Head(er)s Up! Detecting Security Header Inconsistencies in Browsers

Jannis Rautenstrauch
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
jannis.rautenstrauch@cispa.de

Trung Tin Nguyen
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
tin.nguyen@cispa.de

Karthik Ramakrishnan
Georgia Institute of Technology
Atlanta, USA
rkarthik@gatech.edu

Ben Stock
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
stock@cispa.de

Abstract

In the modern Web, security headers are of the utmost importance for websites to provide protection against various attacks, such as Cross-Site Scripting, Clickjacking, and Cross-Site Leaks. As each security header uses a different syntax and has unique processing rules, correctly implementing them is a complex task for both browser and website developers. Inconsistency in browser behavior related to security headers harms websites as their security depends on their users' browsers. At the same time, compatibility issues may deter developers from deploying such headers in the first place.

In this work, we performed a differential evaluation of the security header parsing and enforcement behavior in desktop and mobile browsers to uncover problematic browser differences. We systematically ran 177,146 tests covering 16 security-relevant headers multiple times in 16 browser configurations covering over 97% of the browser engine market share. We identified 5,606 (3.16%) tests that behave inconsistently across browsers. Our subsequent analysis revealed 42 root causes, highlighting the prevalence of implementation issues. 31 of these root causes were yet unknown and resulted in 36 bug reports against the affected browsers and specifications. Many of our reports have already resulted in fixes improving web consistency and users' security. To foster open science and enable browser vendors to continuously test their security header implementations, we open-source our test framework.

CCS Concepts

• **Security and privacy → Browser security**; **Web protocol security**; • **Software and its engineering → Software testing and debugging**.

Keywords

Browser; Security Headers; Differential Testing; Parsing; CSP; XFO; HSTS; XCTO; TAO; COEP; CORP; COOP; PP; CORS

CCS '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765119

ACM Reference Format:
Jannis Rautenstrauch, Trung Tin Nguyen, Karthik Ramakrishnan, and Ben Stock. 2025. Head(er)s Up! Detecting Security Header Inconsistencies in Browsers. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3719027.3765119

1 Introduction

The Web is everywhere. Sadly, security was an afterthought, and vulnerabilities are common. As millions of legacy servers and websites exist, it is difficult for browser vendors to improve the default security model as this would break existing functionality. Instead, browsers introduced many opt-in security headers that websites can use to improve their security. These security header mechanisms are often complex. Web developers make mistakes and often configure them incorrectly or insecurely [51, 42]. At the same time, browser developers have trouble implementing the specifications correctly and diverge from the specifications both by accident [55, 5, 19] and intentionally [4, 7, 17]. As a result of both misconfigured headers and browser differences, websites are often not protected adequately even though their developers think so.

Fortunately, differential testing makes it possible to compare browsers and identify implementation differences without relying on the header specifications that are often ambiguous. In the past, differential approaches discovered implementation and specification issues in individual headers such as X-Frame-Options and Content-Security-Policy [9, 55, 45].

However, prior works [9, 55, 51] and testing suites [12, 27] primarily focused on the correct enforcement of valid header configurations. As websites often use invalid and broken header configurations [45, 25, 41], it is essential to test the parsing behavior of browsers for invalid headers as well.

Our research aims to answer whether security headers are parsed and enforced correctly and consistently across various web browsers. A seemingly natural way of approaching this is to rely on expected-outcome tests such as those provided by WPT [12]. However, this not only requires to have a comprehensive set of expected-outcome tests, but more importantly also specifications that cover all potential edge cases and are unambiguous. Since neither of the requirements is given, we instead rely on differential testing. This not only allows us to identify differences in browser implementations (implying at least one of them is incorrect), but also to pinpoint ambiguities in the specifications by investing the root causes of

the observed differences. To that end, we create an extensive test set with mutated responses for 16 security-related headers, resulting in a total number of 177,146 tests. We initially run the tests in twelve browser configurations multiple times to ensure deterministic results, executing over eleven million test runs in total. We then propose a new clustering approach based on outcome-browser mappings, which enables us to reduce the discovered 4,432 differences to 38 root causes across four main categories. The results show that browser differences within a browser engine are rare, but differences between the three main browser engines (Blink, Gecko, and WebKit) are common. While currently security headers are not parsed and enforced consistently, we hope that thanks to our bug reports to browser vendors and specifications and by open-sourcing our test framework, users will soon experience more consistent protections.

