DiffCSP: Finding Browser Bugs in Content Security Policy Enforcement through Differential Testing

Seongil Wi*, Trung Tin Nguyen†‡, Jihwan Kim*, Ben Stock†, Sooel Son*

*School of Computing, KAIST
†CISPA Helmholtz Center for Information Security
‡Computer Science Graduate School, Saarland University

***Abstract*—The Content Security Policy (CSP) is one of the *de facto* security mechanisms that mitigate web threats. Many websites have been deploying CSPs mainly to mitigate cross-site scripting (XSS) attacks by instructing client browsers to constrain JavaScript (JS) execution. However, a browser bug in CSP enforcement enables an adversary to bypass a deployed CSP, posing a security threat. As the CSP specification evolves, CSP becomes more complicated in supporting an increasing number of directives, which brings additional complexity to implementing correct enforcement behaviors. Unfortunately, the finding of CSP enforcement bugs in a systematic way has been largely understudied.**

In this paper, we propose DiffCSP, the first differential testing framework to find CSP enforcement bugs involving JS execution. DiffCSP generates CSPs and a comprehensive set of HTML instances that exhibit all known ways of executing JS snippets. DiffCSP then executes each HTML instance for each generated policy across different browsers, thereby collecting inconsistent execution results. To analyze a large volume of the execution results, we leverage a decision tree and identify common causes of the observed inconsistencies. We demonstrate the efficacy of DiffCSP by finding 29 security bugs and eight functional bugs. We also show that three bugs are due to unclear descriptions of the CSP specification. We further identify the common root causes of CSP enforcement bugs, such as incorrect CSP inheritance and hash handling. We confirm the risky trend of client browsers deriving completely different interpretations from the same CSPs, which raises security concerns. Our study demonstrates the effectiveness of DiffCSP for identifying CSP enforcement bugs, and our findings have contributed to patching 12 security bugs in major browsers, including Chrome and Safari.

I. Introduction

Content Security Policy (CSP) [4, 5, 6] is a protection mechanism that has become a *de facto* measure of security mitigation. A web developer declares a CSP for their website in the web response header or the response body via a meta tag. This instructs the browser to honor the CSP and govern the execution and inclusion of various web resources, such as scripts or images. The CSP specification has evolved into the level 3 specification (CSP3) [6] since its adoption by major browsers, including Chrome, Firefox, Edge, and Safari.

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.24200
www.ndss-symposium.org

```
1  XSS attack payload:
2    http://[Target URL]/PoC.html#javascript:alert('XSS')
3  CSP: script-src-elem 'sha256-aHbTR...';
4  Target website:
5  <script>
6    let hash = window.location.hash.slice(1);
7    window.location.href = hash;
8  </script>
```

Fig. 1: *An HTML instance that triggers a bug in Chrome; the adversary bypasses CSP enforcement by exploiting this bug.*

CSP has been increasingly deployed on the Web. Most CSPs are meant to mitigate the impact of cross-site scripting (XSS) attacks by limiting the sources from which an adversary can draw their injected scripts. Weichselbaum *et al.* demonstrated that 1,680,000 Internet hosts deployed a CSP, and 86% of the unique CSPs that they crawled were designed to mitigate XSS threats [70]. Roth *et al.* also showed that by 2019, 1,233 out of the 10,000 popular sites listed in the Internet Archive had deployed a CSP [58].

Due to the nature of a client-side security mechanism, CSP depends on browsers to honor a given policy and correctly govern the inclusion of web resources according to the policy. In particular, CSP plays a crucial role in mitigating XSS threats by governing the execution of JS snippets based on their source URL, hash value, or attached nonce. Therefore, a browser bug in CSP enforcement regarding JS execution causes actual behaviors to deviate from the CSP specification and the expectations of site operators. Accordingly, such discrepancies open security holes for an XSS attacker to bypass CSPs and inject adversarial JS snippets.

Figure 1 shows a CSP that defines `script-src-elem` [35] to have the hash value of `sha256-aHbTR...`. The expected behavior is that a browser would only execute the inline JavaScript (JS) snippet in a script tag or `javascript:`, the hash value of which is `sha256-aHbTR...` (i.e., the JS between Lines 6–7). Hence, any injected inline scripts will be blocked according to the defined policy. However, a bug in Chrome and Edge that we discovered in this paper allows an XSS attacker to inject arbitrary JS via `javascript:` navigation regardless of the hash value in the `script-src-elem` directive.

Despite the severity of CSP enforcement bugs, to the best of our knowledge, no previous studies have addressed the systematic identification of browser bugs in CSP enforcement regarding JS execution. In this paper, we hence raise the research question: *How can we identify browser bugs in CSP*

enforcement regarding JS execution?

Contributions. We propose DiffCSP, the first testing framework designed to find CSP enforcement bugs in governing JS execution. To identify bugs in browsers without analyzing their source code, we choose to generate inputs that are highly likely to exhibit erroneous CSP enforcement. Specifically, DiffCSP generates HTML instances that exhibit all the known forms of executing JS snippets and diverse CSPs. DiffCSP then checks whether a generated CSP is violated across these generated HTML instances due to an inherent browser bug by checking whether the embedded JS snippet in each HTML instance is executed.

Devising a testing framework that finds CSP enforcement bugs by generating inputs entails three technical challenges: (1) the testing framework should generate comprehensive inputs (i.e., CSPs and HTMLs) that trigger inherent browser bugs; (2) it should identify unexpected execution results for each generated input that may lead to finding browser bugs; and (3) it should help an analyzer examine a large number of generated inputs that exhibit unexpected behaviors.

To address the first challenge, we define a data-driven HTML grammar by referencing known CSP bugs [10], known XSS attack payloads [8], an HTML security cheat sheet [24], and the ECMAScript specification [17]. That is, this HTML grammar is designed to derive adversarial HTML instances that exhibit all known forms of embedding JS snippets. For the second challenge, we conduct differential testing on generated inputs on three desktop and five mobile browsers and identify a set of inputs that cause inconsistent execution results. That is, we use the inconsistent execution results for each generated input as a bug oracle.

Notably, different browser vendors may support different levels of the CSP specification (e.g., until recently, Safari did not support the `strict-dynamic` keyword introduced in 2016), and we generate many HTML instances that exhibit diverse ways of executing JS snippets. Therefore, DiffCSP reports a large number of inputs that cause behavioral differences. Subsequently, it becomes difficult to analyze all test inputs causing these inconsistencies. To overcome this challenge, we first vectorize the execution result for each generated HTML instance and label this HTML instance as either consistent or inconsistent based on its execution result. Using these vectors and their labels, we compute a decision tree to ease the identification of causes that result in a large number of observed inconsistencies. We then identify all paths leading to inconsistent execution results in the decision tree. For each collected path, we manually analyze the conditions that appear in the path and HTML instances that correspond to this decision path. Therefore, DiffCSP helps us to identify causes while avoiding an analysis of each HTML instance that causes an inconsistency, thus addressing the third challenge.

Using DiffCSP, we found CSP enforcement bugs in three desktop and five mobile browsers. For the desktop browsers, we found 37 bugs in Chrome, Firefox, and Safari. These browser vendors patched 23 bugs; eight of them were patched solely due to our bug reports, and four were patched in response to addressing reports from ours and other bug reporters. For four bugs, we are awaiting the responses from the browser vendors. Out of the 37 identified bugs, 29 bugs undermine security, allowing the attacker to bypass CSPs and inject executable JS snippets. Furthermore, we identified that three bugs stem from the vague or missing descriptions in the CSP standard [6].

```
1  script-src 'self';
2  default-src http://a.com;
```

Fig. 2: *An example of content restriction using CSP.*

In summary, this paper makes the following contributions:

1) We design and implement DiffCSP, the first testing framework that identifies browser bugs in CSP enforcement regarding JS execution.
2) We propose conducting differential testing across multiple browsers by generating a diverse set of test inputs that exhibit unexpected behaviors regarding CSP enforcement.
3) To systematically analyze the observed inconsistencies, we leverage decision trees to pinpoint the root causes for erroneous CSP enforcement.
4) We identify 37 browser bugs, including 29 security bugs. We find that three bugs are due to unspecified rules in the CSP specification, thus recommending revision of the specification. Chromium, Safari, and Firefox browser vendors have patched 12 of them, providing an award for the bugs found in Chrome, Safari, and Firefox with a bounty of 4,000 USD.
5) To support open science and reproducible research, we will release DiffCSP at https://github.com/WSP-LAB/DiffCSP.

II. Background

A. Content Security Policy

A content security policy (CSP) refers to an HTTP response header or a policy defined via a meta element, which enables client software (e.g., browsers) to honor a defined policy. By design, it is a browser's responsibility to enforce a defined CSP while rendering a webpage, governing the inclusion of web resources based on their domain sources or hash values.

Stamm *et al.* [65] originally proposed CSP, with the original goal of mitigating cross-site scripting (XSS) attacks. Browser vendors have adopted CSP and developed its standard specification. The first specification was finalized in 2015 and CSP level 2 was already published in 2016. Since then, CSP3 has remained a working draft, however, is the de facto standard that should be implemented in browsers [6].

A CSP consists of directives, each of which defines a set of values. Figure 2 shows two directives: `default-src` and `script-src`. In the presence of `script-src`, this directive governs the inclusion of scripts. In this case, this means that the page can only execute external scripts hosted on `http://a.com`, but it implicitly disallows inline scripts, inline event handlers, and eval. All other resources (e.g., images or fonts) must be loaded from the page's origin, given the `default-src` fallback directive. This directive controls all resources for which a more specific directive (e.g., `image-src` or `font-src`) is not present in the policy.

The straightforward nature of defining allowed domains simplified the implementation of CSP enforcement in browsers in the early stage of CSP. However, CSP level 3 now supports

25 directives, nine source keywords, and multiple fallback mechanisms, which bring complexity to implementing correct behaviors for all possible CSPs. Considering that CSP level 1 has supported only 10 directives with four source keywords, the usage of CSP has become more complicated due to its supporting various fine-grained security policies [58].

It is natural for browser vendors to experience difficulties in implementing correct behaviors for all possible CSPs. For example, external scripts can be governed by `script-src-elem`, which falls back to `script-src`, which in turn falls back to `default-src`. The `script-src-elem` directive, however, was only recently added to the working draft and may not be implemented by all browsers. Hence, the number of possible combinations of CSPs and the differing levels of support across browsers make it challenging to conduct systematic testing of CSPs.

Consequently, a bug in CSP enforcement poses a serious security threat. Consider a site operator who deploys a CSP that forbids inline scripts without matching hashes in the CSP. A browser bug may enable the adversary to inject a new inline script, which obsoletes the need to deploy CSPs. In this paper, we assume an XSS adversary who abuses such CSP enforcement bugs. The adversary's goal is to conduct successful XSS attacks by exploiting an XSS vulnerability in a target website and bypassing the CSP emplaced on this website.

III. Motivation and Technical Challenges

Recent work [58] has highlighted that XSS mitigation is on par with framing control and TLS enforcement. However, this original goal remains the most complex and important aspect of CSP [43, 70]. It is thus important for browser vendors to ensure that JS execution adheres to a given CSP.

