USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

BUIzz: Finding Policy Enforcement Bugs via Interaction Simulation on the Browser User Interface

Mingi Jung, Donggyu Kim, Mijung Kim, and Seongil Wi, *UNIST*

https://www.usenix.org/conference/usenixsecurity26/presentation/jung

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

BUIzz: Finding Policy Enforcement Bugs via Interaction Simulation on the Browser User Interface

Mingi Jung
UNIST

Donggyu Kim
UNIST

Mijung Kim
UNIST

Seongil Wi
UNIST

Abstract

Modern web ecosystems rely on security policy headers, such as Content Security Policy (CSP) and the SameSite cookie attribute, for client-side defenses. Because browsers enforce these headers, browser bugs in policy enforcement directly undermine these defenses. While recent studies have attempted to find such bugs, they largely overlook bugs triggered by user interactions on the browser user interface (BUI).

In this paper, we propose BUIzz, the first testing framework that identifies policy enforcement bugs triggered by BUI-level user interactions. BUIzz first collects a comprehensive set of interactions by referencing browser manuals, right-click context menus, and known browser bugs. It then executes each interaction and their combinations on the test pages via OS-level simulation. Instead of cross-browser differential testing, BUIzz leverages a pre/post-interaction oracle that checks for enforcement inconsistencies before and after applying interactions, enabling bug detection within a single browser. We demonstrate the efficacy of BUIzz by finding 35 security bugs and three functional bugs across six browsers, including Chrome and Firefox. Our reports have led to fixes for 14 security bugs, resulting in seven CVEs and $14,700 in bug bounties.

1 Introduction

Modern web applications increasingly rely on client-side security policies to prevent or mitigate web threats [96]. For example, websites use the SameSite cookie attribute [45, 72, 77] to mitigate Cross-Site Request Forgery (CSRF) attacks and Content Security Policy (CSP) [12, 64, 88, 94] to mitigate Cross-Site Scripting (XSS) attacks. Notably, Roth *et al.* [89] showed that 8,174 out of the 10,000 Tranco [85] top websites (82%) had deployed at least one of these policies.

Client-side security policies are typically specified in HTTP response headers and enforced by the browser. However, browser bugs in policy enforcement can pose critical security threats [50, 52, 53]. Even when a website developer correctly configures its headers, a browser that incorrectly enforces the policy can fail to provide the intended defense, reopening security holes. For example, a CSRF attacker may bypass the SameSite cookie attribute and induce a forged cross-site request that incorrectly includes a `SameSite=Strict` cookie, which should never be attached in a cross-site context [72].

```
1 <a href="
2   https://victim.com
3   ">
4     Visit!
5 </a>
```

(a) An HTML snippet of `https://attack.com`.

(b) A browser-supported user interaction to trigger a bug (open link in split view).

Figure 1: A SameSite cookie bypass bug in Brave; an adversary exploits the bug to include a `SameSite=Strict` cookie.

Recently, several approaches for systematically finding policy enforcement bugs have been proposed [72, 86, 102]. These methods commonly construct test documents with policies, visit them with browsers under test, and observe enforcement behavior. To avoid relying on complex and often ambiguous policy specifications, they use cross-browser behavioral differences as the bug oracle.

Unfortunately, existing approaches share a critical limitation: they do not account for enforcement bugs that require browser-supported user interactions (e.g., reopening a closed tab with Ctrl+Shift+T or clicking "Open link in new tab") to be triggered. Figure 1 illustrates a SameSite cookie enforcement bug in Brave that we discovered. In particular, when the link at Line 4 in Figure 1a is opened in split view (Figure 1b), a `SameSite=Strict` cookie for `https://victim.com` is incorrectly included in a cross-site request, enabling a CSRF attack.

This class of bugs arises when browsers fail to apply policy enforcement consistently across rendering changes induced by interactions initiated outside the document, namely through the browser user interface (BUI) [4]. As modern browsers expose a rich set of BUI-level interactions that frequently alter rendering state, enforcement inconsistencies can commonly happen in practice (as shown in §C). Such violations

can directly undermine the security and correctness of the browser.

Contributions. We present BUIzz, the first testing framework designed to find policy enforcement bugs by simulating BUI-level user interactions. The core idea of BUIzz is to reuse existing test pages from test suites and prior studies, load them in the browser under test, and systematically simulate BUI-level interactions.

Devising a testing framework that finds policy enforcement bugs by simulating user interactions entails three technical challenges. (1) **Interaction collection**: it should comprehensively enumerate user interactions even though supported interactions vary across browsers, while filtering out interactions unrelated to testing. (2) **Interaction simulation**: it should simulate BUI-level interactions, which are not supported by existing browser automation tools. (3) **Bug oracle**: it should avoid cross-browser differential testing, which incurs prohibitive cost once interactions are considered. Such an oracle also fails when all browsers exhibit the same bug.

To address the first challenge, we collect a comprehensive set of BUI-level interactions by referencing browser manuals, right-click context menus, and known browser bugs. We then filter the set to retain only navigation-related interactions that can change the rendering environment. For the second challenge, we simulate BUI-level mouse and keyboard interactions (and their combinations) by leveraging OS-level automation frameworks.

To overcome the bug oracle challenge, we propose a pre/post-interaction bug oracle that compares the browser's enforcement behavior during the initial page visit with its behavior after applying an interaction. Any inconsistency is flagged as erroneous behavior. Our bug oracle avoids cross-browser differential testing, thereby enabling efficient bug detection within a single browser.

Using BUIzz, we found 38 unique policy enforcement bugs across seven security-related headers from six browsers: Chrome, Firefox, Edge, Opera, Brave, and Whale. Of these, 35 bugs undermine security, enabling attackers to bypass CSP, SameSite cookie, Permission Policy (PP), and Cross-Origin-Opener-Policy (COOP) and HTTP Strict Transport Security (HSTS). We reported all findings to the corresponding vendors; 14 have been patched, and 11 have been acknowledged and are pending fixes. We obtained seven CVEs and received USD 14,700 in bug bounties awarded by Microsoft Edge, Brave, and Naver Whale.

Overall, this paper makes the following contributions:

1. We design and implement BUIzz, the first testing framework that identifies policy enforcement bugs triggered by BUI-level interactions.

2. We propose a pre/post-interaction oracle that compares policy enforcement behavior before and after applying user interactions, enabling bug detection in a single browser.

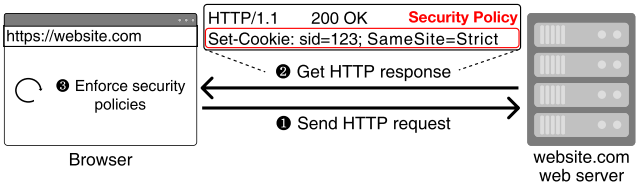

Figure 2: Workflow of HTTP header-based security policies.

3. We identify 38 policy enforcement bugs, including 35 security bugs. Browser vendors have patched 14 of them.

4. We open-source BUIzz to support open science and enable browser vendors to continuously test policy enforcement behaviors (§8).

2 Background

2.1 HTTP Header-based Security Policies

Modern web applications and servers rely on HTTP response headers to configure client-side web security policies [89]. These headers mitigate web attacks such as cross-site scripting (XSS), Cross-Site Request Forgery (CSRF), clickjacking, and Cross-Site Leaks (XS-leaks), as well as network attacks such as packet sniffing and spoofing.

Figure 2 illustrates the workflow of these security policies. ❶ When a browser visits a webpage, ❷ the web server/application specifies the security policies in HTTP response headers, and ❸ the browser interprets the received policies and enforces them on the webpage. We describe several policies and the threats they are intended to mitigate as follows (Table 2 summarizes this information in the first three columns).

Content Security Policy (CSP). CSP [10, 11, 12] governs the execution and inclusion of various resources, such as scripts or images. Originally CSP was proposed to mitigate XSS attacks [94], where arbitrary JavaScript (JS) code is executed on a vulnerable page. In particular, the `script-src` directive restricts where scripts can be loaded from and, by default, blocks inline scripts, making XSS exploitation harder.

CSP also supports sandboxing, which restricts a page's actions via the `sandbox` directive [49]. This sandboxing mitigates security risks from malicious scripts, plugins, or ads by restricting the script execution, file downloads, or popup creation. For example, script execution is blocked unless `allow-scripts` is specified. This directive also controls permissions for form submission (`allow-forms`), opening popup windows (`allow-popups`), and file downloads (`allow-downloads`).

SameSite cookie. The `SameSite` cookie attribute restricts when cookies are sent in cross-site contexts, mitigating CSRF attacks [72, 77]. With `SameSite=Lax`, cookies are sent only for top-level cross-site navigations using safe HTTP methods

(e.g., GET). With `SameSite=Strict`, cookies are not sent on any cross-site requests.

Permissions Policy (PP). PP [35] enables websites to specify which origins may access sensitive browser features (e.g., camera, microphone, geolocation). Similar to CSP, PP is configured via directives that allow or deny each feature for a set of origins. For example, `camera=(self "https://a.com")` permits camera access for the page's own origin and for content from `https://a.com`. By restricting powerful features to a limited set of trusted origins, PP can reduce the impact of XSS attacks and untrusted third-party contents.

Cross-Origin-Opener-Policy (COOP). COOP [13] controls whether a page can keep an opener relationship with pages from other origins (e.g., popups opened via `window.open`). For example, `same-origin` prevents cross-origin pages from accessing the opener via `window.opener` and isolates the page from cross-origin popups. By cutting these cross-origin links, COOP mitigates XS-Leaks [80, 87, 97] and Tabnabbing attacks [69].

HTTP Strict Transport Security (HSTS). HSTS [44] ensures that a site is always loaded over HTTPS. A site can enable HSTS by specifying a `max-age` attribute. Subsequent connections to the site will then use HTTPS for the specified period. In addition, browsers use an HSTS preload list that forces HTTPS for preloaded sites [20].

2.2 Policy Enforcement Bugs

By design, the browser is responsible for enforcing the declared policies while rendering web contents (❸ in Figure 2). Therefore, it is important for browser vendors to ensure that the browser adheres to given policies. However, if the browser fails to enforce these policies correctly, it poses critical security threats. In particular, the browser bugs can reopen the security threats listed in the "Threat to Mitigate" column of Table 2. We refer to such flaws as *policy enforcement bugs* [50, 52, 53].

To detect these bugs, Web Platform Tests (WPT) checks whether browser implementations follow specifications by running curated test cases with expected outcomes. However, WPT test suites are hand-crafted, which limit coverage and can leave many false negatives [72, 86, 102].

Recently, several works have proposed techniques that generate diverse test documents and policy configurations to systematically identify policy enforcement bugs. DiffCSP [102] identifies CSP enforcement bugs, while XSR-Framework [72] targets enforcement bugs related to SameSite cookies. Rautenstrauch *et al.* [86] further study enforcement bugs caused by syntactically malformed header configurations across multiple security headers.

```
1  Test CSP: script-src 'nonce-123'; ...
2  Test HTML:
3  <a id=x>Visit!</a>
4  <script nonce=123>
5    x.href = URL.createObjectURL(
6      new Blob(['<script>alert("CSP Bypass!")</script>'])
7    );
8    x.click();
9  </script>
```

(a) A test page generated by DiffCSP.

(b) Initial page visit (Chrome). (c) Initial page visit (Firefox). (d) A CSP enforcement bug in Chrome triggered by drag&drop.

Figure 3: Motivating example.

3 Motivation

At a high level, existing approaches [72, 86, 102] prepare test HTML documents along with the header policies and visit them using browsers under test to trigger bugs. However, these approaches share a critical limitation: they do not consider policy enforcement bugs triggered by *user interactions* on the browser user interface (BUI).

Yet many security policies are enforced across a sequence of user actions, and those interactions (e.g., context-menu actions and keyboard shortcuts) can change the browser's rendering state. Therefore, effectively testing policy enforcement requires exercising these user interactions, not just by visiting a page and observing document-level behavior like previous work [72, 86, 102].

