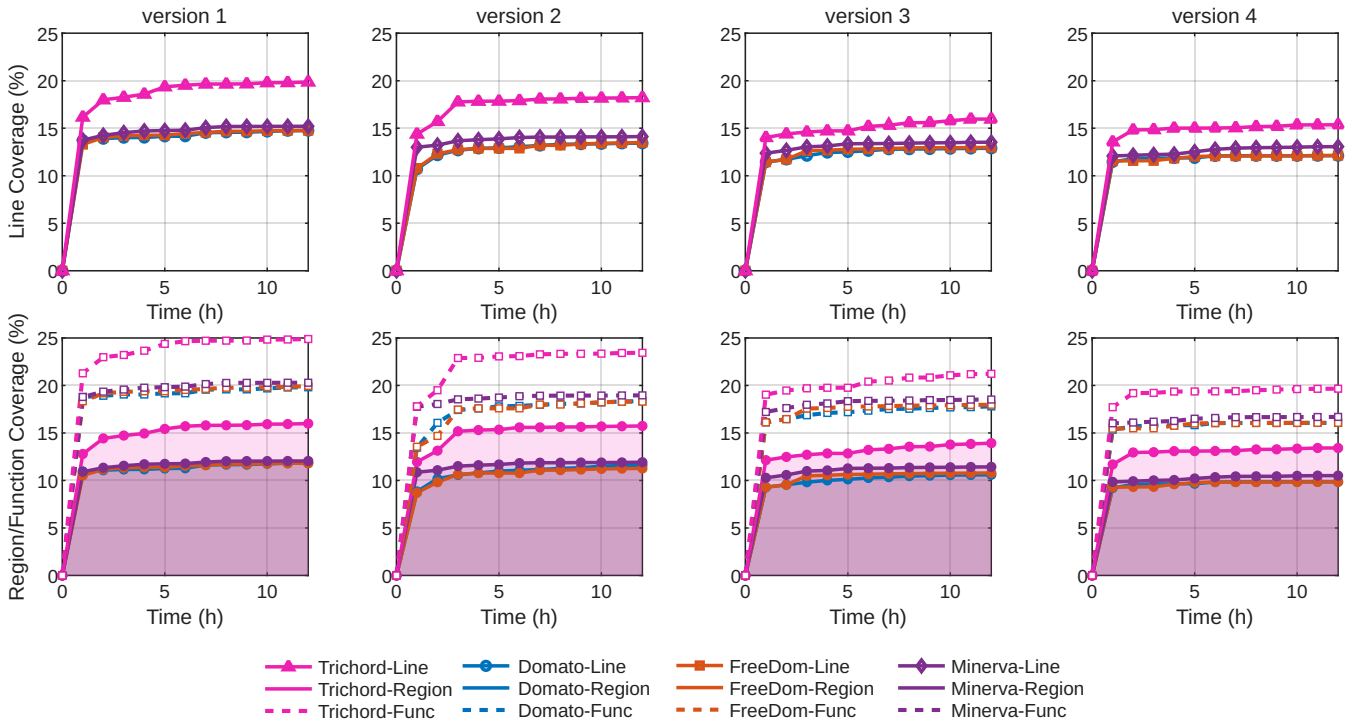

Figure 5: Code coverage (including line, region, and function coverage) comparison between TriChord and baseline fuzzers across four Chromium versions during a 12-hour fuzzing campaign.

Table 3: Number of security issues (OOM and assertion failure bugs are excluded) detected by TriChord and counterparts within 24-hour runs. Issues are manually verified and deduplicated by root cause analysis.

Method	Version 1 [13]		Version 2 [14]		Version 3 [15]		Version 4 [17]		**Sum**
	# Crash	# Issues	# Crash	# Issues	# Crash	# Issues	# Crash	# Issues	# Issues
Domato [19]	3	1	1	1	0	0	0	0	2
FreeDom [45]	5	1	20	2	0	0	6	3	6
Minerva [58]	0	0	0	0	0	0	0	0	0
TriChord (ours)	61	3	2,919*	9	16	1	8	2	15

*The crashes in Version 2, despite their exceptionally high number (2,919), are largely duplicates and can be quickly grouped into 9 distinct security issues based on their 9 unique ASan backtrace logs. Among them, 2,807 crashes stem from four invalid-memory-access issues in *extension::AppHooksDelegate* and *extension::ScriptContext*, while 98 crashes are caused by a container-overflow issue in *TabsHooksDelegate*. The remaining 14 crashes belong to 4 other distinct bugs.

Table 4: Coverage intersection between TriChord and baseline fuzzers on the shared browser-process scope (`src/chrome/browser` minus extension).

Baseline	$\lvert T \cap B\rvert \,/\, \lvert B\rvert$	$\lvert T \setminus B\rvert$	$\lvert B \setminus T\rvert$
Domato	95.7%	18,992	2,394
FreeDom	96.7%	18,791	1,805
Minerva	90.6%	18,222	5,541

T: lines covered by TriChord (71,907); B: lines covered by the baseline; three baselines' union U covers 60,222 lines.

and 16.0% in line coverage, and it discovers only one security issue. These results demonstrate that realistic user-driven events are essential for exercising event handlers, callback mechanisms, and UI-dependent state transitions that remain dormant under API-only execution. It is especially important given browsers' heavy reliance on event-driven programming.