Unfortunately, there have been no previous studies that *systematically* identify browser bugs in enforcing CSP that govern JS execution. We tackle this bug identification problem by generating testing inputs and checking whether these inputs cause unexpected behaviors in enforcing a given CSP. That is, we generate a set of inputs that trigger inherent browser bugs that cause erroneous CSP enforcement.

Finding CSP enforcement bugs via input generation entails three technical challenges: (1) generating comprehensive testing inputs that trigger CSP enforcement bugs, (2) identifying unexpected browser behaviors for the generated inputs that leads to the identification of bugs, and (3) analyzing the causes of the bugs triggered.

Generating inputs. Each browser vendor has already implemented their own regression tests that check for CSP enforcement. It is thus vital to generate a comprehensive set of testing inputs that trigger inherent bugs that the existing regression testing set does not cover. In our study, we observed that the regression tests in WebKit and Chromium missed several test instances that involve page navigation in a child iframe or window instance (§V-D). We also found that these tests were applied as-is to the browser's regression test set, without any combination or mutation.

To address this challenge, we generate diverse CSPs and adversarial inputs of HTML files that exhibit all known ways of embedding executable JS snippets. Specifically, we reference previous browser attacks [10, 11], known XSS attack payloads [8], and the ECMAScript specification [17] in deriving an HTML grammar. We then use this grammar to generate 25,880 HTML instances. Note that DiffCSP conducts testing on 25,880 HTML instances against 1,006 CSPs, whereas the Chromium team has implemented their own web platform tests consisting of 98 HTML files to vet the correctness of CSP enforcement involving JS execution [39].

Implementing bug oracles. Given a test input (i.e., a generated CSP and an HTML instance embedding a JS snippet), the identification of erroneous behaviors should precede the decision regarding whether the testing CSP and HTML instance trigger a CSP enforcement bug. However, given a large number of testing inputs, the manual identification of their correct behaviors by referencing the CSP specification is infeasible. Note that this manual identification involves generating a test case, mapping this test case to the corresponding CSP specifications, extracting the correct behavior, and converting this behavior into a testing oracle, a process that needs to be done just once per case. Considering that we generate 25,880 HTML instances that embed JS snippets and test them against 1,006 different CSPs, the manual identification of correct behaviors for these cases is not scalable.

For this, we conduct differential testing that executes each generated HTML instance with a testing CSP across different browsers. We then identify inconsistent execution results, which elucidate erroneous behaviors to be checked. We assume that the test inputs that cause different results between browsers, even by at least one browser, are highly likely due to inherent bugs in CSP enforcement.

Identifying root causes. Considering that we generate a large number of HTML instances and CSPs for differential testing, it is natural to generate a large number of inputs that cause inconsistent execution results. Thus, the manual analysis of each inconsistent behavior to identify its root causes would be intractable. Accordingly, we propose a new way of analyzing the inconsistent execution results observed. We compute a decision tree for the execution results and analyze this decision tree to identify common factors behind the observed inconsistencies.

IV. Design

A. Workflow

Figure 3 illustrates the overall architecture of DiffCSP, which consists of three components: Generator, Executor, and Analyzer. At a high level, these components work together to conduct three steps: (1) the Generator generates two sets of test inputs: a set of CSPs and another set of HTML instances, each of which contains a JS snippet; (2) the Executor coordinates the testing of browsers to execute each HTML instance for each test CSP and collects JS execution results and their inconsistencies across the testing browsers; and, after the Executor runs all test inputs in all testing browsers, (3) the Analyzer computes a decision tree using the observed execution results. We use this computed decision tree to group testing inputs that share common conditions for observed inconsistencies and then conduct a post-mortem analysis for the testing inputs sampled from these groups.

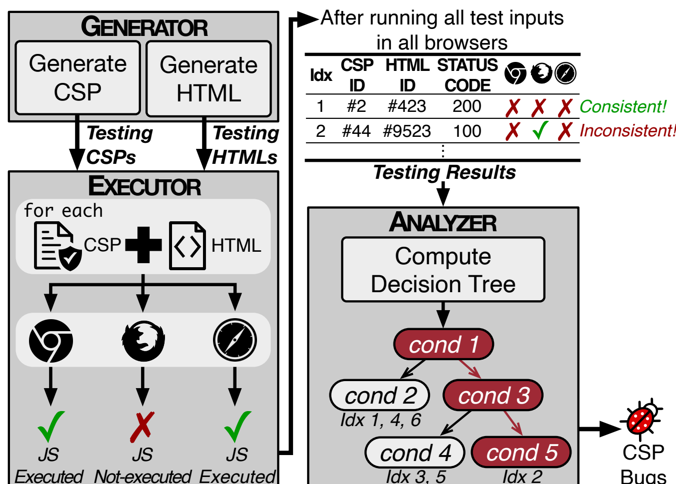


***Fig. 3:** DiffCSP architecture.*

<table>
<tr><td>Directive</td><td colspan="2"><code>default-src</code>, <code>script-src</code>, <code>script-src-elem</code>, <code>script-src-attr</code></td></tr>
<tr><td></td><td>Corner case</td><td>Capitalized directive (<code>Default-src</code>)</td></tr>
<tr><td rowspan="6">Value</td><td>Keyword</td><td><code>none</code> | <code>unsafe-inline</code> | <code>unsafe-eval</code> | <code>self</code> | <code>strict-dynamic</code> | <code>unsafe-hashes</code></td></tr>
<tr><td>Host-source</td><td>Self URL (<code>http://127.0.0.1:8000</code>) | Allowed URL (<code>http://127.0.0.1:8080</code>) | <code>*</code></td></tr>
<tr><td>Schemes</td><td><code>data: blob: http: https:</code></td></tr>
<tr><td>Nonce-source</td><td><code>nonce-123</code></td></tr>
<tr><td>Hash-source</td><td><code>sha256-[HASH]</code></td></tr>
<tr><td>Corner case</td><td>Non-ASCII URL (<code>http://üüü.de</code>) | empty | Capitalized keyword (‘<code>None</code>’) | Capitalized host (<code>Http://127.0.0.1:8000</code>) | Capitalized nonce (‘<code>Nonce-123</code>’)</td></tr>
</table>

***TABLE I:** Elements used to generate testing CSP.*

B. GENERATOR

The GENERATOR generates a set of CSPs and another set of HTML instances, which become test inputs for differential testing. Our goal here is to generate diverse types of test CSPs and HTML instances regarding JS execution. In particular, we compose all known ways of executing simple JS snippets in HTML as well as diverse CSPs governing JS execution. Additionally, the GENERATOR generates a set of status codes that will be mapped to a subset of the generated HTML instances to test the effect of status codes on CSP enforcement.

CSP generation. Table I shows all directives and their values that the GENERATOR leverages to generate CSPs. We design this component to generate policies involving `default-src` [13], `script-src` [33], `script-src-elem` [35], and `script-src-attr` [34] directives, which are all the directives that affect JS execution according to the CSP3 working draft [6].

For these directives, we define 12 representative values grouped into five categories (i.e., keyword, host-source, schemes, nonce-source, and hash-source) by referring to the policy specification [36]. Since the `[HASH]` value in the hash-source group should be different for each tested JS, the value is not fixed when generating a policy; it is changed later before the EXECUTOR executes the test HTML instance (§IV-C).

To define a host-source value, we assume three web servers: (1) a self-origin server (`http://127.0.0.1:8000`), (2) an allowed-origin server (`http://127.0.0.1:8080`), and (3) a blocked-origin server (`http://127.0.0.1:8082`). The self-origin server is designed for testing CSPs containing `self`. The allowed-origin and blocked-origin servers are for fetching allowed and blocked cross-origin JS snippets, respectively.

We also define corner cases for directives and values, which are likely to be provided by site operators. For example, according to the CSP specification, the URLs in directive values should contain only ASCII characters [36][1]. However, it is reasonable that one unfamiliar with the specification may write URLs with non-ASCII characters [26] in a CSP. With the expectation that such exceptional cases render behavioral differences across browsers, we prepare a non-ASCII URL value. We also prepare four more directive values: empty and the capitalized forms of keyword, web server address, and nonce-source [14, 16, 29, 31].

The GENERATOR enumerates all possible combinations of the directives and values above. However, we restrict the GENERATOR to generating CSPs having at most two directive values for each generated directive. Through this process, the GENERATOR generates 1,006 CSPs. For each generated policy, it assigns a unique identifier, `CSP_ID`, which the EXECUTOR and ANALYZER use to associate behavior with a specific CSP.

HTML generation. Table II summarizes an HTML grammar that DiffCSP leverages to generate test inputs of HTML instances. To design this grammar, we referenced known CSP enforcement bugs [10, 11], XSS attack payloads [8], an HTML security cheat sheet [24], and the ECMAScript specification [17]. We emphasize that DiffCSP does not simply reuse the referenced HTML and JS examples; rather, we build grammar out of these referenced files. The GENERATOR is thus able to compose all possible combinations of HTML instances that embed simple JS snippets [62, 69].

The GENERATOR enumerates a set of test HTML instances by traversing the defined HTML grammar in a depth-first manner. From a root grammar rule with the non-terminal symbol of `[HTML]`, the GENERATOR composes an HTML instance by applying an applicable grammar rule and replacing each non-terminal symbol.

Specifically, the GENERATOR starts by composing an HTML instance by applying a derivation rule with the non-terminal symbol of `[HTML]`. From the current HTML instance in composition, it identifies non-terminal symbols (i.e., `[JS_REQ_URL]`, `[JS_INLINE_URL]`, `[PAGE]`, `[SCHEME]`, `[JS]`, and `[HTML]`) and replaces each non-terminal symbol by applying an applicable grammar rule. The GENERATOR repeats this process until either of the following termination conditions is satisfied: (1) the GENERATOR encounters an element in `[JS_REQ_URL]` or `[JS_INLINE_URL]`, or (2) the number of applied grammar rules exceeds five. The first termination condition denotes that the GENERATOR successfully composes an HTML instance that executes an inline script or imports a JS script. The second condition means that the GENERATOR discards the generated HTML instance when it becomes too complicated.

[1] Non-ASCII domain names cannot be entered into a CSP, but instead must be Punycode- encoded.