Motivating example. Figure 3a shows a test page generated by DiffCSP [102], consisting of a CSP header (Line 1) and an HTML snippet (Lines 3–9). Under the policy in Line 1, browsers should allow execution of the nonce-protected script (i.e., Lines 5–8 with `nonce=123`) but must block execution of the nonce-less script (e.g., Line 6, embedded in a `blob` URL [2]). In detail, since a document loaded from a local scheme (e.g., `blob:` and `javascript:`) shares the parent's origin, it must inherit the parent document's CSP (Line 1) [14].

Previous approaches [72, 86, 102] visit the test page using browsers under test and observe their policy enforcement behaviors. Because policy specifications are too complex [88], prior works commonly leverage cross-browser differential testing as the bug oracle [65, 72, 86, 102]. For example, DiffCSP checks inconsistencies of the execution of JS in Line 6 between Chrome and Firefox (Figure 3b vs. Figure 3c). Since no inconsistency is observed, DiffCSP concludes that there is no CSP enforcement bug for this test page.

Unfortunately, this workflow overlooks the search space introduced by BUI-level interactions, resulting in many false negatives. Figure 3d shows a CSP enforcement bug we discovered in Chrome, triggered by a BUI-level interaction on the page shown in Figure 3a. In particular, when the link in

Line 3 of Figure 3a is dragged and dropped into the address bar, the nonce-less script is executed[1].

This class of bugs stems from browsers failing to consistently enforce policies on rendering changes triggered by interactions originating from outside the document's scope (§5.6). There have been no previous studies that systematically identify such bugs. We address this gap by comprehensively simulating BUI-level interactions on the existing test pages.

3.1 Technical Challenges and Our Approach

Identifying policy enforcement bugs triggered by BUI-level interactions poses three technical challenges: (1) collecting a comprehensive set of testing interactions, (2) simulating these interactions, and (3) identifying unexpected browser behaviors under the simulated interactions.

Challenge #1: interaction collection. Browsers provide various user interactions through (i) keyboard interactions (e.g., refresh using F5, reopen using Ctrl+Shift+T) and (ii) mouse interactions (e.g., →"Open in new tab", drag&drop). Notably, the set of supported interactions differs across browsers. Even browsers built on the same rendering engine may add *customized interactions* that are not available in the base engine. For example, although Edge [28] and Whale [33] are Chromium-based, they provide their customized interactions such as "open in split view" and "open in mobile view", respectively. It is important to comprehensively collect the set of user interactions across browsers, including customized ones, to systematically discover policy enforcement bugs.

Challenge #2: interaction simulation. There is no existing browser testing framework that can fully simulate BUI-level interactions, making it difficult to automatically test enforcement policies that depend on such interactions. Widely used automation tools (e.g., Selenium [54] and Playwright [38]) operate at the document level and cannot exercise interactions exposed through the browser UI. In particular, they lack the ability to simulate OS-level inputs, such as global keyboard shortcuts and native mouse interactions, that are beyond the scope of the HTML document.

Challenge #3: bug oracle. Prior work relies on cross-browser differential testing, i.e., detecting execution discrepancies between browsers (e.g., Figure 3b vs. Figure 3c). However, this strategy has fundamental limitations: (i) it cannot detect bugs when all tested browsers exhibit the same erroneous behavior; (ii) it is ineffective when a policy or feature is supported by only a single browser (e.g., Permissions Policy is supported by Chrome but not by Firefox); and (iii) when user interactions are considered, the verification cost increases substantially because multiple browsers must be tested under various interaction sequences.

[1]The parent CSP is not inherited at all. As a result, an XSS attacker can execute arbitrary JS under the vulnerable target origin. Other CSP guarantees, such as TLS enforcement and framing control, are also bypassed.

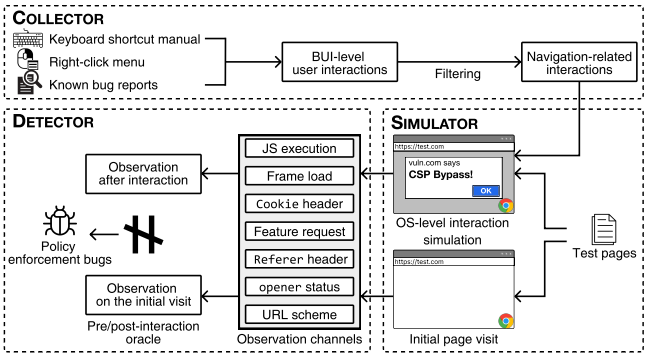

Figure 4: BUIzz architecture.

Our approach. We present BUIzz, a framework that identifies policy enforcement bugs via BUI-level interaction simulation. BUIzz addresses the three aforementioned technical challenges as follows. (1) It collects a comprehensive set of interactions by analyzing browser manuals, context-menu items, and previously reported bugs. (2) It simulates keyboard and mouse interactions at the OS level, going beyond document-level automation. (3) It leverages a pre/post-interaction oracle that compares the browser behavior during the initial page visit with its behavior after applying an interaction (e.g., Figure 3b vs. Figure 3d); any inconsistency is flagged as erroneous behavior.

In this study, we focus on *navigation-related* BUI-level interactions because navigation directly induces changes in the rendering state [72, 102]. Bugs triggered solely by *non-navigation* interactions (e.g., permission toggles via the site information bubble or clipboard access gestures) are out of scope of this paper (§6).

4 Design

Figure 4 illustrates the overall architecture of BUIzz, which consists of three components: Collector, Simulator, and Detector. At a high level, these components work together to conduct three steps: (1) the Collector collects a comprehensive set of BUI-level user interactions (§4.1); (2) the Simulator performs OS-level simulation of the collected interactions (and their combinations) on the test pages (§4.2); and (3) the Detector identifies policy enforcement bugs by checking for inconsistencies between the browser's behavior during the initial visit and its behavior after applying BUI-level interactions (§4.3).

Browsers. BUIzz is OS-dependent because it simulates user interactions at the OS level [68, 74, 82, 104]. We set our scope to Windows and target six browsers: Chrome, Firefox, Edge, Opera, Brave, and Whale (§5.1).

Header policies. We target seven widely studied security policies considered in prior works [64, 72, 86, 89, 102]: CSP (XSS mitigation, sandbox, and framing control), SameSite cookie, PP, COOP, HSTS, XFO, and RP.

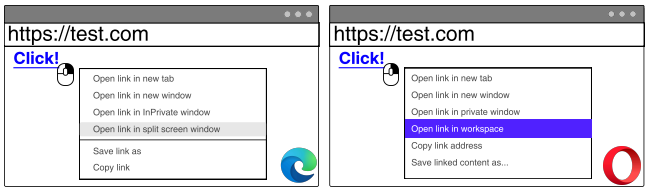

Figure 5: The context menu of Edge and Opera, triggered by right-clicking a link.

4.1 COLLECTOR

The COLLECTOR gathers BUI-level user interactions to be simulated. Our goal here is to compile, for each browser, a comprehensive set of meaningful interactions while excluding those irrelevant to our tests. In particular, we first collect all BUI-level interactions and then filter them to retain only navigation-related interactions.

Collecting BUI-level interactions. The set of supported interactions varies across browsers (§3.1). Moreover, even when browsers support the same action, the interaction required to trigger it can differ. For example, the key combination (i.e., shortcut) for duplicating the current tab is Ctrl+K in Opera but Shift+Ctrl+K in Edge. To comprehensively collect keyboard and mouse interactions for each browser, we reference browser manuals, right-click context menus, and known browser bugs.

For keyboard interactions, the supported interactions are well documented in browser manuals [7, 24, 25, 26, 30, 59]. We therefore crawl these manuals and extract all available keyboard shortcuts along with their corresponding meanings.

For mouse interactions (and mouse–keyboard interaction combinations), we first extract the interactions documented in browser manuals [55, 60]. However, browsers also expose additional mouse interactions that are not explicitly documented, such as "Open link in incognito window" or "Open link in split view". To capture these interactions, we additionally enumerate all context-menu items displayed when right-clicking a link. Figure 5 shows the right-click menus of Edge and Opera, which differ across browsers. We treat each menu item as a distinct mouse interaction, yielding a comprehensive per-browser interaction set. In addition, we include common mouse interactions that are typically undocumented, such as clicking the Forward (→), Back (←), and Reload buttons, as well as left-clicking a link to open it in the current tab.

Using this process, we collected an average of 81 BUI-level user interactions per browser. To check for additional interactions, we examined 22 known browser bug reports involving BUI-level interactions (§C). We confirmed that all interactions required to trigger these bugs are included in our interaction set, showing that our collection provides broad coverage of interactions.

Filtering navigation-related interactions. Not all collected interactions are relevant for testing. For example, a mouse interaction such as clicking "Search with your default search engine" is irrelevant to testing because it does not change the rendering environment of the test pages. We treat an interaction as relevant for testing if it supports navigation to the test pages, which in turn changes the rendering context. We thus filter the collected interactions in a semi-automated manner, retaining only navigation-related ones.

Type	Interaction	Description	Browsers
Mouse	(Left click) the link	Open link in current tab	✓ ✓ ✓ ✓ ✓ ✓
	(Middle click) the link	Open link in background tab	✓ ✓ ✓ ✓ ✓ ✓
	(Drag&drop) the link onto the tab	Open link in new tab and move to page	✓ ✓ ✓ ✓ ✓ ✓
	(Right click) the link → Select a context-menu item	Open link in new window	✓ ✓ ✓ ✓ ✓ ✓
		Open link in new tab	✓ ✓ ✓ ✓ ✓ ✓
		Open link in incognito window	✓ ✓ ✓ ✓ ✓ ✓
		Open link in split view	✗ ✗ ✓ ✗ ✓ ✓
		Open link in mobile view	✗ ✗ ✗ ✗ ✗ ✓
		Open link in side bar	✗ ✗ ✗ ✗ ✗ ✓
		Open link in workspace	✗ ✗ ✗ ✓ ✗ ✗
Keyboard	Ctrl+Shift+T	Reopen closed tab	✓ ✓ ✓ ✓ ✓ ✓
	Alt+←	Navigate backward	✓ ✓ ✓ ✓ ✓ ✓
	Alt+→	Navigate forward	✓ ✓ ✓ ✓ ✓ ✓
	F5	Reload page	✓ ✓ ✓ ✓ ✓ ✓
	Ctrl+F5	Reload without cache	✓ ✓ ✓ ✓ ✓ ✓
	: Ctrl+Shift+K : Ctrl+K	Duplicate tab	✗ ✗ ✓ ✓ ✗ ✓
+	Ctrl+F5+the link	Force open link in current tab	✗ ✗ ✗ ✗ ✗ ✓

Table 1: Navigation-related BUI-level interactions.

To this end, we first discard interactions whose descriptions do not contain navigation-related keywords (i.e., "open", "load", "navigate", and "duplicate"). From the remaining interactions, we then manually extract those related to navigation to the test pages. Specifically, we consider an interaction as navigation-related if it triggers a new HTTP request for the test page. Finally, when a keyboard and a mouse interaction have overlapping semantics (e.g., Alt+→ vs. clicking Forward → button), we retain only the keyboard interaction to reduce simulation complexity.

Filtering process summary. Out of 486 BUI-level interactions collected across six browsers in total, we first reduced them automatically to 225 through keyword-based filtering for navigation-related interactions. We then manually excluded 149 interactions that did not trigger network requests to the test pages and further removed semantically duplicate interactions, resulting in 76 navigation-related interactions in total. For each browser, two researchers spent up to 36 hours validating navigation-related interactions and implementing simulations. This cost varies with the diversity of browser-supported interactions, while common interactions (e.g., Ctrl+Shift+T) require substantially less implementation effort.