Overall, despite the high crash count (2,919) in the full system being dominated by repeated triggering of a few root causes, TriChord still identifies 4.5×, 9×, and 9× more security issues than TriChord$_1$, TriChord$_2$, and TriChord$_3$, respectively. Moreover, 8 security issues, including 5 null dereferences, 1 container-overflow, and 2 heap-use-after-free bugs, are found only by the full system. No reduced variant is able to recover this set of findings on its own.

Taken together, these findings demonstrate that TriChord's three components are complementary rather than interchangeable.

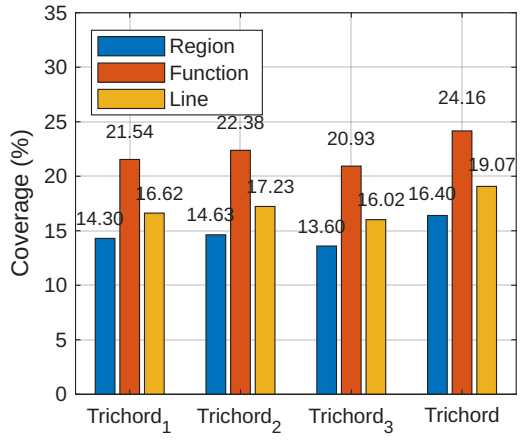

Figure 6: Coverage results of ablation variants over version 2 for a 24-hour run.

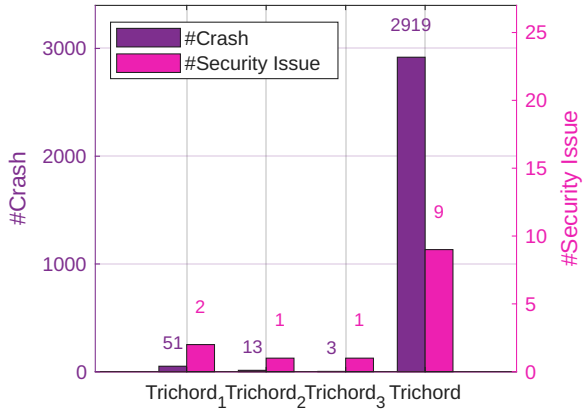

Figure 7: Bug discovery results of ablation variants over version 2 for a 24-hour run.

API-subsystem mapping directs exploration toward relevant privileged functionality, multi-extension orchestration creates realistic interaction states, and user interaction simulation activates event-driven execution paths. Their combination is what enables TriChord to probe deeply into browser subsystems and uncover vulnerabilities that simpler variants consistently miss.

5.4 RQ3: Discovery of 0-Day Vulnerabilities

To evaluate TriChord's ability to discover previously unknown vulnerabilities, we conducted long-running fuzzing campaigns across more than 20 Chromium versions released between 2020 and 2025. Table 5 summarizes the 28 previously unknown bugs uncovered by TriChord, all of which we manually verified before reporting. These findings span multiple browser components and extension interfaces and cover diverse memory-safety failures, including heap-use-after-free, heap-buffer-overflow, use-after-poison, null dereference, and container-overflow. The final impact assessment and disclosure status were determined solely by the Chrome security team. Among the 28 reported bugs, 10 were confirmed by Google[2], 8 were assigned CVE identifiers, and the disclosures yielded $73,000 in bug bounty rewards. Five of the confirmed bugs were classified by Google as high-impact vulnerabilities and later received CVSS 3.x [36] scores of 9.6, 8.8, 8.8, 8.8, and 8.8 from the National Vulnerability Database [37].

Importance of Orchestration. The "Orch." column in Table 5 indicates whether each bug's discovery depends on multi-extension coordination (M), user interaction simulation (U), or both (M+U), as described in §3.5. Of the 28 bugs, 19 require at least one orchestration capability to trigger: 5 depend on multi-extension coordination alone, 3 on user interaction simulation alone, and 11 on both. A clear temporal trend emerges: bugs found in early Chromium versions (83–90) are predominantly triggered by basic extension fuzzing or user interaction simulation alone, whereas bugs in later versions (93+, 114+, 137+, 142+) increasingly require multi-extension coordination or the combination of both mechanisms. This trend suggests that as Chromium's extension infrastructure matures and simpler vulnerabilities are patched, the remaining bugs reside in deeper interaction states that can only be reached through coordinated multi-extension execution and realistic user-driven events. These results reinforce the ablation findings in §5.3 and confirm that orchestrating the fuzzing process becomes increasingly essential for exposing security bugs in modern browser extension implementations.

Covering Both Mature and Recent Chromium Versions. An important observation is that TriChord's effectiveness persists across both older and more recent Chromium releases. Despite substantial changes in Chrome's extension ecosystem and security mechanisms over time, TriChord continues to uncover new vulnerabilities rather than merely reproducing bugs from legacy versions. In particular, 3 of the 10 confirmed bugs were found in 2025-era Chromium builds, specifically versions 137.0.7122.0 and 142.0.7391.0, as shown in Table 5. Moreover, the vulnerability listed as ID-1 in Table 5 was the *first significant* extension API vulnerability identified in this line of work, earning an additional bounty from Google's security team. Together, these results show that extension API implementations remain a meaningful source of security risk even in mature and continuously hardened browser versions.

We next present two case studies that illustrate how TriChord exposes vulnerabilities that are difficult to trigger with conventional browser fuzzing techniques.