To demonstrate the versatility of our framework, we run it several months later on the four most up-to-date browser configurations, showcasing the ability to easily integrate our framework into the release cycle of a browser vendor. In this process, we could confirm that many of the fixes to our bug reports are complete but also discovered four additional root causes causing new differences.

To sum up, we make the following contributions:

(1) We propose a methodology for differential testing of security headers with parametrized tests using thousands of responses and various origin relations. (Section 4)
(2) We run our differential testing pipeline with 177,146 tests for twelve features in 16 browser configurations. (Section 5)
(3) We identify 42 unique root causes resulting in 36 browser and specification bug reports. (Section 7)
(4) We open-source our framework to foster open science and enable browser vendors to continuously test their security header implementations [40].

2 Background

In this section, we first explain the basics of HTTP and continue with security features related to HTTP Headers.

2.1 HTTP and Response Headers

Figure 1 shows an HTTP request and response pair. A request consists of a request-line (method, resource, and version), optional and required headers (additional information about the request, e.g., cookies to authenticate), and an optional body (e.g., form submission content). A response has a status-line (200 OK, response was successful), optional and required headers with metadata (e.g., the content-type of the response body, or security headers to enforce in the browser), and the actual response, e.g., HTML content.

The browser performs one request to the URL entered in the URL bar, and then the server responds. The browser interprets the response based on the metadata in the HTTP response headers, the actual response body, and the context of the request. If the response contains HTML, the browser renders the HTML, which might cause additional requests to included resources. JavaScript on the page can also perform additional requests, but the general process is always a request to the server and a response to the browser.

2.1.1 Versions and Encodings. HTTP has different versions with the same general semantics but different transport encoding and

```
GET /not-frameable/ HTTP/1.1
Host: secure.site
```

(a) HTTP/1.1 request

```
HTTP/1.1 200 OK
Date: Mon, 14 April 2025 23:59:59 AoE
Content-Length: 500
Content-Type: text/html
X-Frame-Options: DENY

(500 bytes of the requested web page)
```

(b) HTTP/1.1 response with XFO header

Figure 1: Example HTTP request and response pair.

protocols. HTTP/1.1 [16] is a plaintext protocol over TCP, and HTTP/2 [48] is a binary protocol over TCP. HTTP/3 [6] is a binary protocol over QUIC. This paper focuses on the semantics of HTTP and uses HTTP/1.1 as the transport protocol, which has the least strict parsing rules due to its age and being a plaintext protocol.

HTTP/1.1 is a plaintext protocol, which means it should be readable by observing the traffic on the wire. The HTTP request-line, status-line, and headers should contain ASCII text bytes only. In contrast, the HTTP request and response bodies are arbitrary bytes, which meaning is specified by the content-type and content-encoding headers (e.g., a gzip-encoded body or an HTML body that is UTF-8 encoded). However, the underlying TCP protocol ignores the content of the message and allows arbitrary bytes everywhere. Thus, HTTP processors need to decide how to deal with other bytes when receiving HTTP responses. They often ignore the invalid bytes or even allow UTF-8 in some headers.

2.1.2 HTTP Fields/Headers. The metadata in HTTP requests and responses is specified in HTTP fields, colloquially called headers. These headers can have a variety of use cases, such as informing the receivers about the cacheability of the message or the content type. This paper focuses on headers that activate or deactivate security features in browsers. Headers have a name (case insensitive in HTTP/1.1) and a value (separated by a colon in HTTP/1.1). The values of headers are usually defined by augmented Backus-Naur form grammars (ABNF) [14] or, for new headers, are defined as Structured Fields [36]. Depending on the header, multiple values in a list, or multiple instances of the same header may be allowed.

2.1.3 HTTP Parsing. Browsers have to parse the received HTTP responses regardless of whether the messages are well-formed. Due to their history of valuing functionality and pleasing users, browsers are error-tolerant for the received content. For example, they try to parse invalid HTML [24] and guess the content type via MIME sniffing [54]. They also deal with invalid responses on the header and status-line levels. They must decide what to do if they receive several headers with the same name, unspecified values, or headers using disallowed bytes. They can use the specified ABNFs for the values, parsing algorithms, or self-created ad-hoc logic. If browser parsing behavior diverges, it can lead to a feature being active in some browsers and not in others which can be disastrous in the case of security headers such as CSP.