Table 1 summarizes the navigation-related BUI-level interactions used in our testing. In total, we leverage 17 interactions, consisting of 10 mouse interactions, six keyboard interactions, and one mouse–keyboard interaction combina-

Algorithm 1: Simulation and Scenario Generation

```
1  def Simulation(browser, interactions, test_pages)
2      foreach test_page ∈ test_pages do
3          pre_interaction ← Visit(browser, test_page)
4          Preprocess(test_page)
5          foreach scenario ∈ ScenarioGen(test_page, interactions) do
6              post_interaction ← OsSimulation(browser, test_page, scenario)
7              Detector(pre_interaction, post_interaction)
8  def ScenarioGen(test_page, interactions)
9      scenarios ← []
10     foreach interaction ∈ interactions do          // Single interaction
11         if test_page is satisfying interaction.precondition then
12             scenarios.append(interaction)
13     /* Two-interaction combinations */
14     scenarios ← CombTwoInteractions(test_page, interactions, scenarios)
15     return scenarios
```

tions. If a browser supports a given interaction, we mark it with a ✓, and ✗ otherwise.

4.2 SIMULATOR

Given the collected interactions, the SIMULATOR simulates them on the browser loading the test page. Specifically, to simulate interactions at the browser UI level, we leverage OS-level simulation techniques. We first describe how we prepare the test pages, then explain the overall simulation workflow, and finally detail the OS-level interaction simulation.

Test pages. We leverage existing test pages (i.e., test headers and HTML snippets) from prior works [72, 102] and the WPT test suites [57]. The "Test Page Sources" and "# of Pages" columns in Table 2 summarize the sources of the test pages (i.e., prior works or WPT) and the number of pages extracted from each source.

For CSP (XSS mitigation) and SameSite cookie, DiffCSP [102] and XSR-Framework [72], respectively, have already constructed richer enforcement test pages beyond WPT; thus, we reuse their sets. Using the full DiffCSP test pages (25,880 HTML files and 1,006 CSP headers [102]) is impractical, as each page must be exercised with 17 BUI-level interactions. We therefore reduce them to a smaller, semantically non-redundant subset that preserves bug coverage, resulting in 500 HTML files and 21 CSP headers (Refer to §B in Appendix).

For PP, we reuse the HTML snippets from DiffCSP, as they embed JS APIs accessing browser features in diverse ways. For HSTS and RP, we reuse the HTML snippets from XSR-Framework, as they issue requests in diverse ways. For the remaining policies, including CSP (sandbox), COOP, CSP (framing control), and XFO, we reuse WPT as the test pages.

In this study, our goal is to expand the existing search space by simulating BUI-level interactions on top of the existing test pages. Generating new test HTML documents or policies [72, 78, 81, 84, 99, 102] is out of scope and orthogonal to our work.

Simulation workflow. Algorithm 1 describes the overall workflow of the SIMULATOR. The `Simulation` function takes as input the browser under test (*browser*), the supported interactions (*interactions*), and the test pages (*test_pages*). For each test page (*test_page*) in *test_pages* (Line 2), it records observations from a baseline visit (*pre_interaction*, Line 3) and after simulating each BUI-level interaction sequence (*post_interaction*, Line 6), and forwards both to the DETECTOR (§4.3) to identify bugs. The types of information that we observe in Lines 3 and 6 are described in §4.3.

Before simulating BUI-level interactions, we preprocess the test page (Line 4) to remove any JS-driven clicks (e.g., removing Line 8 in Figure 3a). This ensures that navigation is triggered by BUI-level interactions rather than DOM-level interactions. In Line 5, the `Simulation` function invokes `ScenarioGen` to generate simulation scenarios, each of which consists of either a single interaction or a two-interaction combination.

The `ScenarioGen` function first plans simulations for each single interaction (Lines 10–12). For each interaction, it checks in Line 11 whether the interaction's precondition is satisfied on the current test page (see the "Precondition" column of Table 8 in Appendix). If so, the interaction is added to *scenarios* (Line 12). This precondition checking is designed to avoid adding meaningless interactions; for example, when a test page contains no links required for navigation, mouse interactions initiated via links are not applicable and are therefore not added to *scenarios* for that page.

This function also adds interaction combinations (Line 14) to capture policy enforcement bugs that arise only under nested BUI-level interactions. Because longer interaction sequences are unlikely to be exercised by real-world victim clients and thus typically indicate lower severity, we limit combinations to two interactions. Weinreich *et al.* [101] similarly observed that users generally browse in short sessions with limited interaction sequences. Additionally, we tested 2,000 sampled three-interaction combinations and found no bugs, further supporting our decision to limit interaction combinations to two.

For all policies except CSP (XSS mitigation), we include all possible two-interaction combinations in `scenarios`. For CSP (XSS mitigation), considering the large number of test pages, we include only one randomly sampled two-interaction combination per test page.

OS-level simulation. The `OsSimulation` function simulates user interactions (and their combinations). Existing browser automation frameworks [38, 54] only support document/page-level interactions through high-level APIs and do not support interactions with browser user interfaces, such as shortcut keys, context-menu operations, or drag&drop. Therefore, we simulate keyboard and mouse interactions at the OS level, which is equivalent to real user interactions.

This function leverages pyautogui [40] to simulate keyboard interactions, accurately emulating user-entered shortcut keys at the OS level. Simulating mouse interactions requires locating the exact on-screen position of the target link on the

Policy	Field	Threat to Mitigate	Test Page Sources	# of Pages	Observation
CSP (XSS mitigation)	`Content-Security-Policy`	XSS attacks (`script-src`)	DiffCSP (21 headers × 500 HTML files)	10,500	JS execution
CSP (sandbox)	`Content-Security-Policy`	Untrusted code execution (`sandbox`)	WPT test suites	5	JS execution
SameSite Cookie	`Set-Cookie`	CSRF attacks (`SameSite=Strict/Lax`)	XSR-Framework (1 header × 207 HTML files)	207	`Cookie` header
PP	`Permissions-Policy`	Abuse of privileged features	1 header × DiffCSP's 500 HTML files	500	Feature request
COOP	`Cross-Origin-Opener-Policy`	XS-Leaks	WPT test suites	16	`opener` status
HSTS	`Strict-Transport-Security`	Man-in-the-Middle attacks	1 header × XSR-Framework's 207 HTML files	207	URL scheme
CSP (framing control)	`Content-Security-Policy`	Clickjacking attacks (`frame-ancestors`)	WPT test suites	9	Frame load
XFO	`X-Frame-Options`	Clickjacking attacks	WPT test suites	9	Frame load
RP	`Referrer-Policy`	`Referer` header leaks	8 headers × XSR-Framework's 207 HTML files	1,656	`Referer` header

Table 2: Summary of security policies, the threats they are intended to mitigate, the test pages used for evaluation, and the observation channels.

test page; however, the link coordinates vary across test pages and can also differ across browsers due to layout differences.

We compute the exact link coordinates in a browser-independent manner by combining pywinauto [41] and Playwright [38]. In particular, pywinauto provides the browser window's (x, y) offset from the screen's top-left corner, and Playwright provides the link's (x, y) offset from the browser window's top-left corner. By summing these two offsets, we obtain the link's absolute on-screen position. We then use pyautogui [40] to perform mouse interactions at the computed link coordinates.

For context-menu interactions, we right-click ( ) at the computed link coordinate, select the target menu item by sending repeated Down-arrow keystrokes ( ), and press Enter ( ) to apply it. For drag&drop interactions, we compute the on-screen coordinates of a valid drop target (e.g., another tab or the address bar) and drag the link to that target.

Several interactions require prerequisite actions. For example, "reopen a closed tab" ( ) requires first closing the test-page tab, and "navigate backward" ( ) requires first visiting another page from the test page to create a history entry. The `OsSimulation` function performs these prerequisite actions before applying the interaction, ensuring that the navigation is effective on the test page. The "Prerequisite Actions" column in Table 8 summarizes the required actions for each interaction.

4.3 DETECTOR

After simulating interactions, the DETECTOR determines whether each interaction triggers a policy enforcement bug. The DETECTOR does not rely on cross-browser differential testing used in prior work [65, 72, 86, 102]; instead, it uses a pre/post-interaction oracle that compares observations before and after applying BUI-level interactions.

To this end, the DETECTOR extracts the test policy's enforcement outcomes from both *pre_interaction* and *post_interaction* (Algorithm 1) via observation channels. It then applies a pre/post-interaction oracle that compares the two outcomes and flags any inconsistency as a policy enforcement bug.

Observation channels. Observation channels specify what information should be observed as the outcome of BUI-level interactions. The enforcement outcome to be observed differs across security policies. For example, for CSP (XSS mitigation), we observe whether embedded JS code is executed, whereas for SameSite cookie, we inspect the `Cookie` header to determine whether `Strict` or `Lax` cookies are sent. Accordingly, we manually define the observation channels for each policy. The "Observation" column of Table 2 summarizes the observation channels used to check enforcement behavior for each security policy.

Pre/post-interaction oracle. Given the extracted enforcement outcomes from `pre_interaction` and `post_interaction`, the DETECTOR applies a pre/post-interaction oracle that compares the two outcomes. Any discrepancy provides evidence of an enforcement violation because security policies should be applied consistently when the same page is revisited through BUI-level navigation. This oracle is single-browser: it does not require another browser as a reference and scales to policies/features that are missing or implemented differently across browsers (§5.4).

5 Evaluation

We evaluate the efficacy of BUIZZ in finding policy enforcement bugs triggered by BUI-level interactions (§5.2). We then explain the identified bugs for each policy and discuss their security implications (§5.3). We also analyze the effectiveness of the pre/post-interaction oracle by comparing it with cross-browser differential testing (§5.4) and evaluate the efficacy of the exercised BUI-level user interactions (§5.5). Finally, we summarize the lessons learned from our study (§5.6).

5.1 Experimental Setup

We started our evaluation in January 2026. The total testing time was 196 hours.

Implementation. We use Playwright [38] 1.53 to visit web pages and pyautogui [40] 0.6.9 to perform OS-level simulation of keyboard and mouse interactions. We also use pywinauto [41] 0.9.54 to extract link coordinates on the screen.

Browsers. We ran a series of experiments on the six browsers: Chrome 138, Firefox 139, Edge 136, Opera

118, Brave 1.79, and Whale 4.32. We selected these browsers based on the following criteria: (i) they are available on Windows and (ii) they have been studied in prior browser-security work [66, 79, 83].

Environment. BUIZZ simulates interactions at the OS level, which requires a foreground window. As a result, we cannot run browsers in headless mode or execute multiple tests concurrently on a single machine. Therefore, we parallelize testing across multiple desktop machines. In total, we use 12 desktops with CPUs ranging from Intel(R) N95 to AMD Ryzen 7 1800X. For each policy-browser pair, we distribute the test page URLs across the 12 machines; each machine visits its assigned pages using the browser under test and simulates BUI-level interactions (Algorithm 1). Refer to Table 6 in Appendix for more details of the experimental environment.

Test pages and scenarios. We collect a total of 13,109 test pages from WPT and prior studies (Table 2). Each test page consists of a security policy and an associated HTML document. The left part of Table 3 summarizes, for each browser, the number of generated scenarios and the corresponding execution time when running on 12 machines in parallel.

The number of scenarios is the sum of single-interaction scenarios and two-interaction combinations considered in Algorithm 1. The number of scenarios varies across browsers because the set of supported BUI-level interactions differs by browser (Table 1). In addition, the number of scenarios differs across policies, as it is influenced by the number of test pages and thus primarily depends on the search space of the WPT and prior studies. Table 5 in the Appendix further breaks down the number of scenarios and execution time for single interactions and two-interaction combinations.

Execution time also varies across browsers depending on the number of test scenarios. In our experiments, it ranges from 28 to 38 hours per browser, with an average execution time of 32.5 hours.

5.2 Bugs Found

The right part of Table 3 summarizes the number of inconsistencies observed by our bug oracle (the "# of Pre/Post-interaction Inconsistencies" column) and the number of browser bugs derived from these inconsistencies (the "Total # of Bugs" column). We found a total of 38 distinct bugs triggered by BUI-level interactions after analyzing 23,987 discrepancies that BUIZZ reported.