Case Study 1: (ID 1 in Table 5) is a high-severity heap-use-after-free vulnerability (CVSS 3.x score 9.6, bounty $20,000) in Chromium's `TabStripModel::FixOpeners` function, responsible for maintaining opener relationships between browser tabs. Figure 8 shows the proof-of-concept (PoC) code generated by TriChord that triggers this vulnerability. The issue arises when a sequence of tab operations creates inconsistent opener references, eventually causing the browser to dereference freed objects. Although extensions cannot directly manipulate the underlying opener references maintained by the browser, they can indirectly corrupt this state through a carefully chosen sequence of `chrome.tabs.create()`, `chrome`

[2]The remaining issues had limited impact (e.g., null dereference), were duplicates of other reports, or had already been fixed internally; see Table 5.

Table 5: Previously unknown security bugs detected by TriChord and verified by us. The final impact assessment and status were determined solely by the Chrome security team based on their bug bounty rules ("None" indicates no bounty).

ID	Version	Bug Type	Bug Location	Impact	Orch.†	Status	Bounty (USD)
1	83.0.4103.116	heap-use-after-free	extensions::GetTabIdForExtensions	High	–	**Confirmed**	20,000
2	84.0.4147.125	heap-use-after-free	task_manager::WorkerTaskProvider	Medium	U	**Confirmed**	5,000
3	84.0.4147.89	heap-use-after-free	TabStrip::CloseTab	High	U	Duplicate	None
4	85.0.4151.0	null dereference	ExtensionAction::ParseIconFromCanvasDictionary	Crash	M	None	None
5	85.0.4151.0	null dereference	extensions::ProcessesGetProcessInfoFunction	Crash	–	None	None
6	86.0.4203.0	heap-use-after-free	cc::ThroughputUkmReporter::ReportThroughputUkm	Bug	–	Duplicate	None
7	88.0.4324.146	heap-buffer-overflow	TabStripModel::MoveWebContentsAtImpl	High	–	**Confirmed**	10,000
8	88.0.4324.146	null dereference	task_manager::PerProfileWorkerTaskTracker	Crash	–	None	None
9	88.0.4324.146	container-overflow	gfx::JPEGCodec::Decode	Crash	M	None	None
10	88.0.4324.146	null dereference	extensions::WebrtcLoggingPrivateFunction::RphFromRequest	Crash	–	None	None
11	90.0.4430.72	use-after-poison	NetworkStateNotifier::NotifyObservers	Medium	U	**Confirmed**	5,000
12	93.0.4577.63	heap-buffer-overflow	TabStripModel::IsTabBlocked	Medium	M	**Confirmed**	10,000
13	93.0.4544.0	null dereference	extensions::AppHooksDelegate::GetDetails	Crash	M+U	None	None
14	93.0.4544.0	null dereference	extensions::AppHooksDelegate::GetIsInstalled	Crash	M+U	None	None
15	93.0.4544.0	null dereference	extensions::AppHooksDelegate::GetRunningState	Crash	M+U	None	None
16	93.0.4544.0	null dereference	extensions::ScriptContext::GetRenderFrame	Crash	M+U	None	None
17	93.0.4544.0	null dereference	blink::SVGImage::RootElement	Crash	–	None	None
18	93.0.4544.0	container-overflow	extensions::TabsHooksDelegate::HandleSendMessage	Crash	M+U	None	None
19	93.0.4544.0	heap-use-after-free	extensions::ResourceBundleSourceMap::GetSource	Crash	M+U	Fixed	None
20	93.0.4544.0	heap-use-after-free	Cr_z_inflate	Crash	M+U	Fixed	None
21	93.0.4544.0	null dereference	GetBindingsSystem	Crash	M+U	None	None
22	92.0.4515.107	heap-use-after-free	task_manager::TaskManagerImpl::Refresh	High	–	**Confirmed**	15,000
23	95.0.4638.32	use-after-poison	blink::WorkerMainScriptLoader	High	M	**Confirmed**	3,000
24	114.0.5679.1	null dereference	blink::LayoutObject::IsDescendantOf	Crash	M+U	None	None
25	137.0.7122.0	heap-use-after-free	ExtensionService	Low	M+U	**Confirmed**	None
26	137.0.7122.0	dangling-pointer-instantiation	extensions::UnpackedInstaller	Crash	M+U	Fixed	None
27	142.0.7391.0	heap-buffer-overflow	viz::SoftwareRenderer	Medium	M	**Confirmed**	3,000
28	142.0.7391.0	use-after-poison	extensions::UnloadExtension	High	–	**Confirmed**	2,000

† Orchestration dependency: M = Multi-Extension Coordination, U = User Interaction Simulation, M+U = Both, – = Neither (basic extension alone).

```
async function main(){
  chrome.tabs.getCurrent(function(var_tab_1){
    chrome.tabs.create({},function(var_tab_2){
      chrome.tabs.update(
        var_tab_1.id,
        {openerTabId:var_tab_2.id})
      chrome.tabs.update(
        var_tab_2.id,
        {openerTabId:var_tab_1.id})
      chrome.tabs.move(
        var_tab_1.id,
        {index:2})
      chrome.tabs.discard(
        var_tab_2.id)
    })
  })
}
main();
```

Figure 8: Case 1's PoC generated by TriChord.

```
async function main(){
  await chrome.processes.getProcessInfo(
    [],true,function(processes){})
}
main();
```

(a) The JS PoC code.

```
#!/bin/bash
xdotool search --name "chrome" windowactivate &
xdotool mousemove <close_button_x> <close_button_y>
xdotool click 1
```

(b) The shell PoC code.