Symbol	HTML/JS Derivation Rule	Feature	Value
[JS_REQ_URL]	http://127.0.0.1:8000/self.js?csp_id=[CSP_ID]&html_id=[HTML_ID]&status_code=[CODE] http://127.0.0.1:8080/allowed.js?csp_id=[CSP_ID]&html_id=[HTML_ID]&status_code=[CODE] http://127.0.0.1:8082/blocked.js?csp_id=[CSP_ID]&html_id=[HTML_ID]&status_code=[CODE]	JS Execution Method	Self-JS: 0 Allowed-JS: 1 Blocked-JS: 2
[JS_INLINE_URL]	http://127.0.0.1:8000/executed?csp_id=[CSP_ID]&html_id=[HTML_ID]&status_code=[CODE]		Inline JS: 3
[PAGE]	self.html | about:blank | self.txt	-	-
[SCHEME]	javascript:'[HTML]' | javascript:[JS] | data:text/html,[HTML] | data:application/js,[JS] | ...	-	-
[JS]	**Category #1: executing inline JS** fetch('[JS_INLINE_URL]')	-	-
	Category #2: evaluating string eval('[JS]') | new Function('[JS]') | setTimeout('[JS]',0)	Included or not	0-1
	Category #3: dynamically fetching JS var o=document.createElement('script'); o.src='[JS_REQ_URL]';document.body.appendChild(o)	Included or not	0-1
	Category #4: redirecting to scheme location='[SCHEME]' | window.open('[SCHEME]') | var o=document.createElement('iframe');o.src='[SCHEME]';document.body.appendChild(o) | ...	Included or not	0-1
	Category #5: expanding document document.body.innerHTML+='[HTML]' | document.write('[HTML]')	Included or not	0-1
	Category #6: writing to opened document w=window.open('[PAGE]');w.document.write('[HTML]')	Included or not	0-1
[HTML]	**Category #1: executing inline JS in script tag** <script>[JS]</script>	Included or not	0-1
	Category #2: fetching JS in script tag <script src=[JS_REQ_URL]></script>	Included or not	0-1
	Category #3: redirecting to scheme <script>x.click()</script> | <object data='[SCHEME]'></object> | <iframe src='[SCHEME]'></iframe> | ...	Included or not	0-1
	Category #4: executing inline JS in event handler <iframe onload='[JS]'></iframe> | <audio src/onerror='[JS]'></audio> | <details ontoggle='[JS]' open>test</details> | ...	Included or not	0-1
	Category #5: writing to frame <iframe srcdoc='[HTML]'></iframe> | <iframe id=x src=[PAGE]> </iframe><script>x.onload=_=>x.contentDocument.write('[HTML]')</script> | ...	Included or not	0-1
	Category #6: changing location of iframe <iframe id=x src=[PAGE]></iframe> <script>x.onload=_=>x.src='[SCHEME]'</script> | ...	Included or not	0-1
	Category #7: evaluating string via frame's function <iframe id=x src=[PAGE]></iframe> <script>x.onload=_=>x.contentWindow.eval('[JS]')</script> | ...	Included or not	0-1
	Category #8: expanding document <svg xmlns=http://www.w3.org/2000/svg>[HTML]</svg> | <template id=x>[HTML]</template> <script>document.body.appendChild(x.content.cloneNode(true))</script> | ...	Included or not	0-1

TABLE II: *Grammar rule to generate testing HTML.*

Idx	Derived HTML
1	<script>**[JS]**</script>
2	<script>fetch('**[JS_INLINE_URL]**')</script>
3	<script>fetch("http://127.0.0.1:8000/executed ?csp_id=[CSP_ID]&html_id=**[HTML_ID]** &status_code=[CODE]")</script>
4	<script>fetch("http://127.0.0.1:8000/executed ?csp_id=[CSP_ID]&html_id=0 &status_code=[CODE]")</script> → ***HTML #0 generated!***
...	...
923	<script nonce=123>**[JS]**</script>
...	...

TABLE III: *Example of derivation process to generate HTMLs.*

Table III shows an example of the composition process. The GENERATOR starts by applying the first rule of [HTML] in Table II and then continues derivation for the scanned non-terminal symbol, i.e., [JS].

For each generated and saved HTML file, the GENERATOR assigns a unique identifier to the [HTML_ID] of the generated page (e.g., Idx 4 in Table III). Note that [CSP_ID] and [CODE] have not yet been assigned at this stage. Later, the EXECUTOR changes these values (§IV-C).

Regarding the testing of nonce-source values in test CSPs, the GENERATOR is required to generate nonce-protected scripts. Therefore, if a <script> tag is found while scanning an element, the GENERATOR derives two pages, a page having <script> and another page having <script nonce=123> (e.g., Idx 1 and 923 in Table III). Then, the GENERATOR respectively performs the composition of the testing code.

In total, the GENERATOR generates 25,880 HTML instances. To boost the testing efficiency of DiffCSP, it merges the generated HTML instances into a number of test units, each of which becomes an HTML file. Here, we set the number of HTML instances in each group to 80 in order not to cause instability in the execution pipeline. However, when a function related to page redirection, such as window.open or location.href, is included in an HTML instance, the inclusion of other HTML instances in the same file will stop the testing of HTML instances that appear after this HTML instance involving page redirection. Thus, each HTML instance involving page redirection is excluded from this optimization process; such an instance is stored in a separate file.

With this approach, the GENERATOR generated 11,663 HTML files from 25,880 HTML instances. Each generated HTML file is then stored in the web root directory of the self-origin server that EXECUTOR accesses for differential testing.

Status code. Specific HTTP status codes may cause security flaws, such as changing the execution context of JS snippets [27, 28] or disabling certain HTTP headers [3]. Motivated by these prior bugs, we define representative status codes to systematically study if and how the status codes affect CSP enforcement. In particular, the GENERATOR leverages five representative status codes (i.e., 100, 200, 300, 400, and 500) when generating test instances. For testing efficiency, we map all status codes to a limited set of the generated HTML instances (§IV-C).

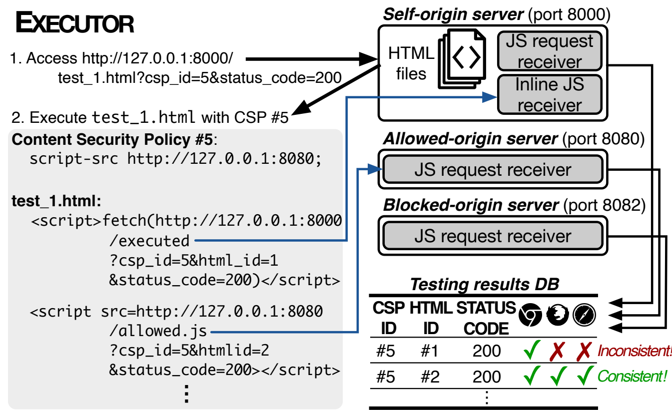


Fig. 4: EXECUTOR *workflow.*

C. EXECUTOR

Given a set of test CSPs and HTML files, the EXECUTOR enumerates all combination pairs. For each pair of a CSP and an HTML file, it executes the HTML file that has the CSP across different browsers and then checks whether each JS snippet embedded in this HTML file is executed.

For testing efficiency, all HTML files are executed with the 200 response code. Additionally, a limited set of HTML files is executed with all the defined status codes; we sample eight HTML files from 11,663 HTML files (i.e., 640 HTML instances) containing at least one of each `[HTML]` category in Table II.

Testing browsers. For differential testing, we leverage three desktop and five mobile browsers, compiling a set of eight browsers. We observe that rendering engines govern JS execution enforcement behaviors. Therefore, we test three rendering engines (i.e., Chromium, Firefox, and WebKit) using executions results of three desktop browsers and two rendering engines (i.e., Chromium and Gecko) using execution results of five mobile browsers (§V-A).

Workflow. Figure 4 illustrates the workflow of the EXECUTOR. Assume that the EXECUTOR tests `test_1.html` with a CSP corresponding to the `CSP_ID` of 5 and with the status code 200. The EXECUTOR instructs a testing browser to send the request to `test_1.html` with the `csp_id` parameter of `CSP_ID` 5 and the `status_code` parameter of `CODE` 200. When the testing server receives this request, it sends a response by setting the CSP corresponding to the `CSP_ID` in the HTTP header and setting the status code corresponding to `CODE`, along with sending the body embedded with `test_1.html`. In this process, `[CSP_ID]` and `[CODE]` are replaced with given parameters. When `[HASH]` exists in the test CSP, the string is replaced with the concatenation of all hashes that correspond to the inline snippets in the HTML file.

The main goal of the execution step is to determine whether or not a script was executed in light of the specified CSP. If the testing browser executes an inline script, this will invoke the `fetch` function. Therefore, the testing server will receive a request to the `executed` path of the self-origin server (the fourth row in Table II). If a testing browser executes a script fetched from cross-origin servers, the browser sends a JS fetching request to the specified URL (see the first to third rows in Table II). Thus, the testing server is able to check whether each embedded script (an inline script or a JS script fetching from a cross-origin) is executed for the given testing triad of an HTML file, a CSP, and a status code by checking the `csp_id`, `html_id`, and `status_code` parameters in an incoming request. The EXECUTOR then stores these execution results in a database for each testing browser.

DiffCSP determines that a tested triad renders inconsistent behaviors when at least one testing browser reports a different JS execution result. For instance, if Chrome allows JS execution for a specific testing triad while Firefox and Safari block JS execution, we consider that the HTML file in this triad potentially triggers a browser bug. Naturally, this approach does not tell which browsers have a bug [42, 47, 48], considering that the majority or minority browser vendors may implement enforcement incorrectly. However, those triads become promising data points for the next analysis step, which involves a manual post-mortem analysis aided by decision trees (§IV-D).

D. ANALYZER

The EXECUTOR reports behavioral inconsistencies for a large number of the generated triads, each of which consists of a CSP, an HTML instance, and a status code. Note that DifFCSP generates 4M and 3.5M triads, for which the execution results may respectively differ by the three desktop and five mobile browsers (§V-B). Manually identifying the root causes of all execution inconsistencies is infeasible; it would demand the analysis of each generated triad that reports an inconsistent execution result.

To address this challenge, we propose to leverage a decision tree. A decision tree is designed to derive a set of human-readable conditions that lead to a classification decision. We leverage this capability to compute a set of conditions that lead to inconsistent execution results. Each leaf node that indicates a decision corresponds to a set of training instances; the decision tree derives the same path and conditions for those training instances.

Using this capability, we leverage a computed decision tree to group generated triads that share the same decision paths. Then, from each group that contains an inconsistent execution result, we pick one triad and analyze this triad with the conditions that appear in the corresponding path. That is, instead of analyzing all generated triads in each group, we analyze one example for each group. In other words, we leverage a decision tree to derive an interpretation of observed execution results and use this interpretation to identify representational inputs for each decision path.

Figure 5 illustrates how we build and leverage a decision tree. The ANALYZER converts each CSP, HTML, and status code triad into a vector. It then uses the generated dataset to train a decision tree.