Bug counting. To count distinct bugs, we group reported inconsistencies by a triple consisting of (1) the triggering BUI-level interaction, (2) the HTML tag or JS API used for navigation, and (3) the navigated URL scheme. For example, the inconsistency in Figure 3a is represented as (drag&drop, `<a>`, `blob:`). We then empirically merge groups that exhibit the same root cause. In Figure 3a, the CSP enforcement bug is caused by drag&drop navigation to a `blob:` URL; under this condition, inconsistencies triggered via different navigation tags (e.g., `<a>` vs. `<img>`) share the same root cause, so we merge the corresponding groups.

To determine whether multiple groups correspond to the same bug, we cross-referenced their triads (interaction, tag/API, scheme). Especially, we conservatively merged groups only when the cross-referencing strongly indicated that they shared the same root cause. In particular, we applied two grouping rules: (1) when one dimension of a triad spans all possible values, we automatically merge the corresponding groups by normalizing that dimension to "Any"; and (2) for SameSite cookie and HSTS bugs, we manually merge `<a>`, `<area>`, and `<svg>` because they all induce top-level navigation. Although the latter rule involves empirically motivated manual merging, the overall grouping process remains largely scalable and reproducible.

Through this process, we reduced the number of groups from 225 to 133, with an average of 22 groups per browser. Each group contains 106 pages on average, ranging from 1 to 336 pages. By analyzing the resulting groups, we identified 38 distinct bugs. We further discuss how the 133 groups were consolidated into 38 bugs and the reliability of the merging process in §6.

Note that when multiple browsers share the same browser engine (e.g., Chromium), we count a shared bug only once, attributing it to the upstream engine. For example, if the same bug appears in , , , , and , we count it only for . Therefore, bugs counted for , , , and in Table 3 correspond to browser-specific bugs introduced by customized rendering features rather than the upstream browser engine.

Of the 38 browser bugs, we manually confirmed that 35 pose a security threat and three are functional bugs. We classify a bug as a security bug when applying an interaction weakens the intended policy enforcement, and as a functional bug when the interaction results in stricter enforcement than intended. Appendix §A describes the three functional bugs, which involve SameSite cookie and RP enforcement.

False positives (FPs). During our analysis, we identified two FP groups related to (1) the `SameSite=Lax` cookie attribute and (2) the CSP `sandbox` directive. Here, an FP means that our oracle flags a pre/post-interaction inconsistency, but the observed behavior after applying interactions matches the specification-defined behavior [5, 46, 48], and all six tested browsers behave consistently.

For example, a JS-driven click on a link inside an iframe results in iframe-level navigation, so `SameSite=Lax` cookies are not included. In contrast, accessing the same link via a BUI-level interaction such as "Open link in new tab" results in top-level navigation, and the `SameSite=Lax` cookie is correctly included [46]. These FPs arise because our oracle only observes pre/post BUI-interaction differences without incorporating specification-specific knowledge.

Security bugs. The 35 security bugs consist of 25, 7, 1, 1, 1 enforcement bugs for CSP, SameSite cookie, PP, COOP, HSTS, respectively. Note that we classify an issue to be a

Browser	# of Scenarios							Total # of Scenarios	Execution Time	# of Pre/Post-interaction Inconsistencies (Distinct Bugs)							Total # of Bugs (Security Bugs)
	CSP	SameSite	PP	COOP	HSTS	XFO	RP			CSP	SameSite	PP	COOP	HSTS	XFO	RP	
Chrome	80,651	6,216	15,821	521	6,216	270	49,728	159,423	28h	1,930 (7)	28 (0)	48 (1)	0 (0)	0 (0)	0 (0)	256 (1)	9 (8)
FireFox	80,651	6,216	15,821	521	6,216	270	49,728	159,423	28h	50 (1)	118 (2)	0 (0)	0 (0)	1 (1)	0 (0)	256 (1)	5 (3)
Edge	94,787	8,772	23,789	809	8,772	378	70,176	207,483	36h	4,794 (3)	112 (0)	192 (0)	0 (0)	0 (0)	0 (0)	512 (0)	3 (3)
Opera	94,765	8,695	21,985	721	8,695	378	69,560	204,799	36h	2,554 (2)	36 (1)	60 (0)	0 (0)	0 (0)	0 (0)	256 (0)	3 (3)
Brave	84,116	6,293	17,625	609	6,293	270	50,344	165,550	29h	4,914 (2)	106 (1)	180 (0)	0 (0)	0 (0)	0 (0)	512 (0)	3 (3)
Whale	105,129	8,807	24,609	849	8,807	378	70,456	219,035	38h	6,082 (10)	126 (4)	240 (0)	112 (1)	0 (0)	0 (0)	512 (0)	15 (15)

Table 3: Overall results of BUIzz.

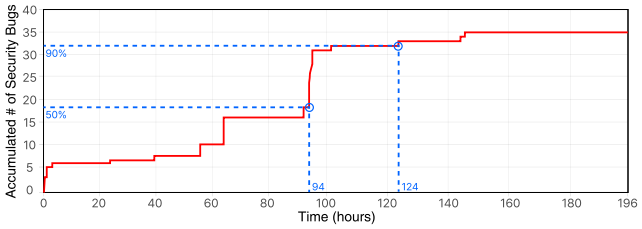

Figure 6: The cumulative number of bugs discovered over full time.

security bug when there is clear evidence that the security guarantees are weakened compared to the initial page visit (i.e., enforcement becomes less restrictive after applying BUI-level interactions).

The security implications of these bugs vary by policy. For example, CSP enforcement bugs can allow an adversary to execute injected JS that should be blocked under the CSP standard. SameSite cookie enforcement bugs allow a CSRF attacker to induce a cross-site request that incorrectly includes `Strict` cookies, enabling a CSRF attack. PP bypass bugs allow an XSS attacker to access sensitive browser features such as the camera or geolocation (§5.3).

We reported all 35 security bugs to the corresponding browser vendors [6, 9, 19, 31, 32, 34]. At the time of writing, 14 bugs have been patched in response to our bug reports. In addition, 11 bugs have been acknowledged by vendors and are scheduled to be fixed. Two bugs were marked as duplicates of previously reported issues, and the remaining seven bugs are still awaiting vendor responses.

We got a total of seven CVEs[2] for our reports. The Microsoft Edge, Brave, and NAVER Whale teams awarded us $14,700 in bug bounties, indicating the high severity of the reported browser bugs.

Time-sensitive analysis. Figure 6 shows the cumulative number of bugs discovered over time. BUIzz identified 50% of the bugs within 94 hours and 90% within 124 hours, indicating that many bugs are discoverable at an early stage of the testing process.

[2]CVE-2025-53600, CVE-2025-62583, CVE-2025-62584, CVE-2025-62585, CVE-2025-53791, CVE-2025-48980, CVE-2025-6923

5.3 Qualitative Analysis

Table 4 lists the 35 identified bugs along with their triggering inputs (i.e., BUI-level interaction, HTML tag or JS API, and URL scheme), bug descriptions, affected browsers, and bug report status. In the following, we describe a per-policy qualitative analysis of the identified enforcement bugs, including their triggering inputs and security implications.

CSP (XSS mitigation) bypass. We found 18 bugs while testing CSP enforcement (Idx #1–#15 in Table 4). Among them, 17 bugs (Idx #1–#14) are triggered when the victim navigates to a local scheme (i.e., `blob:` or `data:`) via mouse interactions (e.g., Figure 3). In these bugs, the parent CSP is not correctly inherited during navigation, or top-level navigation to `data:` URLs is improperly allowed. These behaviors violate the CSP and HTML specifications [3, 14, 15], which can enable XSS and phishing attacks. We also found one bug (Idx #15) where CSP is enforced correctly on the initial page visit, but is stripped after the user closes the tab and reopens it via Ctrl+Shift+T.

Notably, the five Chromium bugs (Idx #1–#5) are triggered by all collected mouse interactions except the middle click interaction (Table 1). In contrast, correctly enforces CSP and restricts top-level navigation for these interactions. This suggests that browsers vary in how thoroughly they consider security implications of BUI-level interactions.

Interestingly, the 12 bugs BUIzz found in , , , and (Idx #6–#14) are browser-specific security bugs that do not exist in the upstream Chromium engine. In particular, these bugs are triggered by BUI-level interactions newly customized by each browser (e.g., "open link in split view"). This indicates a tendency for browser vendors to overlook policy enforcement for newly introduced BUI-level interactions. We further summarize key takeaways from this class of bugs in §5.5 and §5.6.

All identified CSP bypass bugs have critical security implications. On an XSS vulnerable website, a web attacker can execute arbitrary JS under the target origin even if the CSP is correctly configured. More critically, other CSP guarantees, such as TLS enforcement and framing control, are also bypassed.

We validate whether the CSP test page reduction and random sampling of interaction combinations (§4.2) affect the qualitative analysis of the CSP testing results. To assess the impact of CSP test-page reduction, we sampled 2,000 dis-

Idx	Policy	Triggering Condition: BUI-level Interaction	Triggering Condition: Tag / API	Triggering Condition: URL Scheme	Bug Description	Browser	‡Report Status	Bug Count
1		(Drag&drop) the link	Any	`blob:`	CSP is not inherited		Ack.	1
2		(Drag&drop) the link	Any	`data:`	Invalid navigation		Ack.	1
3		the link→ Open link in new window	Any	`data:`	Invalid navigation		Ack.	1
4		the link→ Open link in new tab	Any	`data:`	Invalid navigation		Ack.	1
5		the link→ Open link in incognito mode	Any	`data:`	Invalid navigation		Ack.	1
6		the link→ Open link in split view	Any	`data:`	Invalid navigation	, ,	Rep.	3
7	**CSP:** XSS mitigation	the link→ Open link in split view	Any	`blob:`	CSP is not inherited	,	Fix.	2
8		the link→ Open link in mobile view	Any	`blob:`	CSP is not inherited		Fix.	1
9		the link→ Open link in mobile view	Any	`data:`	Invalid navigation		Rep.	1
10		the link→ Open link in side bar	Any	`blob:`	CSP is not inherited		Fix.	1
11		the link→ Open link in side bar	Any	`data:`	Invalid navigation		Rep.	1
12		the link→ Open link in workspace	Any	`data:`	Invalid navigation		Rep.	1
13		†the link→ Open link in split view→ the link	Any	`data:`	Invalid navigation		Fix.	1
14		†the link→ Open link in split view→ the link	Any	`blob:`	CSP is not inherited		Fix.	1
15		Ctrl+Shift+T (Reopen tab)	Any	`blob:`	CSP is not enforced		Ack.	1
16	**CSP:** Sandbox	Ctrl+Shift+T (Reopen tab)	Any	`http(s):`	Sandboxing is not enforced	,	Ack.	2
17		Ctrl+Shift+K, Ctrl+K (Duplicate tab)	`<a>`	`http(s):`	Sandboxing is not enforced	, ,	:Ack./:Rep.	3
18		†the link→ Open link in split view→ the link	Any	`http(s):`	Sandboxing is not inherited	,	Fix.	2
19	**SameSite cookie**	(Drag&drop) the link	`<a>`	`http(s):`	`Strict` cookie is included	,	Dup.	2
20		the link→ Open link in split view	`<a>`, `<area>`, `<svg>`	`http(s):`	`Strict` cookie is included	,	Fix.	2
21		the link→ Open link in mobile view	`<a>`, `<area>`, `<svg>`	`http(s):`	`Strict` cookie is included		Fix.	1
22		the link→ Open link in side bar	`<a>`, `<area>`, `<svg>`	`http(s):`	`Strict` cookie is included		Fix.	1
23		†the link→ Open link in split view→ the link	`<a>`, `<area>`, `<svg>`	`http(s):`	`Strict` cookie is included		Fix.	1
24	**PP**	Any	Any	`blob:`	PP is not inherited		Ack.	1
25	**COOP**	the link→ Open link in split view	`<a>`	`http(s):`	`window.opener` is created		Fix.	1
26	**HSTS**	the link→ Open link in incognito mode	`<a>`, `<area>`, `<svg>`	`http(s):`	HSTS is not enforced		Dup.	1

† Triggered by the two-interaction combination. ‡ **Ack.**: Acknowledged (promised to fix), **Fix.**: Fixed, **Rep.**: Reported (awaiting response), **Dup.**: Duplicated report.