Figure 9: Case 2's PoC discovered by TriChord.

`.tabs.update()`, `chrome.tabs.move()`, and `chrome.tabs.discard()` operations.

Because the vulnerable state depends on privileged browser managed tab relationships, neither standard Web APIs nor naive manual testing can directly steer execution toward this condition. This case therefore highlights TriChord's ability to reach privileged browser-process states through carefully orchestrated extension API sequences.

Case Study 2: (ID 2 in Table 5) is a heap-use-after-free vulnerability in `task_manager::WorkerTaskProvider`. Triggering this vulnerability requires a precise cross-phase execution sequence: the JavaScript PoC first invokes the relevant API calls, after which a shell script triggers browser shutdown to expose accesses to already destroyed objects. Figure 9 illustrates this two-part PoC. The JavaScript component (Figure 9(a)) calls `chrome.processes.getProcessInfo()` with empty filter criteria and a callback, initiating a query to the browser's task management subsystem. The shell script (Figure 9(b)) uses `xdotool` to simulate user interaction and programmatically trigger browser shutdown by activating the Chrome window and clicking the close button.

The root cause of this use-after-free is a lifecycle ordering bug: `WorkerTaskProvider` (part of `TaskManager`) outlives the `ProfileManager`, so during browser shutdown the `ProfileManager` is destroyed first and `WorkerTaskProvider` is destroyed later. Its

destructor then attempts to unregister a `ScopedObserver` from an already-destroyed `ProfileManager`, resulting in a use-after-free.

This case demonstrates the value of TriChord's user-interaction simulation, which allows API-driven execution to be combined with browser lifecycle events such as shutdown. Conventional fuzzers typically emphasize API invocation in isolation and therefore miss state transitions triggered by user actions or teardown logic. More broadly, the case highlights why realistic user interaction and lifecycle-aware orchestration are essential for specialized security testing of privileged extension APIs.

6 Cross-Browser Evaluation and Discussion

Having demonstrated TriChord's effectiveness on Chromium, we next examine its cross-browser generality and broader practical implications. We first report the completed Firefox port and its evaluation results, then distill lessons for browser vendors, and finally discuss the framework's limitations and future directions.

6.1 Firefox Port and Evaluation

Completed Firefox Port. We completed a Firefox implementation of the full fuzzing pipeline used in our Chromium study: subsystem-oriented API generation, coordinated execution of 2–10 extensions, user-interaction simulation, browser-instance management, and crash collection. The browser-specific changes comprise four parts. First, we adapted the generated calls from the `chrome.*` namespace and callback-based idioms to Firefox's `browser.*` namespace and Promise-based APIs. Second, we constructed Firefox-compatible template and manifest. Third, because Firefox has no single-flag equivalent of Chromium's `--load-extension`, we implemented multi-extension loading by installing each generated extension into a dedicated Firefox profile. Finally, we regenerated the API grammar from Firefox's shipped schemas, removed Chrome-only interfaces, and added Firefox-specific interfaces. These changes preserve all runtime components of TriChord's Chromium pipeline.

Evaluation Results. We evaluated Mozilla's ASan-enabled nightly builds for five Firefox versions (87.0a1, 92.0a1, 137.0a1, 141.0a1, and 155.0a1), spanning five years of development. Using the same orchestration strategy as in our Chromium experiments, we fuzzed each version for 48 hours, for a total of 240 hours. TriChord produced nine distinct failure signatures (Table 6): one heap-use-after-free and one out-of-bounds read reported by ASan, five null dereference crashes, one assertion failure, and one `MOZ_CRASH` failure.

Cross-Browser Design Implications. The Firefox port yields two implications. ① TriChord's high-level testing strategy transfers across WebExtensions implementations: subsystem-oriented generation, multi-extension coordination, and user simulation required no conceptual redesign. The concrete API schemas, templates, asynchronous calling conventions, extension-loading mechanisms, and UI targets nevertheless remain browser-specific configuration artifacts. ② A shared WebExtensions API does not imply identical native-code reachability. The Firefox failure stacks show extension calls crossing its privileged JavaScript dispatch layer (`toolkit/components/extensions/ext-*.js`) into native networking, graphics, DOM, and document-loading components. The UAF and out-of-bounds findings demonstrate that this JavaScript layer does not eliminate native memory-safety exposure, while the different failure locations show that an equivalent API surface can exercise different implementation boundaries. The resulting design lesson is that cross-browser extension fuzzers can reuse orchestration logic, but must derive browser-specific API models and track which native subsystems each implementation actually reaches.

6.2 Lessons for Browser Vendors

Our results yield two concrete lessons for browser vendors.

Harden lifecycle management across asynchronous extension events. Extension API calls often complete asynchronously. While a callback is pending, the extension may be uninstalled, its worker terminated, its tab or window closed, or the browser shut down, leaving the handler to dereference a stale object or execute in a destroyed context. This pattern underlies IDs 1, 2, and 28 in Table 5. Our Firefox evaluation (§6.1) exhibits the same class of failure under a different architecture, where pending Promise runnables are destroyed during shutdown, suggesting a broader design concern across browser architectures.

Test coordinated state transitions, not only individual APIs. Most findings require more than a single invocation of one API: 19 of the 28 bugs we found depend on multi-extension coordination, user interaction, or both. Vendors should incorporate minimized PoCs into regression tests that exercise these coordinated states—e.g., concurrent extensions, install/unload/close interleavings, and user-initiated events—rather than test each handler in isolation.