Dataset. The dimension of a feature vector is 37; 22 features, 14 features, and one feature represent a CSP, an HTML instance, and a status code, respectively. For a generated CSP, each element in Table I is encoded into a feature vector. It consists of 22 binary features, and each feature value represents whether the corresponding feature is present. 14 features are from a generated HTML instance. Each feature and

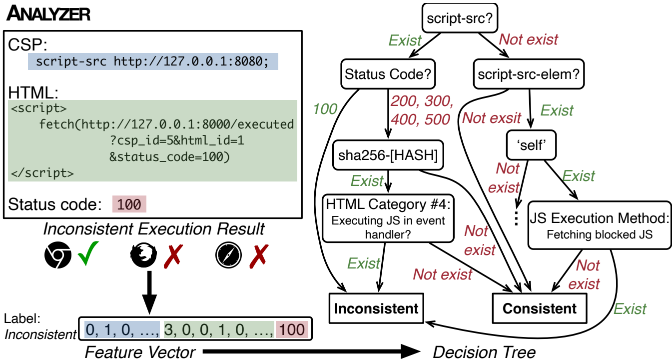


Fig. 5: ANALYZER *workflow.*

Idx	Desktop	Version			
1	Chromium	99			
2	Firefox	95			
3	WebKit (Safari)	15.4			
Idx	**Mobile**	**Version**	**Package Name**	**Device**	**# of Downloads**
4	Chromium	101	com.android.chrome	Nexus	+10,000,000,000
5	Chromium	100	com.opera.browser	Pixel 3	+100,000,000
6	Gecko	100	org.mozilla.firefox	Nexus	+100,000,000
7	Chromium	102	com.brave.browser	Pixel	+50,000,000
8	Gecko	96	org.torproject.torbrowser	Pixel 3	+10,000,000

TABLE IV: *Browsers under testing.*

its value are listed in the third and fourth columns in Table II, respectively. For the status code, we use the status code itself as a feature. For each vector representing a generated triad, we use one of two class labels: inconsistent (positive) or consistent (negative).

Decision tree. Given a computed decision tree, we manually enumerate all paths from a root to the leaf nodes that represent inconsistent execution results. Each path consists of tree nodes, each of which contains a condition that involves features.

Figure 5 shows a decision tree with three paths leading to inconsistent execution results. The first path leading to inconsistent results (*script-src* ∧ *status code 100*) represents a bug in which Chromium-based browsers ignore a CSP when the response comes with the status code 100. This step enables an auditor to identify factors causing the inconsistent results and analyze an instance sampled from the training instances that share this path. This step enables to avoid the analysis of the 276K instances that belong to that path.

The second path (*script-src* ∧ *status code 200–500* ∧ *sha256-[HASH]* ∧ *Executing JS in event handler*) represents a bug where Firefox enables `unsafe-hashes` by default in the `script-src` directive. This inconsistency occurs when a hash-source value exists in the directive value and an event handler in a generated HTML instance.

DiffCSP computes two decision trees: one for the desktop browsers and the other for the mobile browsers. Note that the purpose of computing the decision trees is not to classify a given instance, but to facilitate the analysis of identifying causes for inconsistent execution results. Thus, we do not split the training instances for the testing dataset.

DiffCSP reported 19M execution results for all generated triads. Leveraging a decision tree computed from all these instances generates a complex decision tree, the depth of which is beyond 30, making it intractable to manually analyze 61,479 paths leading to inconsistencies. We conduct the following two optimization methods so that the tree can explain the root causes for the observed inconsistencies while limiting the number of paths to be inspected: (1) we randomly sample consistent training instances to match their number to the number of inconsistent training instances, and (2) we set the decision tree depth to 10 for desktop browsers and mobile browsers, respectively.

We empirically confirmed that when the tree depth is 10, the recall of the tree is kept high (i.e., 0.95 for desktop browsers and 0.92 for mobile browsers) while the number of paths to be inspected decreases (§V-E). In each computed decision tree, we investigated every path that corresponds to at least one generated triad causing an inconsistent execution result. For each path, we sample one triad of an HTML, a CSP, and a status code, corresponding to this path and analyze this triad along with the conditions that appeared in the path. As a result, we examined 525 and 581 paths for desktop and mobile browsers, respectively.

V. EVALUATION

We evaluate the efficacy of DiffCSP to find browser bugs in enforcing CSPs that govern JS execution using three desktop and five mobile browsers (§V-B). We then describe the root causes of the identified bugs, categorize these causes into 10 groups, and explain the security implications of the bugs (§V-C). We also analyze the degree to which the computed decision tree helps find the causes of the identified bugs (§V-E). Finally, we demonstrate the performance of DiffCSP in finding CSP bugs (§V-F).

A. Experimental Setup

Browsers. We ran a series of experiments on the eight browsers listed in Table IV. For desktop browsers, we selected the three most popular browsing engines: Chromium, Firefox, and WebKit [23]. We have observed that all testing results for Chrome and Edge were identical because they use the same rendering engine as Chromium. Since the rendering engine governs JS execution according to a given CSP [2, 10], we used the testing results of Chromium to represent those of Chrome and Edge. We also used WebKit to test the rendering engine of Safari. Note that new Safari features have been implemented in WebKit before their releases. In addition, 11 of the 18 (61%) WebKit bugs that we discovered were present in the latest version of Safari 15.4.

For mobile browsers, we selected five popular browsers. Specifically, we first selected 20 mobile browsers in the order of their downloads from Google Play Store as of May 2022 and then excluded nine ARM-based apps, the APKs of which do not support Android API 30 (i.e., Android 11) x86 system images, which is required for our testing environment using an emulator. We also excluded six mobile browsers using Android WebView. Those execution results depend on the version of the WebView service installed on a mobile device, which may significantly vary across user devices. When they use the latest WebView service, the execution results become identical to those of the Chromium mobile browser.

We observed that the test results for mobile browsers using the same rendering engine were identical. Therefore, we grouped the execution results of the mobile browsers under testing into two groups: Chromium (Chrome, Opera, and Brave) and Gecko (Firefox and Tor).

In summary, we tested three rendering engines (i.e., Chromium, Firefox, and WebKit) for the desktop browsers as well as two engines (i.e., Chromium and Gecko) that corresponds to five mobile browsers.

Environment. For desktop browser testing, we conducted experiments on two machines running x86_64 Ubuntu with 88 CPUs and 378 GB of main memory. To automatically visit each testing page from each browser, we compiled a test script using Playwright 1.18.0 [30]. For each HTML file, we set the timeout of execution to three seconds. When this page involves page redirection and has a single HTML instance, we set the timeout to one second.

For mobile browser testing, we performed our testing on four servers running x86_64 Debian with 192 CPUs and 1.5 TB of main memory. To automate the testing process, we relied on Android Virtual Device (AVD) [7] and the Android Debug Bridge (ADB), which is a command-line tool that helps us communicate with a device [1]. For the testing devices in our emulators, we used a Galaxy Nexus with Android API 25 (i.e., for x86-based apps) and a Pixel 3 with Android API 30 (i.e., for ARM-based apps).

B. Bugs Found

Table V summarizes CSP enforcement bugs that we found using DiffCSP. We found a total of 37 browser bugs after analyzing 7.5M discrepancies that DiffCSP reported. Recall from §IV-D that we identified 525 and 581 paths leading to the inconsistent execution results in the desktop and mobile decision trees, respectively. Each path leading to an inconsistent result corresponds to a set of HTML instances, CSPs, and status codes. From such a set of test inputs, we selected one triad of an HTML file, a CSP, and a status code. We then analyzed this triad and identified the causes along with the conditions that appeared in the path.

Of the 37 browser bugs, we manually confirmed that 29 bugs imposed a security threat and eight were functional bugs. For each security bug, we analyzed its implications by questioning whether an adversary is able to exploit the bug and bypass a certain CSP, thus injecting an arbitrary JS script. We further confirmed whether attack payloads exploiting such a security bug were blocked by other browsers.

Of the 29 security bugs, 27 bugs (93%) enable the adversary to inject an executable JS, the execution of which should be blocked according to the CSP standard. The remaining two bugs entail relatively low-security implications. However, by exploiting these bugs, an adversary is capable of bypassing pre-request checks and sending a request to an arbitrary endpoint.

We clustered the 37 bugs into two groups: (1) bugs due to unclear/incorrect descriptions in the CSP specification and (2) implementation flaws that stem from vendors' mistakes in not properly following the specification. The third column of Table V categorizes the identified vulnerabilities into these two groups (▲ represents group (1) and the others belong to group (2)).

Bug disclosures. We reported all 27 security bugs resulting from vendor's mistakes to the three browser vendors (i.e., Chromium, Firefox, and WebKit, representing Safari). At the time of writing, 23 bugs have been patched by the vendors. Among them, 12 bugs were patched in response to our bug reports, and the other bugs were patched in response to reports from the browser vendors or users. Also, we are currently awaiting responses for four bugs from the vendors. For the reported bugs, the Chromium team awarded us a bug bounty of 4,000 USD.

C. Root Causes

Table V lists the 37 identified bugs and their causes. We further categorized these root causes into 10 groups, as shown in the table. The third column of the table depicts the conditions of the page, the CSP, or the HTTP status code that contributes to triggering the bugs. The fourth column shows the expected behavior (i.e., whether to execute a given JS testing code) according to the CSP specification [6] regarding the conditions in the third column. We manually extracted these expected behaviors from the specification to check the correctness of our findings. When the specification does not describe the expected behavior for the identified conditions, we mark it with ▲ and count the number of bugs to one.

The fifth to ninth columns indicate whether each browser exhibits the expected behavior. When the browser conducts the expected behavior as the specification describes, we mark it with a ✓, and an ✗ otherwise. N/A indicates a bug that still poses a security threat. However, we did not count it as a bug because the corresponding browsers did not support certain directives or directive values. For example, we mark inconsistency #7 as N/A for Firefox because it does not support the processing of the `script-src-elem` directive, even though the CSP3 standard demands its support. In the following, we list five root causes out of the 10 ones. We describe the remaining causes in Appendix IX-A.

Cause #1: Incorrect CSP inheritance. An embedded iframe or a newly opened new window loaded from a local scheme (e.g., `blob`, `data`, `javascript`, or `about`) should inherit the CSP of their parent document [12]. The goal here is to prevent the adversary from bypassing the parent's CSP by opening a child window or embedding a child frame that contains attack code under the adversary's control.