Table 4: Details of the policy enforcement bugs discovered by BUIZZ.

carded cases (including `onclick`, non-ASCII characters, and `about:blank`; §B) and tested them, but found no additional bugs. To evaluate sensitivity to randomness, we repeated the sampling-and-testing process five times on Whale; all runs consistently produced the same two bugs.

This stability is partly attributable to DiffCSP's grammar-based generation [102], which introduces component-level redundancy. Overall, our search space reduction is unlikely to affect the CSP testing results.

CSP (sandbox) bypass. BUIZZ identified seven CSP sandbox enforcement bugs (Idx #16–#18). In all cases, the `sandbox` directive is stripped after applying interactions. The following HTML snippets are used to trigger these bugs:

```
1  Site URL: https://a.com
2  Test CSP Sandbox: sandbox 'allow-popups'
3  <a target="blank" href="https://b.com">
4      Visit!
5  </a>
6
7  Site URL: https://b.com
8  <script>alert("Sandbox bypass!")</script>
```

According to the HTML specification [5, 21, 48], when the user clicks () the link on Line 4 to navigate to a different origin, the `sandbox` directive in Line 2 should be inherited and thus enforced on `https://b.com`. Therefore, the browser should block the execution of the script in Line 8. More generally, most capabilities on the page should be restricted, including script execution, file downloads, and form submission, because the policy in Line 2 does not include `allow-scripts`, `allow-downloads`, or `allow-forms` (§2).

However, we observed that the policy in Line 2 is stripped when the victim reopens or duplicates the `https://b.com` page (Idx #16–#17), allowing the execution of the script in Line 8. Notably, the bugs in Idx #16 occur in all browsers, making it difficult to detect via cross-browser differential testing. BUIZZ identifies these bugs using the pre/post-interaction oracle (§5.4). For bugs in Idx #17, although the "duplicate tab" interaction differs across browsers (Ctrl+Shift+K in vs. Ctrl+K in and), we were able to identify all bugs. This demonstrates the efficacy of our manual-based interaction collection (§4.1).

We also observed two bugs in and (Idx #18) where the `sandbox` directive is not inherited when navigation is triggered by a two-interaction combination. Figure 7 shows the steps to trigger this bug: ❶ open link in a split view via the context menu, and ❷ left-click the link () in the original page. After this click, the `sandbox` directive in Line 2 is incorrectly not inherited.

These bugs allow an attacker to relax the restrictions intended by the page's sandbox protection, enabling the execution of malicious scripts/plugins, malicious file downloads, or form submissions.

SameSite cookie bypass. The seven bugs in Idx #19–#23 enable bypasses of the SameSite cookie attribute. The following

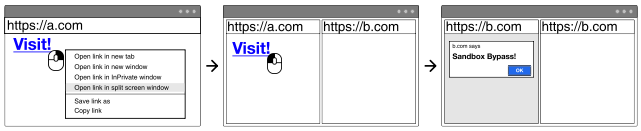

Figure 7: CSP (sandbox) enforcement bugs for Idx #18 triggered by two-interaction combinations.

HTML snippet triggers this behavior:

```
1  Site URL:  https://attacker.com
2  <a href="https://target.com">
3      Visit! // Apply interactions (Idx #19–#23) on this link!
4  </a>
```

According to the specification [16, 45], `SameSite=Strict` cookies bound to `https://target.com` must never be included in a cross-site request initiated by interacting with the link in Line 3. However, , , , and incorrectly include `Strict` cookies when navigation is initiated via certain mouse interactions, such as drag&drop or open link in split view. Notably, the bug (Idx #19), although upstream Chromium supports drag&drop, vendor's customization of this interaction introduces the bug.

These bugs allow an attacker to bypass SameSite cookie protections and mount CSRF attacks. However, because BUIZZ's SameSite cookie testing is constrained to the XSR-Framework [72] search space (Table 2), the identified bugs are limited to cases triggered by GET requests. While this constraint makes exploitation harder, these results still highlight the need to systematically test BUI-level interactions to ensure that browsers enforce the policy consistently.

PP bypass. A bug in Idx #24 strips Permissions Policy (PP). The following page triggers the browser bug:

```
1  Test PP: geolocation=()
2  <a id=x>Visit!</a> // Apply any interactions on this link!
3  <script>
4    x.href = URL.createObjectURL( new Blob([`
5      <script>geolocation.getCurrentPosition();</script>
6    `]));
7  </script>
```

The policy in Line 1 fully blocks access to geolocation. This policy should be inherited when the user navigates to the `blob:` URL in Line 2 [22, 37]. However, in , applying any of the collected interactions to this link causes the policy to not be inherited, allowing the destination page to access the geolocation. As a result, an XSS attacker can bypass PP and abuse sensitive browser features (e.g., geolocation or camera), thereby expanding the impact of the attack.

COOP and HSTS bypass. BUIZZ found one COOP and one HSTS enforcement bug in and , respectively. The former bug occurs when applying the "Open link in split view" interaction to an `<a>` link, causing `window.opener` to be incorrectly created [27]. This enables XS-Leaks and Tabnabbing attacks. The latter bug occurs when navigation is performed via the "Open link in incognito mode" interaction, in which case the request is sent over HTTP without being upgraded. As a result, this navigation becomes vulnerable to man-in-the-middle attacks.

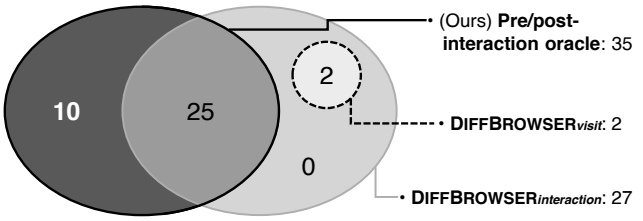

Figure 8: The comparison of distinct security bugs found by BUIZZ, DIFFBROWSER$_{interaction}$, and DIFFBROWSER$_{visit}$.

5.4 Bug Oracle

We compared BUIZZ's oracle with the cross-browser differential testing oracle used in prior work, which we denote as DIFFBROWSER. DIFFBROWSER performs differential testing across six browsers using our testing HTML and policy inputs (Table 2). In particular, we consider two baselines:

- DIFFBROWSER$_{visit}$: runs DIFFBROWSER on the initial page visit only, without applying any BUI-level interactions. This matches the input setting used in prior work [65, 72, 86, 102].
- DIFFBROWSER$_{interaction}$: runs DIFFBROWSER after applying the BUI-level interactions, making its input conditions identical to BUIZZ.

We compared (1) the security bugs discovered by each approach and (2) the time required to detect them.

Bug findings. Figure 8 depicts the Venn diagram of unique security bugs found in six browsers. Note that BUIZZ found 10 bugs that DIFFBROWSER missed. Baselines fail to find these bugs due to three reasons. First, two bugs (Idx #16 in Table 4) occur in all six browsers (and), so DIFFBROWSER observes no cross-browser inconsistency. Second, DIFFBROWSER misses one PP-specific enforcement bug (Idx #24) because does not support PP [18, 36], while the do. Finally, baselines cannot detect seven bugs triggered by browser-specific interactions that only one browser supports (Idx #8–12, 21, and 22).

In summary, cross-browser differential testing fails when all browsers share the same bug or when a bug depends on a single-browser-specific feature. In contrast, BUIZZ can detect these bugs in a single browser by checking a pre/post-interaction policy-enforcement invariant, demonstrating the efficacy of our bug oracle.

We observed that BUIZZ produced two false negatives that only DIFFBROWSER detected. One is the lack of PP support in [18, 36]. BUIZZ cannot detect this issue because it does not use cross-browser references. The other bug is that does not apply HSTS to subresources [17]. This behavior is consistent on the initial visit and after applying interactions. These results demonstrate that our oracle and the cross-browser oracle are complementary for finding policy enforcement bugs. DIFFBROWSER$_{visit}$ (i.e., prior work) found

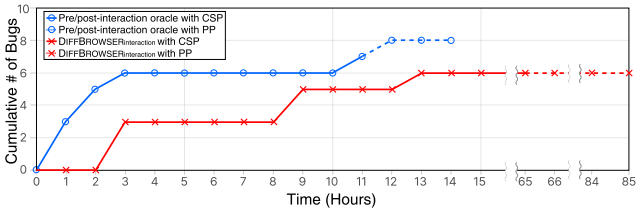

Figure 9: The cumulative number of bugs discovered over time for the pre/post-interaction oracle and DIFFBROWSER$_{interaction}$.

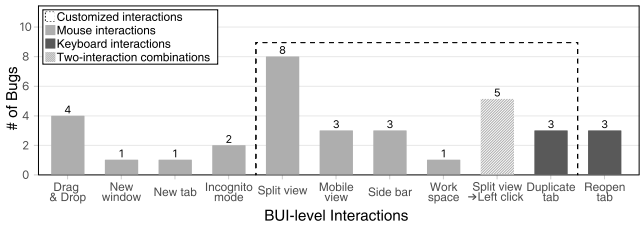

Figure 10: Number of policy enforcement bugs found per BUI-level interaction.

these two issues but missed all other bugs, highlighting the importance of considering BUI-level interactions (§5.5).

Our oracle and DIFFBROWSER$_{interaction}$ both found 25 common bugs, which account for a large portion of the identified bugs. However, within a fixed time budget, our oracle detects bugs substantially faster. To quantify this difference, we now evaluate the performance of BUIZZ and DIFFBROWSER$_{interaction}$.

Performance. We design a vendor-oriented testing scenario [57] to measure the bug-finding time of our oracle and DIFFBROWSER$_{interaction}$. In particular, we assume that Google uses each oracle to test . We measure the time required by each oracle to discover the eight bugs in (#1–5, #15, #16, and #24) under the same pages and interaction scenarios.

To ensure a fair comparison, we also use the same computational environment (§5.1). In particular, our oracle supports single-browser testing, so we ran in parallel on 12 machines. In contrast, DIFFBROWSER$_{interaction}$ requires cross-browser comparison across six browsers, so we used two machines per browser (12 machines in total). In addition, we used the same test pages in the same order for both runs to eliminate any bias due to input ordering.

Figure 9 shows the cumulative number of bugs discovered over time for the pre/post-interaction oracle and DIFFBROWSER$_{interaction}$. DIFFBROWSER$_{interaction}$ found six bugs over 85 hours of execution (it did not detect #16 and #24 for the reasons discussed above), whereas the pre/post-interaction oracle found all eight bugs within 14 hours, which is about 6× faster than DIFFBROWSER$_{interaction}$. For 50% of the identified bugs, our oracle required less than two hours, while DIFFBROWSER$_{interaction}$ required nine hours. In addition, DIFFBROWSER$_{interaction}$ continued running all six browsers after 14 hours without detecting any new bugs, consuming substantial computational resources.

These results show that, from a vendor's perspective of testing its own browser, our oracle is time-efficient because it enables single-browser testing. In contrast, cross-browser differential testing becomes resource-intensive and less efficient because it must run multiple browsers while also considering BUI-level interactions.

5.5 BUI-level User Interactions

Figure 10 presents the number of browser bugs triggered by each BUI-level interaction. In this analysis, we consider 34 of the 35 security bugs found by BUIZZ. We exclude one bug (Idx #24 in Table 4) that can be triggered by any interaction.

We observed that 23 of the 34 bugs (67.6%) are triggered by newly customized BUI-level interactions in , , , and . Interestingly, these are browser-specific bugs that do not exist in the upstream Chromium engine. These results underscore the importance of the COLLECTOR's comprehensive interaction collection, which captures customized interactions from browser manuals and context menus (§4.1).