2.2 Security Features in Browsers

Browsers are in a constant tension between security and functionality. They want to enable users to surf safely online and introduce many new security features to achieve this. However, some security improvements could break existing websites. For example, if browsers only allowed secure HTTP connections (HTTPS), this would significantly reduce the impact of network attacks. However, it would also break legacy websites that do not support HTTPS. To still allow websites better security, browsers introduce security headers that websites can set to opt-in to new defenses. The following explains three security issues that such headers can mitigate.

2.2.1 Frame Control and Clickjacking. A fundamental aspect of the Web is to include other websites using iframes. As these frames can be styled transparently and overlayed by other content, it is possible to trick users into performing actions on embedded sites, such as liking a page or deleting an account, without the user noticing. These attacks are called Clickjacking attacks [23]. One way to stop such attacks is to disallow the framing of a response. Websites can set the X-Frame-Options (XFO) header [53] to deny framing for all sites or all cross-origin sites. Alternatively, the Content-Security-Policy (CSP) header's frame-ancestors directive [50] allows for more fine-grained control about who can frame a response.

2.2.2 XSS Mitigation and Script Control. The most prominent web security issue is called Cross-Site Scripting (XSS) [22], where unauthorized script content is injected into a page and executed in the context of the page. While this attack has been known for over 20 years [32], and many defenses and countermeasures have been proposed, it is still one of the leading threats to websites. One mitigation technique is the Content-Security-Policy (CSP) [49], initially designed to restrict script content on a site and now allows the control of many features. The `script-src` directive controls where scripts can be loaded from and, by default, disallows inline scripts, making XSS harder to exploit.

2.2.3 HTTPS Enforcement. Another big issue on the Web are insecure HTTP connections. While most mixed content is nowadays blocked by default, top-level connections to HTTP are problematic as they are allowed (some sites only support HTTP) and can be manipulated in transit. The main option for a site to ensure to always be loaded via HTTPS is HTTP Strict Transport Security (HSTS) [26]. A site can set the Strict Transport Security header with a `max-age` attribute, and all following connections to this site will be performed via HTTPS for the specified time. In addition, there is a preload list of sites always loaded via HTTPS [47].

3 Key Ideas

Security headers control crucial security features in browsers. For example, the X-Frame-Options (XFO) header can be used to disallow the embedding of a URL into an `iframe`, `object`, or `embed`, protecting the URL from clickjacking attacks. The browser parses the XFO header and decides whether to allow the embedding based on the content of the header, the origin of the embedded URL, and the origins of all ancestors. For meaningful protection, all browsers should offer the same consistent level of protection; otherwise, the security of a website depends on the visitor's browser that could differ from the browsers tested by the web developers.

3.1 Parsing Challenges

Parsing security headers is notoriously tricky, specifications are often complex, potentially resulting in differences across browsers. For example, at the time of writing, Chrome allows whitespace between header names and the colon, whereas Safari and Firefox consider such headers to be invalid. Additionally, only Firefox redirected responses with status code `300` until recently. These differences can be caused by unintended implementation mistakes, intentional design choices, or something in between, e.g., the specification is unclear, and the developers had to implement something. Most importantly, such differences are problematic as they often reduce security (e.g., an XFO header that only protects visitors using Firefox and leaves users of other browsers unprotected) and pose compatibility issues. Thus, it is crucial to discover and fix browser differences related to security headers.

3.2 Differential Testing

While frameworks such as web-platform-tests (WPT) [12] exist to check that browser implementations follow the specifications and are compatible with each other, they have fundamental shortcomings in the area of security headers. Their tests have to specify the correct outcome and thus can only test well-defined behavior. Additionally, their tests and responses are hand-crafted, focusing on the correct behavior for a small set of responses instead of testing the full range of responses, missing many potential differences.