We found that Safari incorrectly inherits a parent CSP, allowing string-to-JS execution in child pages even when the parent CSP blocks `eval()`. The following snippet shows a test HTML instance that exhibits this bug.

```
1 CSP: script-src 'nonce-123';
2 <iframe id="x" src="about:blank"></iframe>
3 <script nonce=123>
4     let hash = window.location.hash.slice(1);
5     x.onload=_=>x.contentWindow.eval("'" + hash + "'");
6     x.contentWindow.location.reload();
7 </script>
```

Note that because `unsafe-eval` is not included in the CSP, a site operator would expect the string evaluation by `eval()` to be blocked. Also, since `nonce-123` is included in the CSP,

Category	Idx	Page / CSP / HTTP header condition	Expected Behavior (Manually Extracted)	Desktop: Chromium	Desktop: Firefox	Desktop: Safari	Mobile: Chromium	Mobile: Firefox	# of Bugs
		Cause #1: Incorrect CSP inheritance							
Security Bugs	1	Dynamically calling eval() from the about: frame (‘unsafe-eval’ is not specified)	eval() should be blocked	✓	✓	✗	✓	✓	1
	2	Dynamically changing frame.src to JS URL with embedded inline script	Inheritance should occur	✓	✓	✗	✓	✓	1
	3	Dynamically changing frame.src to data URL with embedded HTML	Inheritance should occur	✓	✓	✗	✓	✓	1
	4	Dynamically changing frame.src to blob URL with embedded HTML	Inheritance should occur	✓	✓	✗	✓	✓	1
	5	Dynamically changing frame.src to JS URL with embedded HTML	Inheritance should occur	✗	✓	✗	✗	✓	2
	6	Dynamically writing JS embedded HTML to static file	▲ (✓ indicates inheritance has occurred)	✗	✓	✗	✗	✓	1
		Cause #2: Incorrect hash handling							
	7	javascript:[JS] with script-src-elem [hash-source]	JS from JS URL should not be executed	✗	N/A	✓	✗	N/A	1
	8	script-src [hash-source]; script-src-elem ‘none’	Hashed script should not be executed	✓	N/A	✗	✓	N/A	1
	9	script-src [hash-source] ‘unsafe-hashes’; script-src-attr ‘none’	Hashed script should not be executed	✓	N/A	✗	✓	N/A	1
		Cause #3: Non-ignored directive values							
	10	default-src ‘strict-dynamic’ ‘unsafe-inline’	‘unsafe-inline’ should be ignored	✗	N/A	N/A	✗	N/A	1
	11	script-src ‘strict-dynamic’ [host-source]	[host-source] should be ignored	✓	✓	✗	✓	✓	1
		Cause #4: Non-supporting specific directives							
	12	The script-src-elem directive	Directive should be supported	✓	✗	✓	✓	✗	1
	13	The script-src-attr directive	Directive should be supported	✓	✗	✓	✓	✗	1
		Cause #5: Non-supporting specific directive values							
	14	The nonce-source in the default-src directive	Value should be supported	✓	†✗	✓	✓	†✗	1
	15	The hash-source in the default-src directive	Value should be supported	✓	✗	✓	✓	✗	1
	16	‘strict-dynamic’ in the default-src directive	Value should be supported	✓	✗	✗	✓	✗	2
		Cause #6: Auto-enabling directive values by default							
	17	Auto-enabled ‘unsafe-hashes’ in the script-src directive	Value should be disabled by default	✓	✗	✓	✓	✗	1
	18	Auto-enabled * in the script-src-elem directive	Value should be disabled by default	✓	N/A	✗	✓	N/A	1
		Cause #7: Auto-enabling directive values on specific conditions							
	19	Auto-enabled ‘unsafe-inline’ in the script-src-elem directive **Condition:** ‘strict-dynamic’ is specified in the script-src-elem directive	Value should be disabled by default	✓	N/A	‡✗	✓	N/A	1
	20	Auto-enabled ‘unsafe-inline’ in the script-src-elem directive **Condition:** The hash-source is specified in the script-src-attr directive	Value should be disabled by default	✓	N/A	✗	✓	N/A	1
	21	Auto-enabled ‘unsafe-inline’ in the script-src-elem directive **Condition:** The hash-source is specified in the script-src-elem directive	Value should be disabled by default	✓	N/A	✗	✓	N/A	1
	22	Auto-enabled ‘unsafe-inline’ in the script-src-attr directive **Condition:** The hash-source is specified in the script-src-elem directive	Value should be disabled by default	✓	N/A	✗	✓	N/A	1
	23	Auto-enabled ‘unsafe-inline’ in the script-src-attr directive **Condition:** The hash-source is specified in the script-src-attr directive	Value should be disabled by default	✓	N/A	✗	✓	N/A	1
		Cause #8: Non-supporting CSP for specific status code							
	24	100 status code in HTTP header	CSP should be enabled	✗	✓	✓	¶✓	✓	1
		Cause #9: Incorrect handling of malformed CSPs							
	25	Non-ASCII character within directive value	▲ (✓ indicates the directive is disabled)	✓	✗	✓	✓	✗	1
		Cause #10: Allowing out-going requests							
	26	Parser-inserted script (i.e., script tag) with ‘strict-dynamic’	Out-going JS request should be blocked	✓	✗	✓	✓	✗	1
	27	Parser-inserted script (i.e., written script tag) with ‘strict-dynamic’	Out-going JS request should be blocked	✗	✓	✓	✗	✓	1
Functional Bugs	28	Nested srcdoc with JS fetching (The URL of the JS is specified in the CSP)	JS from allowed URL should be executed	✓	✗	✓	✓	✗	1
	29	Nested data scheme with JS fetching (The URL of the JS is specified in the CSP)	JS from allowed URL should be allowed	✓	✗	✓	✓	✗	1
	30	Calling cloneNode() for inline script (Nonce-source for inline script specified in CSP)	Nonced inline script should be executed	✓	✗	✓	✓	✗	1
	31	Executing javascript:[JS] (hash for [JS] specified in CSP)	▲ (✓ indicates script is executed)	✗	✓	✗	✗	✓	1
	32	NoCSPiframe.contentWindow.eval() (‘unsafe-eval’ is not specified)	eval() should be allowed	✓	✓	✗	✓	✗	2
	33	JS fetching from NoCSPiframe.srcdoc with following CSP: default-src ‘none’; script-src ‘unsafe-inline’ [Allowed URL]	JS from allowed URL should be executed	✓	✗	✗	✓	✗	2
Total									37

† Only for inline scripts (e.g, <script nonce=123>[Inline script]</script>), not for JS fetching (e.g., <script src=[URL] nonce=123></script>).
‡ Only for inline scripts existing in script tag, not in the JS navigation.
¶ The bug was patched in Chromium 100.

TABLE V: *Experimental results with eight major browsers: Chromium, and Gecko (Firefox), WebKit (Safari).*

they would expect that only the nonce-protected JS will be allowed. However, Safari allows string-to-JS execution under certain page conditions, allowing an attacker to bypass the emplaced CSP and execute arbitrary scripts if the page is vulnerable to XSS attacks. Especially, we observed that calling eval() is allowed if it is called dynamically from a child window or iframe with the src property of about:blank. We reported this bug to the WebKit team, and the vendor fixed the bug.

We also observed that Safari did not conduct CSP inheritance when it involves javascript:, data:, or blob: navigation. Also, in Chromium, this kind of bug is triggered when the navigation to a URL involves the javascript: scheme. These bugs occur because the browsers do not check the CSP of a parent document when asynchronous navigation is involved (e.g., dynamically changing the src attribute of an existing iframe to a local scheme URL).

For example, consider the following test CSP and webpage that triggers inconsistency #3.

```
1  CSP: script-src 'nonce-123';
2  <iframe id="z" src="self.html"></iframe>
3  <script nonce=123>
4      z.addEventListener("load",() => { z.src =
5        "data:text/html,<script>alert(1)<script>";});
6  </script>
```

In this example, when the load event of the iframe is fired, the src attribute of the iframe is changed to the data URL that executes the JS snippet in Line 5. It is expected that this iframe will inherit the parent's CSP, thus blocking JS execution of this embedded HTML instance. However, the identified bug contributes to bypassing CSP enforcement, which allows the inline script in Line 5 to be executed. We emphasize that the defined CSP does not affect the executed script at all. Therefore, the injected script is capable of bypassing frame busting as well as TLS enforcement even if the CSP defines

those directives.

We have reported these bugs to the WebKit team and the vendor has fixed these bugs. We have also reported the identified bug to the Chromium team and are awaiting their response.

Regarding the CSP inheritance that involves static files, we observed that Chromium and Safari do not pass over the CSP of a parent window to its child window or iframe when they open static files. Note that site operators cannot define a CSP via meta tags in static files, such as `.txt`, `.js`, and `.ico`, due to the nature of these file formats. Site operators may set a CSP in an HTTP(S) response header when delivering these files. However, the current practice overlooks assigning CSPs for static files in real-world services [22, 37, 41].

An XSS attacker is able to exploit this bug to establish a same-origin document context for XSS attacks [47, 64]. Assume that a target website allows `unsafe-inline` in the script-src directive. Then, an XSS attack may open a window with the target website's favicon file path. The adversary then writes an attack script code in this window, which will be executed using the first-party origin without any CSP restrictions. One may argue that the `unsafe-inline` requirement in the target website does not necessitate the injection of an attack script in the new window. However, the XSS attacker is able to bypass other restrictions imposed by the CSP, such as TLS enforcement (`block-all--mixed-content` and `insecure-requests`), frame busting (`frame-ancestors`), fetch (`connect-src`), and invoking `eval()`. Furthermore, 56% of websites using CSPs have deployed `unsafe-inline` in their directives [58].

We reported these bugs to each vendor, and the Chromium vendor stated that, according to the CSP standard, the case of inheriting CSPs is limited to navigation by the local scheme, so this behavior is working as expected. However, they acknowledged that this bug has the security implications aforementioned, and this issue is under discussion.

Cause #2: Incorrect hash handling. As described in Figure 1, Chromium allowed the execution of arbitrary inline scripts present in `javascript:` even when their hash value does not match any hash values in the `script-src-elem` directive. If a website has an XSS vulnerability and blocks arbitrary JS execution by checking hash values in `script-src-elem`, the attacker can exploit this bug to trivially bypass hash checking and inject an executable script. We reported this bug to the respective vendor; however, this bug had already been patched internally in May 2022.

We also found incorrect directive fallback when handling a hash-source value in WebKit. Note that browsers should only use `script-src` as a fallback when both `script-src-elem` and `script-src-attr` are not explicitly set in a given CSP [21]. However, WebKit uses the presence of a hash-source in the `script-src` directive even when `script-src-elem` or `script-src-attr` exists in a CSP. This means that the interpretation of a CSP satisfying the conditions above significantly differs by browsers, which may permit non-allowed scripts to be executed. This bug had been patched in the latest version of WebKit at the time we discovered it.

Cause #3: Non-ignored directive values. According to the specification pertaining to `strict-dynamic`, host-source, and scheme-source expressions as well as `unsafe-inline` and `self` keyword-source should be ignored when `strict-dynamic` is specified in the `script-src` or `default-src` directive of a given CSP.

Unfortunately, we observed that Chromium did not ignore `unsafe-inline` even when `strict-dynamic` was specified in `default-src`. We also found that Safari did not ignore the host-source when `strict-dynamic` is specified in `script-src`.

Since neither Firefox nor Safari supports `strict-dynamic` in `default-src` (inconsistency #16), it may seem impossible to find inconsistency #10 through differential testing. However, the following testing page and CSP enabled the finding of this bug:

```
1 CSP: default-src 'unsafe-inline' 'strict-dynamic';
2 <script>
3     var o = document.createElement('script');
4     o.src = 'http://127.0.0.1:8000/test.js';
5     document.body.appendChild(o);
6 </script>
```

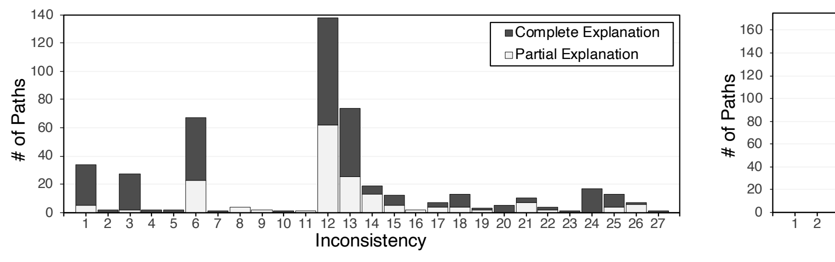
In Firefox and Safari, due to inconsistency #16, the inline script in Lines 3–5 is executed. However, the execution of the script generated by this script stops as it tries to fetch JS code from a URL not specified in the CSP. However, Chromium executes inline scripts due to this bug because `unsafe-inline` is not ignored. Moreover, interestingly, it allows arbitrary JS fetching from dynamically created scripts due to the `strict-dynamic` effect.