Of the analyzed bugs, 82% (28/34) and 17.65% (6/34) are triggered by mouse and keyboard interactions, respectively. This skew arises because many customized user interactions are implemented through mouse-driven UI elements.

The "open link in split view" interaction triggers the largest share of bugs (38.2%). We attribute this tendency to split view's rendering model: a single tab hosts two pages simultaneously and enables cross-pane interactions, which makes policy enforcement harder to apply correctly. Since split view is becoming a common interaction pattern (e.g., Recently, 143 added split view support in Nov. 2025 [8]), this feature requires continuous and thorough testing.

Recall from §5.4 that previous approaches missed all 35 bugs found by BUIZZ because they do not consider BUI-level interactions. In contrast, BUIZZ detects these bugs by applying BUI-level interactions via OS-level simulation (§4.2).

5.6 Summary and Lessons

The bugs we study arise because browsers fail to consistently apply policy enforcement to interactions that occur outside the document's scope (i.e., browser user interface). We attribute this oversight to three factors: (1) lack of BUI-level testing, (2) over-reliance on upstream engine security, and (3) lack of specification guidance.

Lack of BUI-level testing. Across browser vendors, a common response to our bug reports and known bug reports (§C) was that they had overlooked policy enforcement for the reported BUI-level interaction. This calls for systematic BUI-level enforcement testing. However, such testing remains difficult because existing browser testing pipelines rely on Web-

Driver [58]. WebDriver operates through the document/page-rendering context and has restricted access to browser processes responsible for the BUI; therefore, it is not designed to exercise BUI-level interactions.

Because BUIzz simulates BUI-level interactions at the OS level (§4.2), it can be smoothly integrated into existing testing pipelines. We encourage browser vendors to incorporate our framework into their development cycle to verify that their fixes are complete and correct.

Over-reliance on upstream engine. Prior studies showed most enforcement bugs are largely confined to the upstream browser engine [65, 72, 86, 102]. However, our results show 67.6% of bugs originate from vendor-added BUI features rather than the upstream engine. We found all customized BUI features (Table 1) trigger policy enforcement bugs.

We argue that this pattern reflects an over-reliance on the upstream engine: vendors add or modify BUI features while assuming that upstream security guarantees will also hold for their customizations, and thus do not perform security testing for these changes as rigorously as upstream engine developers do. This indicates that downstream browsers need thorough security testing for newly customized features.

Lack of specification guidance. Existing specifications describe "navigation" at a high level, but they rarely state explicit requirements for policy enforcement under common BUI-driven navigations (e.g., open link in a new tab or reopen a closed tab) [12, 13, 35, 43, 45, 49]. We believe that clearer documentation of these requirements would reduce cases where vendors overlook these interactions or interpret enforcement behavior inconsistently across BUI-level actions.

6 Limitations and Discussion

OS dependency. BUIzz is OS-dependent because it relies on OS-level interaction simulation (§4). Our current implementation and evaluation are limited to Windows. To validate the cross-platform generalizability of the found bugs, we performed the following checks: (1) the set of navigation-related interactions is semantically consistent across operating systems; (2) all bugs identified by BUIzz were reproducible on Linux and macOS; and (3) manually applying the interactions on Linux and macOS to 50 randomly sampled pages per policy that showed no bugs on Windows did not reveal any additional bugs.

Nevertheless, the same interaction may be realized differently across OSes, so BUIzz may miss OS-specific bugs. This limitation can be addressed by porting BUIzz to other platforms. For example, BUIzz could leverage AppleScript [39] for macOS and pyatspi [1] for Linux to enable OS-level interaction simulation.

Search space. Despite using the same test pages as prior studies [72, 102] and WPT [57], existing approaches missed all 35 bugs identified by BUIzz. These results show that expanding the search space requires not only diversifying test HTML documents and policies, but also incorporating BUI-level interactions.

On the other hand, the two prior research works on CSP-XSS mitigation [102] and the SameSite cookie [72] provide a richer set of test pages (Table 2), which allowed us to uncover more BUI-level bugs than for other policies. This suggests that expanding test pages can synergize with BUI-level testing.

Interaction collection. We acknowledge that our interaction collection is not necessarily complete. As a result, if there exist unknown BUI-level interactions, BUIzz may miss policy enforcement bugs triggered by those interactions, producing false negatives. However, among 60 reported browser bugs since 2015 that are triggered by BUI-level interactions (§C), including the 38 bugs we found, BUIzz covers all interactions required to trigger them. This suggests that our interaction collection mechanism based on browser manuals and context menus is robust. Note that we used a broad set of keywords (i.e., "open", "load", "navigate", and "duplicate") to conservatively filter navigation-related interactions. After manually reviewing the full interaction set, we found no additional navigation-related interactions.

User involvement. To trigger a bug, a victim user should interact with the browser to navigate the page. In our setting, exploiting a policy enforcement bug additionally requires the victim to trigger a specific user interaction that activates the buggy interaction-based navigation. We assume an attacker-controlled website can serve as a stepping stone and entice the victim to perform the required interaction through convincing content and UI cues.

Local environment. BUIzz serves test pages through a local HTTP server with `/etc/hosts` redirection and mkcert-generated certificates [29] added to the system root store. Such a local setup may differ from production environments in ways that could affect enforcement behavior. However, we believe that our local setup is appropriate for most of the evaluated policies, as they are primarily enforced at the origin level. For HSTS, we used domains outside the preload list, which enabled BUIzz to identify one HSTS bug.

Bug oracle. The pre/post-interaction oracle can produce FPs and FNs. Regarding FPs, the oracle may flag a pre/post-interaction inconsistency even though the post-interaction behavior matches the specification-defined behavior (§5.2). Regarding FNs, the oracle may miss a bug when the same enforcement bug occurs both on the initial visit and after applying interactions (§5.4). To address these, we propose an LLM-based, specification-aware oracle as future work.

Bug grouping reliability. We identified 38 distinct bugs by analyzing 133 groups. To assess grouping reliability, we examined the risks of under-merging and over-merging. We observed 72 under-merged groups caused by shared upstream engine bugs. After accounting for each browser's upstream engine, these groups were straightforward to deduplicate, resulting in 38 distinct bug groups.

To assess potential over-merging, we manually analyzed

10 sampled pages from each group and examined whether the observed inconsistencies mapped to different root causes. We found no evidence that distinct bugs had been incorrectly merged. To further validate our grouping decisions, for the 14 fixed bugs, we reran all 3,049 pages in the corresponding groups on the patched browsers and confirmed that none of them triggered the previously reported bugs. Overall, these results suggest that our cross-referenced grouping poses a low risk of distorting the final bug count.

Interaction coverage. BUIzz focuses on testing navigation-related interactions and may therefore miss bugs triggered by non-navigation interactions. It also limits interaction sequences to two steps, meaning bugs that require three or more steps may not be detected. Extending BUIzz to incorporate non-navigation interactions or support longer interaction sequences is an interesting direction for future work, as it would require addressing both the expanded search space and the increased complexity of interaction simulation.

Extending to other bug classes. Our approach is not limited to testing the enforcement of header-based security policies. It can also be extended to identify other classes of browser bugs, such as memory-safety bugs and universal XSS vulnerabilities. Indeed, among the known bugs we studied (§C), six are memory-safety issues and three are UXSS issues. To identify these bugs, one can reuse test pages provided by existing tools (e.g., FreeDom [103] or FuzzOrigin [78]) and run BUIzz on those pages to simulate BUI-level interactions. We believe that browser research on BUI-level interactions is still at an early stage, and that follow-up work can build on our analysis to uncover other types of browser bugs.

7 Related Work

Differential testing. Prior work has proposed leveraging differential testing as a bug oracle to identify various classes of bugs, including policy enforcement bugs [65, 72, 86, 102], rendering bugs [92, 93, 106], and certificate validation bugs [63, 67]. The key insight is that different implementations are expected to exhibit consistent behavior under the same specification; therefore, any inconsistency indicates a bug.

Wi *et al.* [102] proposed DiffCSP, which identifies CSP enforcement bugs involving JS execution via cross-browser differential testing. Franken *et al.* [72] analyzed bypasses of third-party request and cookie policies using a wide range of test pages that issue cross-site requests in various forms. Rautenstrauch *et al.* [86] evaluated policy enforcement bugs caused by syntactically invalid and broken header parsing across different browsers. Calzavara *et al.* [65] proposed a formal framework for analyzing inconsistencies in framing control and developed a policy analyzer to assess the state of clickjacking mitigation.

Unlike prior work that relies on comparing different implementations, our pre/post-interaction oracle observes differences in enforcement behavior between the initial page visit and the visit after applying BUI-level interactions, enabling bug detection within a single implementation.

Policy adoption. Another line of recent research focuses on measuring trends in the adoption of security policies in the wild [64, 88, 90, 94, 95, 100]. Many of these works demonstrate that developers often misconfigure security headers, resulting in inadequate protection. Calzavara *et al.* [64] showed that CSP policies are seldom updated to mitigate insecure practices. Roth *et al.* [89] analyzed inconsistencies in security policy configurations caused by varying client characteristics, including browsers, devices, and language settings.

These studies generally assume correct browser enforcement. In contrast, our work examines policy enforcement bugs within browsers, showing that even correctly specified policies can be bypassed.

Browser security. Several works discover memory-safety bugs in browser engines [73, 75, 76, 81, 84, 98]. CodeAlchemist [75] uses semantics-aware assembly. Die [84] leverages aspect-preserving mutations to discover bugs in JS engines. Fuzzilli [73] leverages a custom intermediate representation to generate syntactically and semantically valid JS code.

There has been a surge of research studying web security policies provided by browsers [61, 62, 65, 70, 72, 80, 86, 96, 102, 105]. Kim *et al.* [78] proposed FuzzOrigin, which detects UXSS bugs in browsers via origin fuzzing. Shou *et al.* [91] presented CorbFuzz to check the enforcement of Cross-Origin Read Blocking (CORB) policy. Franken *et al.* [71] studied the lifecycle of browser security policy bugs. Bernardo *et al.* [61] formalized nine web invariants and evaluated them against the WPT test suites to find security flaws in client-side security mechanisms. Different from them, our work considers simulating BUI-level interactions to identify browser bugs. Our work is orthogonal to these approaches and can be combined with them to uncover additional bugs.

8 Conclusion

We present BUIzz, the first approach that systematically identifies policy enforcement bugs triggered by BUI-level interactions. BUIzz collects a comprehensive set of BUI-level interactions from browser manuals, context menus, and known bugs. It then simulates these interactions at the OS level to exercise realistic browser UI behaviors. We also propose a novel pre/post-interaction oracle that checks that policy enforcement remains consistent before and after applying BUI-level interactions. Across six browsers, BUIzz found 35 security bugs and three functional bugs, demonstrating its effectiveness in identifying policy enforcement bugs triggered by BUI-level interactions.

Acknowledgments

We would like to thank the anonymous reviewers for their concrete feedback. This work was supported by Innovative Human Resource Development for Local Intellectualization program through the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (IITP-2026-RS-2022-00156361 and RS-2024-00337414) and National Research Foundation of Korea (NRF) (RS-2025-00561150 and RS-2026-25497375).

Ethical Considerations

We conducted all experiments in local environments without any external network access. All domains used in the experiments were locally generated virtual domains, ensuring that our testing had no effect on real-world external systems. We did not involve real users, real accounts, or personal data in any part of the study.

Responsible disclosure. BUIZZ found 35 security bugs across six browsers. Recognizing the criticality of these findings and the widespread use of browsers, we responsibly disclosed all bugs to the corresponding vendors in a timely manner. At the time of writing, all affected vendors had at least 90 days to address the reported issues.

All vendors responded within one week and engaged cooperatively in our disclosure process. Whenever vendors responded, they acknowledged the issues as security bugs. For the 14 patched bugs, the average time from report to fix was 54 days. We verified that these issues no longer reproduce on the patched browser versions. In particular, we reran all 3,049 pages in the corresponding test groups on the patched browsers and confirmed that none triggered the previously reported bugs.