Writing a comprehensive test suite that covers the expected outcomes for all edge cases is difficult. Even the experts implementing these features in browsers fail to do so even though they write many WPT tests with expected outcomes in parallel with their implementation. With differential testing, one does not need to understand the complex specifications; instead, one only needs to reactively understand the correct outcome for the discovered difference. If the correct outcome is unclear, one can still report the ambiguity in the specification, including the current behavior of browsers. Thus, we decided to use a differential testing approach visualized in Figure 2. For testing a feature such as XFO, we create a test page recording whether a resource embedding was successful without specifying the correct behavior. We then visit the test page with thousands of generated response resources in various browsers and compare the recorded outcomes across the browsers. In this example, only Firefox blocks cross-origin framing for an XFO value of `SAME ORIGIN` whereas users of Chrome are not protected from clickjacking attacks as the invalid value is ignored and the page renders. Each discovered difference indicates a potential problem. By clustering the recorded differences and performing a manual root cause analysis, we estimate each root cause's impact and report them to browser developers. Here, we discovered that Firefox incorrectly removed whitespace from anywhere within an XFO header instead of only stripping leading and trailing space [38].

3.3 Threat Model

When building or updating their website's code, developers need to rely on their own browser to test new functionality and security mechanisms. Thus, it is paramount that irrespective of the used browsers, the developer observes the same security (or lack thereof) as any visitor of the site will. Hence, for our threat model

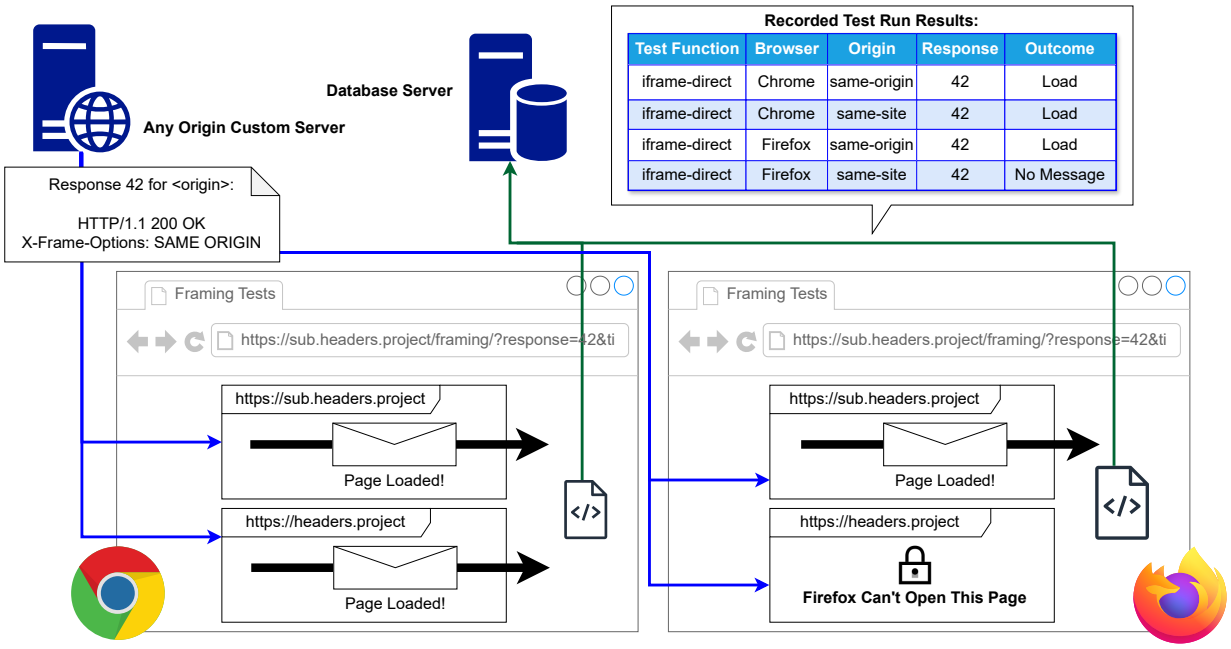

Figure 2: Differential testing framework showing a difference in XFO parsing between Chrome and Firefox.

we consider developers using one browser and deploying security header configurations that, due to browser inconsistencies only work in some browsers, thus unknowingly leaving many users unprotected.

4 Methodology

This section explains how we detect and analyze browser differences in security header implementations. We start with an illustrative example and then detail our header selection, test function creation, and response creation process. Finally, we describe how we analyze the obtained data.