Site operators may specify `unsafe-inline` or host-source in a CSP, which they expect to be ignored by `strict-dynamic` in CSP3-supported browsers [38]. However, this bug allows the injection of an inline JS script, which should be blocked. We reported these bugs to the Chromium and WebKit vendors, and they patched these bugs.

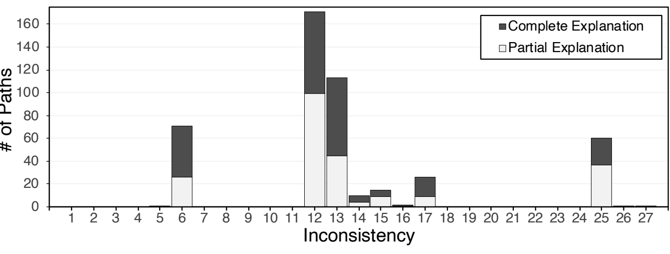
Cause #4: Non-supporting specific directives. Firefox does not support the `script-src-elem` [35] and `script-src-attr` [34] directives. Despite the known fact that Firefox supports CSP3 [9], we noticed that Firefox did not enforce the expected behaviors when these directives were present in given CSPs.

This inconsistency bug poses a security threat. Assume a site operator specifies their CSP with `script-src-elem` '`none`' to prevent the execution of inline scripts. Firefox enables the execution of inline scripts, contrary to the operator's expectations. Since these bugs are already known, we have not reported them. Firefox browser vendors had not implemented these two directives for approximately three years. Recently, Firefox implemented them in July 2022 [20] just two weeks before our submission.

Cause #5: Non-supporting specific directive values. Firefox does not support nonce-source, hash-source, and `strict-dynamic` in the `default-src` directive. Also, Safari does not support the `strict-dynamic` in the `default-src` directive. These inconsistent behaviors stem from Firefox and Safari not implementing a fallback mechanism; according to the specification, nonce-source, hash-source, and `strict-dynamic` in the `default-src` direc-



(a) Desktop browsers.



(b) Mobile browsers.

Fig. 6: *The number of decision tree paths for each root cause.*

tive should be applied as a fallback when the `script-src` directive is missing [13, 21].

These incomplete fallback behaviors have various security implications. Consider a website that defines the CSP of `default-src [hash-source] 'unsafe--inline'`. Note that, according to the specification, `unsafe-inline` should be ignored if nonce-source, hash-source, or `strict-dynamic` is specified. Chromium honored this policy by not allowing any inline scripts. However, Firefox and Safari executed inline scripts, allowing an adversary to execute an injected JS script.

The Firefox vendor has been aware of this bug, but it has not been fixed since the CSP2 was released six years ago [18]. After we discovered these three bugs via DiffCSP, we reported to the vendor that these bugs hadn't been fixed yet, and recently, in November 2022, these bugs were finally patched. Regarding the bug in Safari, we reported the bug to the corresponding vendor via WebKit Bugzilla, and the vendor acknowledged and fixed the bug.

D. Summary and Lessons

Complex CSP specification. We observed that the majority of bugs (18 out of 29 security bugs) were eventually triggered by combinations of multiple directive values in test CSPs. This result shows that different browsers interpret the same CSPs in different ways. Interestingly, we noticed that many of the bugs occurred due to divergent interpretations regarding new CSP3 directives and fallback mechanisms. In particular, 15 out of 18 bugs are caused by directives or values introduced in CSP3 (four from `strict-dynamic`, six from `script-src-elem`, four from `script-src-attr`, and one from `unsafe-hashes`).

We believe that this trend is inevitable because the expected behaviors become complicated as the number of directives and values increased to 24 and 11, respectively, in the CSP3 specification [58]. At the same time, this increasing complexity in CSPs requires systematic browser testing. We believe that our differential testing approach using a large number of adversarial HTML instances contributes to finding holes in existing regression tests in browser vendors.

CSP bypass via page redirection. We observed that eight out of 29 bugs (28%) were due to the improper updating of a CSP to enforce when page redirection occurs. In particular, when an iframe or a window was loaded once and its source or content was changed asynchronously, we observed many cases in which the parent CSPs were not properly inherited. However, we observed that Firefox correctly followed the CSP standard for these cases. WebKit, on the other hand, did not systematically test these edge cases, potentially allowing an attacker to execute arbitrary JS from the same origin.

Specification bugs. We have also identified that unclear or insufficient descriptions in the specification contributed to causing inconsistencies #6, #25, and #31. For inconsistency #6, we recommend inheriting the CSP of a parent window to its child window or iframe when these child instances open static files, and these files are delivered without any CSP headers.

To clarify the specification regarding inconsistency #25 (§V-C), we recommend the following two procedures: (1) remove only bogus source expressions, not entire directives, except in the worst case (i.e., when only `none` is left in the directive); and (2) accommodate and parse non-ASCII characters for source-expressions.

Regarding inconsistency #31, the specification said that hash-source can be applied to `javascript:` navigation [15] if `unsafe-hashes` is present. However, it does not clearly specify what exactly to hash (i.e., whether to include `javascript:`). Chromium-based browsers and Safari use the entire attribute value (i.e., `sha256(javascript:alert(1))`), but Firefox only uses everything after `javascript:` (i.e., `sha256(alert(1))`) to compute the hash value. Although this inconsistency does not pose a security threat, we recommend clearly specifying which part should be computed as a hash in the specification.

E. Decision Tree

Recall that we analyzed 525 and 581 paths in the decision trees of the desktop and mobile browsers, respectively. Figures 6a and 6b show the number of decision paths that contribute to identifying each security bug. The x-axis represents each inconsistency bug index, and the y-axis represents the number of paths that correspond to each found bug. We observed that a large number of decision paths corresponds to inconsistencies #12 and #13. This is because Firefox does not support the `script-src-elem` and `script-src-attr` directives. In our decision trees, these conditions are the majority cause of observed execution inconsistencies.

We also studied how helpful the decision trees are in analyzing the root causes of inconsistent execution results. We measured whether the root causes are well explained in conditions in the paths of the decision trees. In particular, we counted the number of paths that (1) completely explain the root cause and (2) partially explain the root cause, respectively. Here, the complete explanation path means that the conditions for a directive, a directive value, an HTML instance, and a

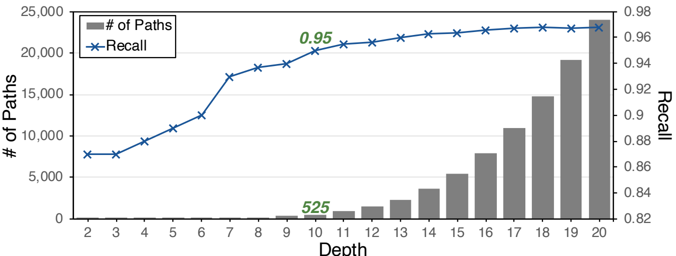


Fig. 7: The relationships between the depth of the decision tree, the recall of the training instances, and the number of decision paths.

status code are all present in the path to describe the observed inconsistency. For example, the second path of Figure 6a is a complete explanation path because its conditions have the `script-src` directive, the directive value of hash-source, an HTML instance that has JS execution in an event handler, and the 200 status code.

We observed that 538 of the 941 paths (57%) that map to their causes of the security bugs are complete explanation paths. We also observed that 24 out of the 27 inconsistencies (89%) were completely explained by at least one path. For the three inconsistencies (#8, #9, and #11) that were not fully explained, the number of instances that cause the inconsistency is small; the branching for that condition occurs below depth 10 in the tree. However, increasing the depth of the tree for desktop browsers to 14 reveals a path that completely explains the root causes of these inconsistencies.

Decision depth. We analyzed how recall and the number of paths to inspect change while varying the depth of the desktop decision tree. Figure 7 shows that increasing the depth causes an increase in the number of paths that need to be manually inspected. It also contributes to increasing the recall rate of the training instances, meaning that the decision tree becomes more explanatory by reducing non-explainable inconsistent execution results (i.e., false negatives). Note that we observed an analogous pattern in the mobile decision tree. We choose to set the depth of the decision tree to 10, which exhibits a high recall rate and produces an acceptable number of paths to analyze.

False positives (FPs). Among the 525 and 581 paths in the desktop and mobile decision trees, 39 and 42 were FPs out of 11,663 HTML files, respectively. These FPs correspond to the cases in which our testing frameworks reported false execution results because the execution of the generated test files exceeded a given timeout. Recall that we set the number of HTML instances in each file to 80 to boost the testing efficiency (§IV-C). This means that tests towards the end of the file may not be executed before the timeout, leading to inconsistent behavior and, thus, a false report. In practice, these FP cases can be scheduled again for further testing with an extended timeout to remove false positives.

F. Performance

A total of 200 hours with 88 cores and 153 hours with 192 core CPUs were consumed to find CSP enforcement bugs for desktop and mobile browsers, respectively. In addition, two researchers spent two days analyzing the decision trees to identify the root causes of the detected 37 bugs.

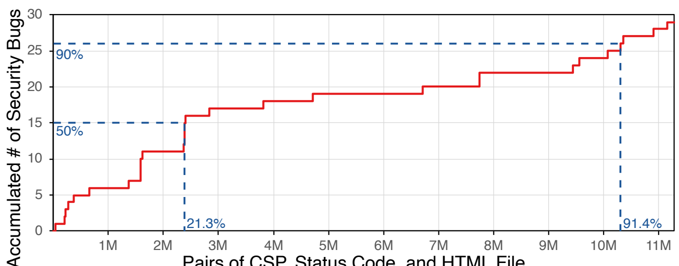


Fig. 8: The cumulative number of security bugs over executed test inputs.

Figure 8 shows the cumulative number of unique security bugs over the tested triads, each of which consists of a CSP, an HTML instance, and a status code. In particular, on the x-axis, we show the cumulative number of tested triads, and these triads are sorted in the order in which DiffCSP fetches those for execution. We increase the bug count on the first triad on the x-axis of which inconsistent execution results contribute to finding a bug. For 50% of the identified security bugs, DiffCSP tested less than 21.3% of the generated triads. For 90% of the bugs, it required testing 91.4% of the triads. The results of this analysis show that extending the test generation grammar and running DiffCSP has the potential to discover more bugs when DiffCSP is capable of generating new HTML instances involving JS execution.

VI. Limitations and Discussion

We leverage the inconsistent execution results for each generated input as a bug oracle. Therefore, DiffCSP cannot find a bug if all the browsers under testing exhibit the same bug [42, 47, 48]. One can overcome this limitation by defining expected behaviors for each input, enabling the testing of a single browser implementation. However, defining the correct behavior for each combination of 25,880 HTML instances and 1,006 CSPs is an open research problem. The automatic extraction and adoption of these correct behaviors represent interesting technical challenges that we leave to future research.