Stakeholder #1: browser vendors. We provided browser vendors with a 90-day disclosure window before public release, giving them sufficient time to patch the reported bugs. Before reporting, we manually validated the issues to confirm that they were genuine, reproducible bugs rather than false positives. When reporting the bugs, we also provided PoC pages that reliably trigger each bug, avoiding burdening vendors with low-quality reports.

Stakeholder #2: website operators and end users. Website operators and end users are the primary stakeholders in this setting because they cannot directly mitigate browser-side enforcement bugs on their own. Accordingly, coordinated private disclosure to the appropriate browser vendors and upstream maintainers was the most effective way to minimize potential harm by enabling timely patching.

Stakeholder #3: malicious actors. We acknowledge the potential risks of releasing BUIZZ, as it could be misused by malicious actors. Nevertheless, we argue that security through obscurity is not a sustainable approach to ensuring the security of browsers. By releasing BUIZZ, we aim to support vendors and researchers in hardening security policy enforcement and ultimately reduce the attack surface available to adversaries.

Open Science

To foster transparency and facilitate further research, we have made our implementation of BUIZZ in public: https://github.com/WebSec-Lab/BUIzz or https://doi.org/10.5281/zenodo.20422485. The repository includes the source code, scripts, and documentation required to reproduce the key experiments presented in this paper.

References

[1] AppleScript Language Guide. https://developer.apple.com/library/archive/documentation/AppleScript/Conceptual/AppleScriptLangGuide/introduction/ASLR_intro.html.

[2] blob: Urls. https://developer.mozilla.org/en-US/docs/Web/URI/Reference/Schemes/blob.

[3] Blocking top-level navigations to data urls for firefox 59. https://blog.mozilla.org/security/2017/11/27/blocking-top-level-navigations-data-urls-firefox-59/.

[4] Browser user interface. https://grokipedia.com/page/browser_user_interface.

[5] Browsing context sandboxing and user indicated targets for links. https://github.com/whatwg/html/issues/1526.

[6] Bugzilla - the issue tracker for firefox and other mozilla products. https://bugzilla.mozilla.org/home.

[7] Chrome keyboard shortcuts. https://support.google.com/chrome/answer/157179.

[8] Chrome's New 'Split View' Is Now My Favorite Way to Use the Internet. https://lifehacker.com/tech/google-chromes-new-split-view.

[9] Chromium issue tracker-home. https://issues.chromium.org/home.

[10] Content security policy level 1. https://www.w3.org/TR/CSP1/.

[11] Content security policy level 2. https://www.w3.org/TR/CSP2/.

[12] Content security policy level 3. https://www.w3.org/TR/CSP3/.

[13] Cross-Origin-Opener-Policy. https://html.spec.whatwg.org/multipage/browsers.html#the-coop-headers.

[14] CSP inheriting to avoid bypasses. https://www.w3.org/TR/CSP3/#security-inherit-csp.

[15] data: Urls. https://developer.mozilla.org/en-US/docs/Web/URI/Reference/Schemes/data.

[16] Directly user-initiated requests. https://www.w3.org/TR/fetch-metadata/#directly-user-initiated.

[17] FireFox Bug 1882069: HSTS not working on iframes and other subresources. https://bugzilla.mozilla.org/show_bug.cgi?id=1882069.

[18] Firefox Bugzilla - Implement and ship Permissions-Policy header. https://bugzilla.mozilla.org/show_bug.cgi?id=1694922.

[19] Hackerone - brave software bugbount program. https://hackerone.com/brave?type=team.

[20] HSTS Preload List Submission. https://hstspreload.org/.

[21] HTML Living Standard - Sandboxing. https://html.spec.whatwg.org/multipage/browsers.html#sandboxing.

[22] HTML Living Standard - shared document creation infrastructure. https://html.spec.whatwg.org/#shared-document-creation-infrastructure.

[23] Http-redirect fetch. https://fetch.spec.whatwg.org/#http-redirect-fetch.

[24] Keyboard shortcuts. opera://settings/keyboardShortcuts?search=shortcut.

[25] Keyboard shortcuts - perform common firefox tasks quickly. https://support.mozilla.org/en-US/kb/keyboard-shortcuts-perform-firefox-tasks-quickly.

[26] Keyboard shortcuts in microsoft edge. https://support.microsoft.com/en-us/microsoft-edge/keyboard-shortcuts-in-microsoft-edge-50d3edab-30d9-c7e4-21ce-37fe2713cfad.

[27] link-type-noopener. https://html.spec.whatwg.org/multipage/links.html#link-type-noopener.

[28] Microsoft Edge. https://www.microsoft.com/en-us/edge/download/.

[29] mkcert. https://github.com/filosottile/mkcert.

[30] Mouse. whale://settings/mouse.

[31] Msrc researcher portal - report an issue. https://msrc.microsoft.com/report/vulnerability/new.

[32] Naver bugbounty - report vulnerability. https://bugbounty.naver.com/.

[33] Naver Whale. https://whale.naver.com/en/.

[34] Opera security team - report an issue. https://security.opera.com/en/report-security-issue/.

[35] Permissions policy. https://www.w3.org/TR/permissions-policy/.

[36] Permissions Policy - Browser Support. https://caniuse.com/permissions-policy.

[37] Permissions Policy W3C Working Draft - create a permissions policy for a navigable. https://www.w3.org/TR/permissions-policy#algo-create-for-navigable.

[38] Playwright. https://github.com/microsoft/playwright.

[39] Pyatspi. https://github.com/GNOME/pyatspi2.

[40] PyAutoGUI. https://pyautogui.readthedocs.io/en/latest/.

[41] PyWinAuto. https://github.com/pywinauto/pywinauto.

[42] Referer. https://httpwg.org/specs/rfc9110.html#rfc.section.10.1.3.

[43] Referrer Policy. https://www.w3.org/TR/referrer-policy/.

[44] RFC 6797: HTTP Strict Transport Security (HSTS). https://datatracker.ietf.org/doc/html/rfc6797.

[45] RFC6265: Same-site Cookies. https://datatracker.ietf.org/doc/html/draft-west-first-party-cookies-07.

[46] RFC6265: Same-site Cookies - Top-level Navigations. https://datatracker.ietf.org/doc/html/draft-west-first-party-cookies-07#section-5.2.

[47] "same-site" and "cross-site" requests. https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-rfc6265bis-05#section-5.2.

[48] sanboxed iframe by middle click/ctrl(shift)+click/ctrl(shift)+enter. https://bugzilla.mozilla.org/show_bug.cgi?id=1102224.

[49] sandbox. https://www.w3.org/TR/CSP3/#directive-sandbox.

[50] Sandbox escape: bypass allow-popups-to-escape-sandbox. https://issues.chromium.org/issues/40057525.

[51] The sec-fetch-site http request header. https://www.w3.org/TR/fetch-metadata/#sec-fetch-site-header.

[52] Security: Samesite cookie bypass via background-fetch. https://issues.chromium.org/issues/40057062.

[53] Security: Security: Csp does not propagate to blob: Uris. https://issues.chromium.org/issues/40095900.

[54] Selenium. https://www.selenium.dev/.

[55] Use mouse shortcuts to perform common tasks in firefox. https://support.mozilla.org/en-US/kb/mouse-shortcuts-perform-common-tasks.

[56] W3c tag observations on private browsing modes. https://www.w3.org/2001/tag/doc/private-browsing-modes.

[57] web-platform-tests. https://web-platform-tests.org/.

[58] WebDriver W3C Working Draft. https://www.w3.org/TR/webdriver2/.

[59] What keyboard shortcuts can i use in brave? https://support.brave.app/hc/en-us/articles/360032272171-What-keyboard-shortcuts-can-I-use-in-Brave.

[60] What keyboard shortcuts can i use in brave? - mouse. https://support.brave.app/hc/en-us/articles/360032272171-What-keyboard-shortcuts-can-I-use-in-Brave.

[61] Pedro Bernardo, Lorenzo Veronese, Valentino Dalla Valle, Stefano Calzavara, Marco Squarcina, Pedro Adão, and Matteo Maffei. Web platform threats: Automated detection of web security issues with WPT. In *Proceedings of the USENIX Security Symposium*, pages 757–774, 2024.

[62] Fraser Brown, Deian Stefan, and Dawson Engler. Sys: A Static/Symbolic tool for finding good bugs in good (browser) code. In *Proceedings of the USENIX Security Symposium*, pages 199–216, 2020.

[63] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 114–129, 2014.

[64] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. Content security problems? evaluating the effectiveness of content security policy in the wild. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1365–1375, 2016.

[65] Stefano Calzavara, Sebastian Roth, Alvise Rabitti, Michael Backes, and Ben Stock. A tale of two headers: A formal analysis of inconsistent Click-Jacking protection on the web. In *Proceedings of the USENIX Security Symposium*, pages 683–697, 2020.

[66] Pinji Chen, Jianjun Chen, Mingming Zhang, Qi Wang, Yiming Zhang, Mingwei Xu, and Haixin Duan. Cross-origin web attacks via HTTP/2 server push and signed HTTP exchange. In *Proceedings of the Network and Distributed System Security Symposium*, 2025.

[67] Yuting Chen and Zhendong Su. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 793–804, 2015.

[68] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. NTFuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 677–693, 2021.

[69] Philippe De Ryck, Nick Nikiforakis, Lieven Desmet, and Wouter Joosen. Tabshots: Client-side detection of tabnabbing attacks. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, pages 447–456, 2013.

[70] Jan Drescher, David Klein, and Martin Johns. Are your sites truly isolated? automatically detecting logic bugs in site isolation implementations. In *Proceedings of the Network and Distributed System Security Symposium*, 2026.

[71] Gertjan Franken, Tom Van Goethem, Lieven Desmet, and Wouter Joosen. A bug's life: analyzing the lifecycle and mitigation process of content security policy

bugs. In *Proceedings of the USENIX Security Symposium*, pages 3673–3690, 2023.

[72] Gertjan Franken, Tom Van Goethem, and Wouter Joosen. Who Left Open the Cookie Jar? a comprehensive evaluation of Third-Party cookie policies. In *Proceedings of the USENIX Security Symposium*, pages 151–168, 2018.

[73] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. FUZZILLI: Fuzzing for JavaScript JIT compiler vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, 2023.

[74] HyungSeok Han and Sang Kil Cha. IMF: Inferred model-based fuzzer. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 2345–2358, 2017.

[75] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines. In *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[76] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the USENIX Security Symposium*, pages 445–458, 2012.

[77] Soheil Khodayari and Giancarlo Pellegrino. The state of the SameSite: Studying the usage, effectiveness, and adequacy of samesite cookies. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 1590–1607, 2022.

[78] Sunwoo Kim, Young Min Kim, Jaewon Hur, Suhwan Song, Gwangmu Lee, and Byoungyoung Lee. FuzzOrigin: Detecting UXSS vulnerabilities in browsers through origin fuzzing. In *Proceedings of the USENIX Security Symposium*, pages 1008–1023, 2022.

[79] Young Min Kim and Byoungyoung Lee. Extending a hand to attackers: Browser privilege escalation attacks via extensions. In *Proceedings of the USENIX Security Symposium*, pages 7055–7071, 2023.

[80] Lukas Knittel, Christian Mainka, Marcus Niemietz, Dominik Trevor Noß, and Jörg Schwenk. Xsinator.com: From a formal model to the automatic evaluation of cross-site leaks in web browsers. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1771–1788, 2021.

[81] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. Montage: A neural network language model-guided JavaScript engine fuzzer. In *Proceedings of the USENIX Security Symposium*, pages 2613–2630, 2020.

[82] Tongxin Li, Xueqiang Wang, Mingming Zha, Kai Chen, XiaoFeng Wang, Luyi Xing, Xiaolong Bai, Nan Zhang, and Xinhui Han. Unleashing the walking dead: Understanding cross-app remote infections on mobile webviews. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 829–844, 2017.