6.3 Limitations and Future Directions

Supporting Evolving Subsystems. Browser extension ecosystems continue to change, and any practical testing framework must evolve with them. To support this, the extension API generation rules we developed (building on Domato) are publicly available and designed for extension as new APIs and subsystems emerge. Our template-based workflow is decoupled from API generation, making it easier to update execution contexts, permissions, and invocation patterns without redesigning the full framework. More broadly, TriChord's modular architecture allows individual components to be refined or replaced as browser internals and extension capabilities evolve, helping preserve long-term testing relevance.

Detecting More Complex Vulnerabilities. TriChord currently focuses on vulnerabilities that can be triggered through interactions among 2 to 10 extensions within a 30-second observation window. This scope is intentional. First, it prioritizes vulnerabilities with relatively low attack complexity and high temporal reliability, i.e., issues that present the most immediate threats. Second, increasing the number of concurrent extensions or substantially extending the observation window would raise orchestration complexity, computational cost, and overall fuzzing time.

Even so, this design does not bound the broader research space. Future work could explore larger groups of interacting extensions, longer-lived event chains, and vulnerabilities that emerge only after delayed cleanup, shutdown, or rare background execution conditions. Such extensions would complement the current system by targeting deeper and less frequent failure modes that lie beyond the present efficiency-oriented configuration.

Table 6: Distinct failure signatures observed by TriChord in ASan-enabled Firefox builds.

ID	Version	Failure Type	Location
F1	87.0a1	heap-use-after-free	`nsTArray::IsAutoArray` via `AutoIPCStream`
F2	87.0a1	null dereference	XPCOM shutdown
F3	87.0a1	null dereference	`NS_CycleCollectorSuspect3`
F4	87.0a1	null dereference	`XRE_InitChildProcess`
F5	87.0a1	null dereference	`js::ContextChecks::check`
F6	92.0a1	out-of-bounds read	`mozilla::gl::GLContext::MakeCurrent`
F7	137.0a1	assertion failure	`OpenWindowInternal`
F8	141.0a1	null dereference	`DocumentLoadListener::Open`
F9	155.0a1	`MOZ_CRASH`	`CustomElementRegistry::RunCustomElementCreationCallback`

7 Related Work

Research most closely related to our work comes from two directions: browser extension security and browser fuzzing. The former primarily studies malicious or vulnerable extensions themselves, while the latter focuses on testing web-facing browser components. Neither line of work directly addresses specialized security testing of the browser-side implementation of privileged extension APIs.

7.1 Browser Extension Security

Prior work on browser extension security mainly falls into three categories. First, a large body of work studies malicious, abusive, or privacy-invasive extensions. Representative examples include large-scale analyses of malicious extensions in the Chrome Web Store [26], update-based threat detection [39], Hidden Markov Model-based detection of malicious or vulnerable behavior [42], security-header tampering by extensions [2], and more recent learning-based detection and mitigation techniques [41, 61]. Related work also highlights major privacy risks, including automated content extraction [53], inconsistencies between declared and actual data practices [6], and browser-extension fingerprinting attacks [3, 28, 43, 44]. These studies are important for understanding extension behavior in the wild, but they primarily analyze what extensions do, rather than how the browser implements the privileged APIs that extensions invoke.

Second, prior work has examined architectural weaknesses in extension systems and vulnerabilities inside extensions themselves. Carlini et al. [7] provided an early analysis of Chrome's extension architecture, revealing weaknesses in the permission model and privilege separation. Kim and Lee [29] found dozens of extension vulnerabilities enabling privilege escalation, UXSS, and credential theft, while Yu et al. [56] proposed CoCo for extension vulnerability detection via coverage-guided concurrent abstract interpretation. Again, the main focus of these efforts is extension logic, permissions, and extension-side attack surfaces, rather than flaws in the browser-side implementation of privileged extension APIs.

Third, work that comes closest to API-level security remains limited. Moreno et al. [35] studied abuse of browser-facing functionality through the Chrome DevTools Protocol [23], but their goal was to analyze API misuse in developer-oriented interfaces rather than implementation vulnerabilities in extension API handlers inside the privileged browser process. As a result, the security of extension-to-browser interactions through privileged extension APIs remains largely unexplored.

7.2 Browser Fuzzing Techniques

Browser fuzzing has made substantial progress on web APIs and other browser components. For general browser API fuzzing, Domato [19], FreeDom [55], and Minerva [60] demonstrate the effectiveness of structured generation, full-stack browser exploration, and targeted browser-API analysis. Tacoma [51] further increases the fuzzing coverage of complex browser functionality through fine-grained semantic alignment.

Other work has specialized in individual browser subsystems. For JavaScript engines, Fuzzilli [25] uses grammar-aware program generation, DUMPLING [50] enables differential testing through fine-grained transformations, Favocado [18] targets the binding layer between JavaScript engines and native components, and Xu et al. [54] propose a graph-based intermediate representation for coverage-guided exploration. For WebAssembly and graphics-related components, Waltzz [59], RGFuzz [40], and DarthShader [4] show the benefits of domain-specific testing for specialized runtimes and shader compilers.

The distinction relevant to our setting is the browser state being constructed. The evaluated browser fuzzers generate web- or renderer-facing inputs, and binding-layer fuzzers such as Favocado invoke native interfaces from JavaScript contexts, but they do not install and coordinate multiple isolated extensions together with browser-UI events to mutate shared state in privileged subsystems. Reaching that state requires an extension-oriented workflow that combines subsystem-related calls across namespaces and actors.