We also note that our sampling strategy of selecting one representative test instance among the test instances that share the same path may cause false negatives in identifying a new cause. This can happen when the test instances that correspond to a certain path yield both consistent and inconsistent execution results. For such cases, we manually analyzed all of the 187 and 217 paths that leading to leaf nodes having both inconsistent and consistent execution results in the desktop and mobile decision trees, respectively. However, this additional analysis did not reveal any new causes.

We acknowledge that our HTML grammar is not necessarily complete in generating all possible HTML forms for executing JS snippets. Therefore, if there exist unknown HTML forms of executing JS snippets, DiffCSP may miss CSP enforcement bugs, producing false negatives. However, note that, among 47 reported CSP bugs involving JS execution since 2010, including 18 ones that we found, DiffCSP generated HTML instances that triggered 46 bugs (97%).

We emphasize that the grammar rules that we proposed include not only the general rules corresponding to the HTML and ECMAScript specifications but also the adversarial generation rules corresponding to the 28 known CSP bugs. We

observed that these adversarial payloads were applied as-is to the browser's regression test set, without any combination or mutation. These adversarial grammar rules helped DiffCSP to identify 10 more bugs, which would not be discovered by only using the general grammar rules and the known payloads.

DiffCSP supports extending the current HTML grammar by adding new derivation rules. Security researchers and practitioners are thus able to add their own derivation rules to attempt to find more bugs. Specifically, they can extend the grammar by symbolizing and classifying new HTML or JS snippets with reference to the grammar in Table II. The testing CSPs can also be extended by adding new directives or values to Table I.

We adopt the interpretation of the computed decision trees to group test instances that share the same decision paths, thereby avoiding to analyze each inconsistent execution result and its corresponding test input. However, to identify common causes of these bugs, we still manually analyzed 525 and 581 paths for the desktop and mobile browsers, respectively. This step required two authors to spend two days identifying the causes of these paths.

We note that the conditions that appear in the decision tree paths help distinguish bugs due to unsupported features from bugs caused by developer mistakes. For example, five inconsistencies due to unsupported specific directives and directive values are distinguishable from other inconsistencies because those unsupported features appeared in the conditions in the paths leading to the groups having those bugs. These conditions can help browser vendors quickly disregard such cases.

VII. Related Work

Content Security Policy. Prior work has largely focused on studying the prevalence and struggles with CSPs [44, 49, 51, 56, 57, 71, 72], finding that building applications that are compliant with a safe CSP is a challenging task that few sites master. Follow-up work has then attempted to aid developers in the process of building CSPs. Pan *et al.* [56] proposed CSPAutoGen, which automatically composes declarations for enforcing a CSP in real-time through both analyses of existing resources and rewriting to enable compliance. Doupé *et al.* [49] introduced an automatic code rewriting technique to extract trusted inline scripts from web applications and use these extracted scripts in the `script-src` directive in generated CSPs.

Another recent line of research focuses on analyzing the trend of insecurity regarding CSPs deployed on the Web [43, 45, 50, 58, 60, 64, 65, 66, 68, 70]. Weichselbaum *et al.* [70] examined the effectiveness of deployed CSPs in 1,680,000 hosts on the Internet. They also demonstrated that approximately 95% of the collected policies offered little security protection against XSS attacks due to the usage of `unsafe-inline` and unsafe endpoints. Calzavara *et al.* [43, 45] examined the updates in CSPs and demonstrated that the CSPs were not frequently updated to mitigate insecure practices. Roth *et al.* [58] performed a historical analysis of how CSP adoption has evolved from 2012 to 2018. They found that many CSPs can be bypassed through expired domains or domains with typos, but more importantly, developers often struggle for years to set up a CSP or give up entirely. They nevertheless provided clear evidence of the uptake in CSP deployment, necessitating a thorough analysis of the enforcement in browsers. Eriksson and Sabelfeld [50] then analyzed the not-yet implemented `navigate-to` directive and proposed to automate the process of curating policies with the directive.

All of this research has demonstrated that curating a functional and secure CSP is a challenging task. However, none of the works have attempted to systematically analyze to what extent CSPs are properly implemented in browsers, in particular for edge cases. In contrast, with DiffCSP, our work aims to understand enforcement bugs in browsers, which may even result in the bypassing of seemingly secure CSPs.

For their 2016 paper, Calzavara *et al.* [43] also evaluated browser supports for CSPs. Contrary to DiffCSP using differential testing, they manually composed tests and modeled their expected behaviors, leaving many corner cases unexplored. In particular, they utilized visual cues to model the expected behavior corresponding to each page and manually accessed the test page from each browser to examine whether the visual cues were well represented (e.g., JS should be executed and an alert should appear). Due to this manual approach, they modeled 15 tests, contributing to finding one CSP bug involving JS execution.

By contrast, DiffCSP conducts scalable and systematic CSP enforcement testing by leveraging various types of adversarial HTML instances and differential testing, enabling to avoid a manual analysis to identify correct behaviors for each generated test, thereby helping to find a total of 37 bugs. By conducting differential testing, we narrow down the scope of promising tests that invoke potential bugs and then sample representative tests using the decision trees to manually investigate.

Browser security policy testing and analysis. There has been a surge in research in the study of web security policies provided by browsers [47, 54, 55, 67]. In particular, several works have focused on testing SSL/TLS implementation [42, 48], same origin policy [61, 63], HTTPOnly cookies [73], HSTS [53], and clickjacking protection [46] Calzavara *et al.* [47] found that the inconsistent adoption of security mechanisms across different pages within the same origin can express conflicting security requirements. Roth *et al.* [59] discovered that client-side policies, including CSPs, X-Frame-Options, HSTS, and security cookies, were applied differently when accessing the same site through different settings. Recently, there have been several studies on security policies in mobile browsers [52, 54, 55]. Luo *et al.* [54] investigated the browser supports for eight different security mechanisms, including CSP, HSTS, and referrer header, across 351 unique browser versions. Kondracki *et al.* [52] demonstrated that enabling data-saving functionality in mobile browsers poses security threats, including TLS man-in-the-middle attacks and HSTS deactivation.

VIII. Conclusion

With CSP's adoption rates rising and more sites to mitigate the impact of cross-site scripting (XSS) flaws every day, it is imperative that enforcement of these policies is consistent and

secure across *all* browsers. While prior work had found individual bugs in CSP, our community lacked a comprehensive and systematic way of testing CSP implementations.

To close this research gap, we proposed DiffCSP, the first differential testing framework designed to identify CSP enforcement bugs regarding JS execution. Our key contributions are (1) to propose an HTML grammar enumerating all known HTML instances that execute simple JS snippets, (2) to conduct differential testing to identify the correct behavior for each generated CSP and HTML instance, and (3) to analyze a large volume of execution inconsistencies by leveraging decision trees. Our testing uncovered critical flaws in major browsers, including Chrome, Firefox, and Safari, which allow an XSS attacker to fully bypass CSPs. We found 29 security bugs and eight functional bugs, demonstrating the effectiveness of DiffCSP in finding CSP enforcement bugs.

Acknowledgments

The authors would like to thank the anonymous reviewers for their concrete feedback. This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT), No.2020-0-00209 and Korea Internet & Security Agency (KISA) grant funded by the Korea government (PIPC) (No.1781000003, Development of a Personal Information Protection Framework for Identifying and Blocking Trackers).

References

[1] "Android debug bridge (ADB)," https://developer.android.com/studio/command-line/adb.

[2] "The Chromium projects - Blink (rendering engine)," https://www.chromium.org/blink/.

[3] "Content-disposition attachment vs 4xx status code from server," https://bugzilla.mozilla.org/show_bug.cgi?id=364354.

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

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

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

[7] "Create and manage virtual devices," https://developer.android.com/studio/run/managing-avds.

[8] "Cross site scripting (XSS) vulnerability payload list," https://github.com/payloadbox/xss-payload-list.

[9] "CSP browser support," https://content-security-policy.com/.

[10] "CSP bugs in Chromium," https://bugs.chromium.org/p/chromium/issues/list?q=component:Blink%3ESecurityFeature%3EContentSecurityPolicy.

[11] "CSP bugs in Firefox," https://bugzilla.mozilla.org/buglist.cgi?query_format=advanced&short_desc_type=allwordssubstr&resolution=FIXED&short_desc=csp%20bypass.

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

[13] "default-src," https://www.w3.org/TR/CSP3/#directive-default-src.

[14] "Directives," https://www.w3.org/TR/CSP3/#framework-directives.

[15] "Does element match source list for type and source?" https://www.w3.org/TR/CSP3/#match-element-to-source-list.

[16] "Does url match source list in origin with redirect count?" https://www.w3.org/TR/CSP3/#strict-dynamic-usage.

[17] "ECMA-262, 13th edition, june 2022 ECMAScript 2022 language specification," https://262.ecma-international.org/13.0/.

[18] "Firefox bugzilla - CSP: Enforce 'strict-dynamic' and nonce within default-src," https://bugzilla.mozilla.org/show_bug.cgi?id=1313937.

[19] "Firefox bugzilla - CSP script-src with hashes allow inline event handlers to match the hash (even if 'unsafe-hashes' is not present)," https://bugzilla.mozilla.org/show_bug.cgi?id=1683506.

[20] "Firefox bugzilla - implement CSP 'script-src-elem' and 'script-src-attr' directives," https://bugzilla.mozilla.org/show_bug.cgi?id=1529337.

[21] "Get fetch directive fallback list," https://www.w3.org/TR/CSP3/#directive-fallback-list.

[22] "GitHub robots.txt," https://github.com/robots.txt.

[23] "Global market share held by leading desktop internet browsers from january 2015 to december 2021," https://www.statista.com/statistics/544400/market-share-of-internet-browsers-desktop/.

[24] "HTML5 security cheatsheet," https://html5sec.org/.

[25] "Informational 1xx," https://www.rfc-editor.org/rfc/rfc9110.html#section-15.2.

[26] "Internationalized domain names for applications (IDNA): Definitions and document framework," https://datatracker.ietf.org/doc/html/rfc5890.

[27] "Issue 882270: Security: url spoofing using 304 status code," https://bugs.chromium.org/p/chromium/issues/detail?id=882270.

[28] "Page.navigate doesn't fail for status code other than 200 (eg. 204 - no content)," https://bugzilla.mozilla.org/show_bug.cgi?id=1618863.

[29] "Parse a serialized CSP," https://www.w3.org/TR/CSP3/#parse-serialized-policy.

[30] "Playwright enables reliable end-to-end testing for modern web apps." https://playwright.dev/.

[31] "scheme-part matching," https://www.w3.org/TR/CSP3/#match-schemes.

[32] "Script directives pre-request check," https://www.w3.org/TR/CSP3/#script-pre-request.

[33] "script-src," https://www.w3.org/TR/CSP3/#directive-script-src.

[34] "script-src-attr," https://www.w3.org/TR/CSP3/#directive-script-src-attr.

[35] "script-src-elem," https://www.w3.org/TR/CSP3/#directive-script-src-elem.

[36] "Source lists," https://www.w3.org/TR/CSP3/#framework-directive-source-list.

[37] "Twitter robots.txt," https://twitter.com/robots.txt.