[83] Kazuki Nomoto, Takuya Watanabe, Eitaro Shioji, Mitsuaki Akiyama, and Tatsuya Mori. Browser permission mechanisms demystified. In *Proceedings of the Network and Distributed System Security Symposium*, 2023.

[84] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing JavaScript engines with aspect-preserving mutation. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 1629–1642, 2020.

[85] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[86] Jannis Rautenstrauch, Trung Tin Nguyen, Karthik Ramakrishnan, and Ben Stock. Head (er) s Up! detecting security header inconsistencies in browsers. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 3057–3070, 2025.

[87] Jannis Rautenstrauch, Giancarlo Pellegrino, and Ben Stock. The leaky Web: Automated discovery of cross-site information leaks in browsers and the web. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 2744–2760, 2023.

[88] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. Complex security policy? a longitudinal analysis of deployed content security policies. In *Proceedings of the Network and Distributed System Security Symposium*, 2020.

[89] Sebastian Roth, Stefano Calzavara, Moritz Wilhelm, Alvise Rabitti, and Ben Stock. The security lottery: Measuring client-side web security inconsistencies. In *Proceedings of the USENIX Security Symposium*, 2022.

[90] Sebastian Roth, Lea Gröber, Michael Backes, Katharina Krombholz, and Ben Stock. 12 angry developers-a qualitative study on developers' struggles with csp. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 3085–3103, 2021.

[91] Chaofan Shou, Ismet Burak Kadron, Qi Su, and Tevfik Bultan. CorbFuzz: Checking browser security policies with fuzzing. In *Proceedings of the International*

Conference on Automated Software Engineering, pages 215–226. IEEE, 2021.

[92] Suhwan Song, Jaewon Hur, Sunwoo Kim, Philip Rogers, and Byoungyoung Lee. R2z2: Detecting rendering regressions in web browsers through differential fuzz testing. In *Proceedings of the International Conference on Software Engineering*, pages 1818–1829, 2022.

[93] Suhwan Song and Byoungyoung Lee. Metamong: Detecting render-update bugs in web browsers through fuzzing. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 1075–1087, 2023.

[94] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with Content Security Policy. In *Proceedings of the international conference on World wide web*, pages 921–930, 2010.

[95] Marius Steffens, Marius Musch, Martin Johns, and Ben Stock. Who's hosting the block party? studying third-party blockage of CSP and SRI. In *Proceedings of the Network and Distributed System Security Symposium*, 2021.

[96] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. How the web tangled itself: Uncovering the history of client-side web (in) security. In *Proceedings of the USENIX Security Symposium*, pages 971–987, 2017.

[97] Avinash Sudhodanan, Soheil Khodayari, and Juan Caballero. Cross-Origin State Inference (COSI) Attacks: Leaking web site states through XS-Leaks. In *Proceedings of the Network and Distributed System Security Symposium*, 2020.

[98] Liam Wachter, Julian Gremminger, Christian Wressnegger, Mathias Payer, and Flavio Toffalini. DUMPLING: Fine-grained differential JavaScript engine fuzzing. In *Proceedings of the Network and Distributed System Security Symposium*, 2025.

[99] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 579–594, 2017.

[100] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. CSP is dead, long live CSP! on the insecurity of whitelists and the future of content security policy. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1376–1387, 2016.

[101] Harald Weinreich, Hartmut Obendorf, Eelco Herder, and Matthias Mayer. Not quite the average: An empirical study of web use. *ACM Transactions on the Web*, 2(1):1–31, 2008.

[102] Seongil Wi, Sooel Son, Trung Tin Nguyen, Jiwhan Kim, and Ben Stock. DiffCSP: Finding browser bugs in Content Security Policy enforcement through differential testing. In *Proceedings of the Network and Distributed System Security Symposium*, 2023.

[103] Wen Xu, Soyeon Park, and Taesoo Kim. FreeDom: Engineering a state-of-the-art DOM fuzzer. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 971–986, 2020.

[104] Guangliang Yang, Jeff Huang, and Guofei Gu. Iframes/Popups are dangerous in mobile WebView: Studying and mitigating differential context vulnerabilities. In *Proceedings of the USENIX Security Symposium*, pages 977–994, 2019.

[105] Chijin Zhou, Quan Zhang, Lihua Guo, Mingzhe Wang, Yu Jiang, Qing Liao, Zhiyong Wu, Shanshan Li, and Bin Gu. Towards better semantics exploration for browser fuzzing. *Proceedings of the ACM Programming Languages*, 7:604–631, 2023.

[106] Chijin Zhou, Quan Zhang, Bingzhou Qian, and Yu Jiang. Janus: Detecting rendering bugs in web browsers via visual delta consistency. In *Proceedings of the International Conference on Software Engineering*, pages 2702–2713, 2025.

A Details of Three Functional Bugs

Strict cookie omission. omitted `SameSite=Strict` cookies during top-level navigations in HTTP 301–308 redirects. Since redirections [23] in user-initiated top-level navigations have a `Sec-Fetch-Site` [47,51] value of `none`, cookies with the `SameSite=Strict` attribute should be sent.

Referer header omission. In and , when navigation is triggered with "Open link in incognito window," the Referer header is omitted. Intuitively, omitting the Referer header in this context appears reasonable from a privacy perspective. However, according to the current specifications [42,56], the Referer header is expected to be included when transitioning to incognito mode.

B Reducing the DiffCSP Test Pages

We describe how we reduce the DiffCSP [102] test pages from 25,880 HTML files and 1,006 CSP headers to 500 HTML files and 21 CSP headers. As a first step to reduce headers, we excluded directives or schemes (e.g., `filesystem:`) that are

Policy	# of Pages	# of Scenarios (Single Interaction)							# of Scenarios (Two-Interaction Combination)						
								Execution Time							Execution Time
CSP (XSS mitigation)	10,500	69,720	69,720	83,664	83,664	73,164	93,996	83h	10,500	10,500	10,500	10,500	10,500	10,500	11h
CSP (sandbox)	5	35	35	42	42	36	48	3m	126	126	203	181	146	207	10m
SameSite Cookie	207	1,065	1,065	1,278	1,278	1,072	1,299	1h 15m	5,151	5,151	7,494	7,417	5,221	7,508	7h
PP	500	3,320	3,320	3,984	3,984	3,484	4,476	4h	12,501	12,501	19,805	18,001	14,141	20,133	17h
COOP	16	120	120	144	144	128	168	9m	400	400	665	577	481	681	35m
HSTS	207	1,065	1,065	1,278	1,278	1,072	1,299	1h 15m	5,151	5,151	7,494	7,417	5,221	7,508	7h
CSP (framing control)	9	45	45	54	54	45	54	3m	225	225	324	324	225	324	17m
XFO	9	45	45	54	54	45	54	3m	225	225	324	324	225	324	17m
RP	1,656	8,520	8,520	10,224	10,224	8,576	10,392	10h	41,208	41,208	59,952	59,952	41,768	60,064	53h

Table 5: Corpus and scenario.

Idx	OS	CPU	Main memory
1	Windows 10 Pro	AMD Ryzen 7 1800X (8 cores)	32 GB
2	Windows 10 Pro	Intel(R) i7-6700 CPU (4 cores)	32 GB
3	Windows 10 Pro	Intel(R) i7-6700 CPU (4 cores)	16 GB
4	Windows 10 Pro	Intel(R) i7-6700 CPU (4 cores)	16 GB
5	Windows 11 Education	Intel(R) i7-6700 CPU (4 cores)	16 GB
6	Windows 11 Education	Intel(R) i7-7700 CPU (4 cores)	16 GB
7	Windows 11 Education	Intel(R) i7-7700 CPU (4 cores)	8 GB
8	Windows 11 Pro	Intel(R) N95 (4 cores)	16 GB
9	Windows 11 Pro	Intel(R) N95 (4 cores)	16 GB
10	Windows 11 Pro	Intel(R) N95 (4 cores)	16 GB
11	Windows 11 Pro	Intel(R) N95 (4 cores)	16 GB
12	Windows 11 Pro	Intel(R) N95 (4 cores)	16 GB

Table 6: Experimental machines.

Idx	CVE number / Issue number	Interaction
1	CVE-2024-7978 / (Chrome) 40060358	Drag and drop
2	- / (Chrome) 40057769	Drag and drop
3	CVE-2021-30544 / (Chrome) 40055982	Back and forward
4	- / (Chrome) 40092751	Open link in new tab
5	- / (Chrome) 41166597	middle click
6	- / (Chrome) 40076093	Drag and drop
7	- / (Chrome) 40055439	Drag and drop
8	CVE-2021-30593 / (Chrome) 40055890	Drag and drop
9	CVE-2021-30525 / (Chrome) 40055514	Drag and drop
10	- / (Chrome) 368562236	Drag and drop
11	CVE-2024-1675 / (Chrome) 41486208	Drag and drop
12	- / (Chrome) 40057823	Drag and drop
13	- / (Chrome) 40053054	Open link in new tab & Reload
14	- / (Bugzilla) 1102224	Open link in new tab
15	- / (Bugzilla) 1037766	Open link in new tab
16	- / (Bugzilla) 1559128	Open link in new tab
17	- / (Bugzilla) 1330364	Open link in new tab
18	CVE-2023-25741 / (Bugzilla) 1813376	Drag and drop
19	CVE-2019-11698 / (Bugzilla) 1543191	Drag and drop
20	- / (Bugzilla) 1297186	Drag and drop
21	- / (Bugzilla) 1812611	Drag and drop
22	- / (Bugzilla) 1316104	Open link in new tab

Table 7: Known bugs related to BUI-level user interactions.

Interaction	Precondition	Prerequisite Actions
Open link in current tab	Need a link	-
Open link in new background tab	Need a link	-
Open link in new tab and move to page	Need a link	-
Open link in a new window	Need a link	-
Open link in new tab	Need a link	-
Open link in an incognito window	Need a link	-
Open link in a split window	Need a link	-
Open link in a mobile view	Need a link	-
Open link in a side bar	Need a link	-
Open link in a workspace	Need a link	-
Reopen closed tab	-	1. If there is a link in the document: - 🖱 the link 2. Close a tab via Ctrl+W
Navigate backward	-	1. If there is a link in the document: - 🖱 the link 2. Visit to other website
Navigate forward	-	1. If there is no link in the document and the Back button is disabled: - Visit to other website - Visit to testing document 2. If there is a link in the document: - 🖱 the link 3. Navigate backward via Alt+←
Reload page	-	If there is a link in the document: - 🖱 the link
Reload without cache	-	If there is a link in the document: - 🖱 the link
Duplicate tab	-	If there is a link in the document: - 🖱 the link
Force open link in the current tab	Need a link	-

Table 8: Navigation-related interactions.

no longer supported. Subsequently, we removed headers that do not semantically affect security enforcement. For example, corner cases (e.g., `non-ASCII URL`) were excluded, as these headers are intended to test parser robustness.

From a CSP testing perspective, values with the same semantic meaning were normalized into a single form. As an example, scheme (e.g., `https://` vs. `http://`) and domain variations were unified, as a bypass observed in one case generally applies to other cases as well.

To reduce HTML test cases, we consolidated test instances that evaluate identical CSP enforcement aspects. For example, since `blob:` and `about:blank` URIs share the same semantic implications regarding CSP inheritance, we unified them into a single representative case. Also XSS payloads were unified into a single representative tag (e.g., `<script>`). Finally, we deduplicate headers and HTML corpus based on hash.

Although we empirically reduced the number of test pages, we believe that this reduction is strategic in that it enables efficient evaluation of the impact of BUI-level testing without compromising the DiffCSP search space. Importantly, the reduction was performed in a reasonable manner to ensure coverage of all 37 bugs previously identified by DiffCSP.

C Known Bugs

We collect 22 known CVEs and issues related to user interactions from Chromium and Bugzilla. Table 7 shows the list of known CVEs and issues that are related to user interactions.