Existing Chromium extension tests [47] exercise some of this functionality for correctness and regression prevention, but are not vulnerability-oriented fuzzing systems. TriChord addresses the resulting gap through subsystem-oriented state construction and coordinated extension–browser orchestration; the ablation results further show that removing multi-extension coordination or user interaction reduces the nine discovered issues to two (§5.3).

8 Conclusion

This paper identifies and systematically investigates an under-explored browser attack surface: the browser-side implementation of privileged extension APIs. To test this interface, we presented TriChord,

which combines subsystem-guided state construction with multi-extension and user-event orchestration to exercise privileged browser states that the evaluated browser fuzzers do not instantiate.

Across multiple Chromium versions, TriChord subsumed 89.7% of the baselines' combined coverage on shared browser-process code, reached 6,053 extension-handler lines absent from all baselines, and demonstrated consistently stronger vulnerability discovery performance. The ablation results further confirmed that coordinated extensions and user interaction are important for reaching the vulnerability-triggering states exposed by the full system. In real-world security testing, TriChord uncovered 28 previously unknown bugs, 10 of which were confirmed by Google, leading to 8 CVE assignments and $73,000 in bug bounty rewards. These findings include high-severity vulnerabilities with the potential to enable privilege escalation or contribute to sandbox escape.

Acknowledgements

We thank all the reviewers for their constructive comments, and we are particularly grateful to the reviewer who provided valuable guidance during the minor revision process. Daoyuan Wu's role in this research was partially supported by SDF Academic Research Award Grant (Grant No.: 631463), Lingnan University's Direct Grant (DR26F6), Innovation and Impact Fund (KT26A3), and Faculty Research Grants (SDS25A13, SDS25A21). We used OpenAI GPT-5.4 and 5.6 to fix English throughout the paper. All responses were then manually checked to ensure accuracy.