[38] "Usage of 'strict-dynamic'," https://www.w3.org/TR/CSP3/#strict-dynamic-usage.

[39] "web-platform-tests for CSP in Chromium," https://chromium.googlesource.com/chromium/src/+/02495a2c0b813fd89d2759482255d08f2b0643f8/third_party/blink/web_tests/external/wpt/content-security-policy.

[40] "WHATWG fetch living standard — last updated 25 may 2022," https://fetch.spec.whatwg.org/.

[41] "YouTube favicon.ico," https://www.youtube.com/favicon.ico.

[42] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, "Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2014, pp. 114–129.

[43] S. Calzavara, A. Rabitti, and M. 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*, 2016, pp. 1365–1375.

[44] ——, "CCSP: Controlled relaxation of content security policies by runtime policy composition," in *Proceedings of the USENIX Security Symposium*, 2017, pp. 695–712.

[45] ——, "Semantics-based analysis of content security policy deployment," *ACM Transactions on the Web*, vol. 12, no. 2, pp. 1–36, 2018.

[46] S. Calzavara, S. Roth, A. Rabitti, M. Backes, and B. Stock, "A tale of two headers: A formal analysis of inconsistent Click-Jacking protection on the web," in *Proceedings of the USENIX Security Symposium*, 2020, pp. 683–697.

[47] S. Calzavara, T. Urban, D. Tatang, M. Steffens, and B. Stock, "Reining in the web's inconsistencies with site policy," in *Proceedings of the Network and Distributed System Security Symposium*, 2021.

[48] Y. Chen and Z. Su, "Guided differential testing of certificate validation in SSL/TLS implementations," in *Proceedings of the International*

Symposium on Foundations of Software Engineering, 2015, pp. 793–804.

[49] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna, “deDacota: Toward preventing server-side XSS via automatic code and data separation,” in *Proceedings of the ACM Conference on Computer and Communications Security*, 2013, pp. 1205–1216.

[50] B. Eriksson and A. Sabelfeld, “AutoNav: Evaluation and automatization of web navigation policies,” in *Proceedings of the Web Conference*, 2020, pp. 1320–1331.

[51] M. Fazzini, P. Saxena, and A. Orso, “AutoCSP: automatically retrofitting CSP to web applications,” in *Proceedings of the International Conference on Software Engineering*, 2015, pp. 336–346.

[52] B. Kondracki, A. Aliyeva, M. Egele, J. Polakis, and N. Nikiforakis, “Meddling middlemen: Empirical analysis of the risks of data-saving mobile browsers,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 2020, pp. 810–824.

[53] M. Kranch and J. Bonneau, “Upgrading HTTPs in mid-air,” in *Proceedings of the Network and Distributed System Security Symposium*, 2015.

[54] M. Luo, P. Laperdrix, N. Honarmand, and N. Nikiforakis, “Time does not heal all wounds: A longitudinal analysis of security-mechanism support in mobile browsers,” in *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[55] A. Mendoza, P. Chinprutthiwong, and G. Gu, “Uncovering HTTP header inconsistencies and the impact on desktop/mobile websites,” in *Proceedings of the international conference on World wide web*, 2018, pp. 247–256.

[56] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou, “CSPAutoGen: Black-box enforcement of content security policy upon real-world websites,” in *Proceedings of the ACM Conference on Computer and Communications Security*, 2016, pp. 653–665.

[57] S. Roth, M. Backes, and B. Stock, “Assessing the impact of script gadgets on CSP at scale,” in *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, 2020, pp. 420–431.

[58] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock, “Complex security policy? a longitudinal analysis of deployed content security policies,” in *Proceedings of the Network and Distributed System Security Symposium*, 2020.

[59] S. Roth, S. Calzavara, M. Wilhelm, A. Rabitti, and B. Stock, “The security lottery: Measuring client-side web security inconsistencies,” in *Proceedings of the USENIX Security Symposium*, 2022.

[60] S. Roth, L. Gröber, M. Backes, K. Krombholz, and B. Stock, “12 angry developers-a qualitative study on developers’ struggles with CSP,” in *Proceedings of the ACM Conference on Computer and Communications Security*, 2021, pp. 3085–3103.

[61] J. Schwenk, M. Niemietz, and C. Mainka, “Same-Origin policy: Evaluation in modern browsers,” in *Proceedings of the USENIX Security Symposium*, 2017, pp. 713–727.

[62] T. Shiba, T. Tsuchiya, and T. Kikuno, “Using artificial life techniques to generate test cases for combinatorial testing,” in *Proceedings of the IEEE Annual International Computer Software and Applications Conference*, 2004, pp. 72–77.

[63] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee, “On the incoherencies in web browser access control policies,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 2010, pp. 463–478.

[64] D. F. Some, N. Bielova, and T. Rezk, “On the content security policy violations due to the same-origin policy,” in *Proceedings of the international conference on World wide web*, 2017, pp. 877–886.

[65] S. Stamm, B. Sterne, and G. Markham, “Reining in the web with content security policy,” in *Proceedings of the international conference on World wide web*, 2010, pp. 921–930.

[66] M. Steffens, M. Musch, M. Johns, and B. 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.

[67] B. Stock, M. Johns, M. Steffens, and M. Backes, “How the web tangled itself: Uncovering the history of client-side web (in) security,” in *Proceedings of the USENIX Security Symposium*, 2017, pp. 971–987.

[68] P. Stolz, S. Roth, and B. Stock, “To hash or not to hash: A security assessment of CSP’s unsafe-hashes expression,” in *Proceedings of the IEEE Security and Privacy Workshops - SecWeb*, 2022.

[69] J. Wang, B. Chen, L. Wei, and Y. Liu, “Skyfire: Data-driven seed generation for fuzzing,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 2017, pp. 579–594.

[70] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. 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*, 2016, pp. 1376–1387.

[71] M. Weissbacher, T. Lauinger, and W. Robertson, “Why is CSP failing? trends and challenges in CSP adoption,” in *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses*, 2014, pp. 212–233.

[72] M. Wilhelm, S. Roth, and B. Stock, “RetroCSP: Retrofitting universal browser-support for CSP,” in *Proceedings of the IEEE Security and Privacy Workshops - SecWeb*, 2022.

[73] Y. Zhou and D. Evans, “Why aren’t HTTP-only cookies more widely deployed,” *Proceedings of Web*, vol. 2, 2010.

IX. Appendix

A. Additional Root Causes

Cause #6: Auto-enabling directive values by default. We observed that Firefox enabled unsafe-hashes in the script-src directive by default. We argue that this default behavior deviates from the CSP specification and poses a security threat.

```
1  CSP: script-src-elem 'sha256-Mg0QnPgA...';
2  <img onload =
3    "document.write(this.getAttribute('data'))"
4  </img>
```

Consider a webpage with the CSP of script-src 'sha256-Mg0QnPgA...', as shown above. Note that the inline script in Line 3 must be blocked because unsafe-hashes is not specified in the CSP. However, Firefox allows the execution of the inline script in the onload handler due to this bug. In this case, the adversary is able to reuse the existing event handler to execute an injected script [68]; they are able to inject the arbitrary tag embedding the exploiting payload in the data attribute, thus allowing the XSS attack. This bug had been reported to Firefox by others over three years ago, but Mozilla hasn't patched it [19]. After we found this bug via DiffCSP, we mentioned to the vendor that this bug hadn't been fixed yet, and recently, in November 2022, it was finally patched.

We also found that Safari enables * in the script-src-elem directive by default. This default behavior bug imposes a security threat; when a site operator attempts to block all script requests by specifying script-src-elem 'none', Safari allows fetching scripts from arbitrary endpoints, contrary to the operator's expectations. We reported this bug to the WebKit team, and the vendor patched the bug.

Cause #7: Auto-enabling directive values on specific conditions. We have found several bugs in Safari in which specific directive values are automatically enabled under certain conditions. For example, Safari automatically activates unsafe-inline for the script-src-elem directive when strict-dynamic is present. Also, when a hash-source is specified in the script-src-elem or script-src-attr directive, we observed that unsafe-inline is auto-enabled for both directives. Unfortunately, inconsistencies #20–#23 could be triggered even with an arbitrary hash value rather than the hash of an inline script in a testing webpage. These bugs are thus able to enable unsafe-inline in certain CSPs, thus allowing an XSS attacker to inject inline scripts.

We have identified these bugs in our differential testing. However, they had already been patched by the time that we identified them.

Cause #8: Non-supporting CSP for specific status code. We observed that Chromium ignored a CSP when the HTML response came with the status code of 100. We further confirmed that this behavior also occurred when the status code is 101 or 102. This means that Chromium-based browsers ignore any CSPs when a webpage is fetched with one of these response status codes.

Note that the 1xx class of status code indicates that a server has received a request and continues to generate its response [25]. According to the WHATWG fetch standard, a response with the status code 100 or 102 should be ignored, and status code 101 should have a body set to null [40]. We observed that Safari and Firefox correctly followed this standard. However, Chromium treated the content of these responses normally as if they had the status code of 200, while ignoring the headers of these responses. Therefore, when a naive developer writes a web page with any of the response codes above, the attacker is able to inject an arbitrary JS script regardless of the CSP in this webpage.

Chromium developers acknowledged and patched our reporting of this bug and stated that this bug contributes to bypassing not only CSP but also other security policies, including HTTP Strict Transport Security (HSTS) and X-Frame-Options.

Cause #9: Incorrect handling of malformed CSPs. We observed that Chromium and Safari ignored an entire given directive when the directive contains an invalid value. In contrast, Firefox only drops an invalid value in the directive. For instance, consider the CSP of script-src http://a.com http://<non-ASCII-chars>.com. Chromium and Safari ignore the script-src directive, allowing the execution of any scripts. Conversely, Firefox still honors the directive by allowing the execution of scripts from http://a.com and blocking the execution of other inline and external scripts.

Interestingly, the CSP specification describes the Chromium and Safari behavior as valid [29]. The reason is to prevent the worst scenario that the page itself does not work due to a malformed CSP. Assume that we specify default-src http://<non-ASCII-chars>.com. If a browser only drops the invalid value, the CSP becomes default-src, which is semantically equivalent to default-src 'none', which blocks all resources. On the other hand, if a browser drops the entire directive in compliance with the CSP specification, it blocks no resources on the page. Although the handling of malformed CSPs adopted by the current specification provides convenience to end users, it poses a security threat.

Cause #10: Allowing out-going requests. According to the script directives pre-request check in the CSP specification [32], when there is strict-dynamic in the script-src directive, the request from the parser-inserted script (e.g., regular script tags) must be blocked. However, Firefox does not follow this specification and sends outgoing requests for <script src=[URL]></script> even though CSP is script-src 'nonce-123' 'strict-dynamic'. In Chromium-based browsers, the same bug occurs when a page is <script nonce=123>document.write('<script src=[URL]><script>');</script>, and its CSP is script-src 'nonce-123' 'strict-dynamic' [URL].

Since script execution is still blocked, the impact of the problem for XSS attacks is limited. However, such scripts can be used for the exfiltration of sensitive data. We have reported these bugs; however, the vendors have not responded yet.