References

[1] [n. d.]. Chromium Source Repository: src/chrome/browser. https://source.chromium.org/chromium/chromium/src/+/main:chrome/browser/.
[2] Shubham Agarwal. 2022. Helping or hindering? how browser extensions undermine security. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 23–37. https://dl.acm.org/doi/10.1145/3548606.3560685.
[3] Shubham Agarwal, Aurore Fass, and Ben Stock. 2024. Peeking through the window: fingerprinting browser extensions through page-visible execution traces and interactions. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 2117–2131. https://dl.acm.org/doi/10.1145/3658644.3670339.
[4] Lukas Bernhard, Nico Schiller, Moritz Schloegel, Nils Bars, and Thorsten Holz. 2024. DarthShader: Fuzzing WebGPU shader translators & compilers. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 690–704. https://arxiv.org/abs/2409.01824.
[5] Chromium Blog. 2008. Multi-process Architecture. https://blog.chromium.org/2008/09/multi-process-architecture.html.
[6] Duc Bui, Brian Tang, and Kang G Shin. 2023. Detection of inconsistencies in privacy practices of browser extensions. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2780–2798. https://ieeexplore.ieee.org/document/10179338.
[7] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. 2012. An evaluation of the google chrome extension security architecture. In *21st USENIX Security Symposium (USENIX Security 12)*. 97–111. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/carlini.
[8] Chrome Developers. 2024. API reference | Chrome for Developers. https://developer.chrome.com/docs/extensions/reference/api.
[9] Chrome Developers. 2024. Chrome Extensions Documentation. https://developer.chrome.com/docs/extensions.
[10] Chrome-Stats. 2025. Chrome Extension Statistics (Aug 2025). https://chrome-stats.com/chrome/stats.
[11] Chromium. 2024. Chromium Docs: Sandbox. https://chromium.googlesource.com/chromium/src/+/HEAD/docs/design/sandbox.md.
[12] Chromium Project. [n. d.]. Chromium Browser vs Google Chrome. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/chromium_browser_vs_google_chrome.md.
[13] Chromium Project. 2020. Chromium Source Code (85.0.4151.0). https://chromium.googlesource.com/chromium/src/+/refs/tags/85.0.4151.0.
[14] Chromium Project. 2021. Chromium Source Code (93.0.4544.0). https://chromium.googlesource.com/chromium/src/+/refs/tags/93.0.4544.0.
[15] Chromium Project. 2023. Chromium Source Code (114.0.5720.2). https://chromium.googlesource.com/chromium/src/+/refs/tags/114.0.5720.2.
[16] Chromium Project. 2024. crx3.proto. https://source.chromium.org/chromium/chromium/src/+/main:components/crx_file/crx3.proto.
[17] Chromium Project. 2025. Chromium Source Code (137.0.7118.3). https://chromium.googlesource.com/chromium/src/+/refs/tags/137.0.7118.3.
[18] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. 2021. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases.. In *NDSS*. https://adamdoupe.com/publications/favocado-ndss21.pdf.
[19] Ivan Fratric. 2017. Domato: A DOM Fuzzer. GitHub repository. https://github.com/googleprojectzero/domato.
[20] Google. 2024. Chrome Web Store - Extensions. https://chromewebstore.google.com/category/extensions.
[21] Google Chrome Developers. 2023. Chrome Extensions Manifest V2 Documentation. Chrome Developer Documentation. https://developer.chrome.com/docs/extensions/mv2.
[22] Google Chrome Developers. 2023. Migrating to Manifest V3. Chrome Developer Documentation. https://developer.chrome.com/docs/extensions/develop/migrate/what-is-mv3.
[23] Google Chrome Developers. 2024. Chrome DevTools | Chrome for Developers. https://developer.chrome.com/docs/devtools.
[24] Google Chrome Developers. 2024. Manifest file format | Chrome for Developers. https://developer.chrome.com/docs/extensions/reference/manifest.
[25] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. 2023. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities.. In *NDSS*. https://www.ndss-symposium.org/wp-content/uploads/2023-290-paper.pdf.
[26] Nav Jagpal, Eric Dingle, Jean-Philippe Gravel, Panayiotis Mavrommatis, Niels Provos, Moheeb Abu Rajab, and Kurt Thomas. 2015. Trends and lessons from three years fighting malicious extensions. In *24th USENIX Security Symposium (USENIX Security 15)*. 579–593. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/jagpal.
[27] Jordan Sissel. [n. d.]. xdotool. https://github.com/jordansissel/xdotool.
[28] 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 *31st USENIX Security Symposium (USENIX Security 22)*. 735–752. https://www.usenix.org/conference/usenixsecurity22/presentation/karami.
[29] Young Min Kim and Byoungyoung Lee. 2023. Extending a hand to attackers: browser privilege escalation attacks via extensions. In *32nd USENIX Security Symposium (USENIX Security 23)*. 7055–7071. https://www.usenix.org/conference/usenixsecurity23/presentation/kim-young-min.
[30] LLVM Project. 2025. AddressSanitizer: A Fast Memory Error Detector. LLVM Documentation. https://clang.llvm.org/docs/AddressSanitizer.html.
[31] LLVM Project. 2025. Clang Coverage: Source-based Code Coverage Tool. LLVM Documentation. https://clang.llvm.org/docs/SourceBasedCodeCoverage.html.
[32] Mozilla Developer Network (MDN). 2024. Browser Extensions - Mozilla | MDN. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions.
[33] MDN Web Docs. 2024. *Web APIs | MDN*. https://developer.mozilla.org/en-US/docs/Web/API.
[34] Microsoft. 2024. Microsoft Edge Addons. https://microsoftedge.microsoft.com/addons/Microsoft-Edge-Extensions-Home.
[35] José Miguel Moreno, Narseo Vallina-Rodriguez, and Juan Tapiador. 2023. Chrowned by an extension: abusing the Chrome DevTools protocol through the debugger API. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 832–846. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10190532.
[36] National Institute of Standards and Technology. 2024. CVSS v3 Calculator. https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator.
[37] National Institute of Standards and Technology. 2024. National Vulnerability Database. https://nvd.nist.gov/.
[38] Scott Orgera. 2022. How to Use the Google Chrome Task Manager. https://www.lifewire.com/google-chrome-task-manager-4103619.
[39] Nikolaos Pantelaios, Nick Nikiforakis, and Alexandros Kapravelos. 2020. You've changed: Detecting malicious browser extensions through their update deltas. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 477–491. https://doi.org/10.1145/3372297.3423343.
[40] Junyoung Park, Yunho Kim, and Insu Yun. 2025. RGFuzz: Rule-Guided Fuzzer for WebAssembly Runtimes. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 920–938. https://www.computer.org/csdl/proceedings-article/sp/2025/223600a003/21B7PWv1JGU.
[41] Joel Sam and J Ancy Jenifer. 2023. Mitigating the Security Risks of Browser Extensions. In *2023 International Conference on Sustainable Computing and Smart Systems (ICSCSS)*. IEEE, 1460–1465. https://ieeexplore.ieee.org/abstract/document/10169483/.

[42] Hossain Shahriar, Komminist Weldemariam, Mohammad Zulkernine, and Thibaud Lutellier. 2014. Effective detection of vulnerable and malicious browser extensions. *Computers & Security* 47 (2014), 66–84. https://www.sciencedirect.com/science/article/pii/S0167404814000984.
[43] Konstantinos Solomos, Panagiotis Ilia, Soroush Karami, Nick Nikiforakis, and Jason Polakis. 2022. The dangers of human touch: fingerprinting browser extensions through user actions. In *31st USENIX Security Symposium (USENIX Security 22)*. 717–733. https://www.usenix.org/conference/usenixsecurity22/presentation/solomos.
[44] 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*. 2675–2688. https://dl.acm.org/doi/10.1145/3548606.3560576.
[45] Georgia Tech SSLab. 2020. FREEDOM: Engineering a State-of-the-Art DOM Fuzzer. GitHub repository. https://github.com/sslab-gatech/freedom.
[46] StatCounter Global Stats. 2025. Browser Market Share Worldwide. https://gs.statcounter.com/browser-market-share/.
[47] Chromium Team. 2023. Extension Tests Documentation. https://source.chromium.org/chromium/chromium/src/+/main:extensions/docs/extension_tests.md.
[48] Unknown Author. 2023. CRX3 Design Doc. https://docs.google.com/document/d/e/2PACX-1vRAnTtMX8tY-KN4EhwskmfMIHQgWX1buU5S8AJesILAXG8Re50e5IiBRyozaJc2vFDavRVvjwziScAA/pub.
[49] W3C WebExtensions Community Group. [n. d.]. WebExtensions Community Group. https://www.w3.org/community/webextensions/.
[50] Liam Wachter, Julian Gremminger, Christian Wressnegger, Mathias Payer, and Flavio Toffalini. 2025. DUMPLING: Fine-grained Differential JavaScript Engine Fuzzing. In *NDSS*. https://www.ndss-symposium.org/ndss-paper/dumpling-fine-grained-differential-javascript-engine-fuzzing/.
[51] Jiashui Wang, Peng Qian, Xilin Huang, Xinlei Ying, Yan Chen, Shouling Ji, Jianhai Chen, Jundong Xie, and Long Liu. 2024. Tacoma: Enhanced Browser Fuzzing with Fine-Grained Semantic Alignment. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1174–1185. https://dl.acm.org/doi/abs/10.1145/3650212.3680351.
[52] Mozilla Extension Workshop. 2024. Developing Extensions for Firefox for Android. https://extensionworkshop.com/documentation/develop/developing-extensions-for-firefox-for-android/.
[53] Xie, Qinge et al. 2024. Arcanum: Detecting and Evaluating the Privacy Risks of Browser Extensions on Web Pages and Web Content. https://www.usenix.org/conference/usenixsecurity24/presentation/xie-qinge.
[54] Haoran Xu, Zhiyuan Jiang, Yongjun Wang, Shuhui Fan, Shenglin Xu, Peidai Xie, Shaojing Fu, and Mathias Payer. 2024. Fuzzing JavaScript Engines with a Graph-based IR. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 3734–3748. https://dl.acm.org/doi/10.1145/3658644.3690336.
[55] Wen Xu, Soyeon Park, and Taesoo Kim. 2020. Freedom: Engineering a State-of-the-Art DOM Fuzzer. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 971–986. https://dl.acm.org/doi/abs/10.1145/3372297.3423340.
[56] Jianjia Yu, Song Li, Junmin Zhu, and Yinzhi Cao. 2023. CoCo: Efficient Browser Extension Vulnerability Detection via Coverage-guided, Concurrent Abstract Interpretation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2441–2455. https://doi.org/10.1145/3576915.3616584.
[57] Matt Zeunert. 2024. Chrome Extension Statistics: Data From 2024. *DebugBear* (2024). https://www.debugbear.com/blog/chrome-extension-statistics.
[58] Chijin Zhang. 2022. Minerva: A Browser Fuzzer Augmented by API Mod-Ref Relations. GitHub repository. https://github.com/ChijinZ/Minerva.
[59] Zhang, Lingming et al. 2025. Waltzz: WebAssembly Runtime Fuzzing with Stack-Invariant Transformation. https://www.usenix.org/conference/usenixsecurity25/presentation/zhang-lingming.
[60] Chijin Zhou, Quan Zhang, Mingzhe Wang, Lihua Guo, Jie Liang, Zhe Liu, Mathias Payer, and Yu Jiang. 2022. Minerva: browser API fuzzing with dynamic mod-ref analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1135–1147. https://dl.acm.org/doi/abs/10.1145/3540250.3549107.
[61] Tiago Zonta, Mithileysh Sathiyanarayanan, et al. 2024. A Holistic Review on Detection of Malicious Browser Extensions and Links using Deep Learning. In *2024 IEEE 3rd International Conference on AI in Cybersecurity (ICAIC)*. IEEE, 1–6. https://ieeexplore.ieee.org/abstract/document/10433842/.

Open Science

To promote reproducibility and advance browser security research, we make the artifacts of our study publicly available. The TriChord source code, including extension templates, API mapping results, and generation scripts, is provided at https://github.com/AIS2Lab/Trichord to ensure that our work is accessible and reusable by the broader security research community.

Ethical Considerations

In this section, we clarify the potential ethical considerations associated with this research. When conducting this research, we took several ethical precautions to ensure that our work posed no risk to real-world systems, individuals, or intellectual property. The following paragraphs detail our approach to responsible disclosure, testing environments, potential misuse, and data collection.

Responsible disclosure: Throughout our research, we followed a careful vulnerability assessment and disclosure process. For each security issue discovered in Chromium, we first evaluated its practical impact and reproducibility. Crashes resulting only in non-exploitable conditions (such as simple null pointer dereferences) or those with inconsistent reproduction paths were excluded from disclosure. Our reporting efforts focused exclusively on issues with genuine security implications. Vulnerabilities were promptly reported to the Chrome security team through official channels, accompanied by detailed technical information to facilitate reproduction and remediation. We kept or will keep all discovered issues confidential until patches were released or until the standard 90-day disclosure window had passed.

Testing environments: All of our experiments were conducted in isolated environments that did not interact with real-world browser instances or user data. We performed our testing exclusively on local installations of Chromium, ensuring that no production environments or regular users were affected by our fuzzing activities. The bugs we discovered only occurred in our controlled testing setup.

Risks of misuse: While TriChord may be misused to discover vulnerabilities for malicious purposes, such misuse would require significant technical expertise and resources. Furthermore, the detailed methodology and implementation information we provide are intended to support security researchers and browser developers seeking to improve their testing practices. We believe that the security benefits of our research substantially outweigh the potential risks of misuse.

Data collection: Our research did not involve any human subjects or personal data. All data collected and analyzed in this study was sourced exclusively from public Chromium repositories, bug tracking systems, and our own fuzzing results. No personally identifiable information was processed at any stage of our research.

Intellectual property: All tools and software, both open-source and commercial, used in this study were utilized in full compliance with their respective licenses. We have properly cited and acknowledged all existing works that contributed to our research.