4.1 Example and Terminology

Figure 2 shows an example execution of our differential testing infrastructure for two framing tests in Chrome and Firefox. Both browsers open the same URL pointing to our test server, which executes two tests: one frames a same-origin URL and another a same-site URL; both URLs contain response ID 42 in a GET parameter. The framed URL responds with an X-Frame-Options header of `SAME ORIGIN`, which is not a valid value according to the specifications [53]. Chrome ignores the header with the invalid value and both frames load. Firefox ignores the space, interprets the value as `SAMEORIGIN`, and thus blocks the cross-origin frame. The frames that are successfully loaded send a message to the top-level page. The page then sends the recorded outcomes to our database server.

In the above example, we visit the *template page* for framing. There is one template page for each considered *feature*. The concrete settings are specified via URL parameters. Each template page implements one or more *test functions*. These JavaScript functions perform specific actions and, ultimately, report an outcome. In our example, the test function is called `Framing/iframe direct` and creates an iframe with a given URL and records whether the iframe loads successfully. The framed resource URLs look like https://<origin>/framed-response/?response=<id>. In our example, the origin relation (OR) to the top-level page is either same-origin or same-site. The response (R) has ID 42. This response has status code 200 and an X-Frame-Options header with the value `SAME ORIGIN`. The body of the page sends a postMessage to the parent. Formally, we call the instantiation of a test function (TF) with an origin relation and a concrete response a *test*: $T = (TF, OR, R)$. The execution of a test in a concrete browser (B) is then called *test run*: $TR = (T, B)$.

4.2 Header Selection Process

We want to test HTTP response headers that control privacy and security features in browsers. The behavior of the header should be documented by an official web specification group such as the Web Hypertext Application Technology Working Group (WHATWG), the World Wide Web Consortium (W3C), or the Internet Engineering Task Force (IETF). However, it can be a working draft as many specifications only ever reach official standard status years after they are implemented in browsers.

We started with the list of security headers on MDN Web Docs [33], removed any headers that did not fulfill our criteria, and added security-relevant headers, missing from the MDN list, based on our expert knowledge. We describe the complete list of tested headers in Section 5.2.

4.3 Test Function Creation

We considered each selected header's purpose, abstracted it to a feature, and created a template page. For example, the XFO header is

responsible for Framing-Control, which can be tested via a Framing test. Some headers, such as CSP, can have multiple purposes, and we created several template pages for them. We reuse the same template page if two headers control the same feature (e.g., XFO and CSP `frame-ancestors` both are responsible framing-control).

For each feature, we created one or several test functions. For example, for framing, we frame a URL as either `iframe`, `object`, or `embed` and optionally enable the sandbox directive. In addition, we use different origin relations: same-origin, cross-origin same-site, and cross-site. Such tests are required to assess whether the whole ancestor chain is considered or only the parent.

To write the test functions, we build on the Web Platform Test project's testharness.js framework [10], which we adapt to our needs. In the default test harness, the correct behavior must be defined up front, and the test either passes or fails. We change testharness.js to record an outcome, such as framing worked or failed, and always pass the test if *any* outcome was successfully saved such that we can perform pure differential testing. In addition, we adapted the WPT infrastructure to allow sending arbitrary bytes everywhere in the responses and have a URL parameter that selectively turns off HTTPS support to test for mixed content issues.

4.4 Response Generation

We aim to cover the browsers' header parsing and enforcement code as comprehensively as possible. Thus, in addition to creating complex test functions and using multiple origin relations, we require a large set of responses.

For each header, we create one *blocking* and one *non-blocking* response. For example, `DENY` to activate XFO and disallow framing and `INVALID` which should not activate XFO. In addition, we create responses to test for more complex behavior: a response that redirects to test whether the redirect or the security header takes precedence, a response with an empty header to check for the default behavior, and other well-specified values such as `SAMEORIGIN` for XFO. We call these initially created responses *basic responses*.

The basic responses are helpful to test for general browser differences, such as differences in HTTP upgrades, and for header-specific enforcement differences, such as a browser not considering the entire ancestor chain. However, they are unsuitable for detecting subtle differences in the parsing process. Thus, we create a second, more extensive set of responses called *parsing responses*.

For each header, we use seed values from WPT [11], Siewert et al. [45], crawler.ninja [25], and the headers specifications,. We group the seed values into *block*, *allow*, *partial*, *legacy*, and *other* categories. In addition, we collect legacy and alternative header names such as `X-WebKit-CSP` for Content-Security-Policy.

We then combine these seed values with various status codes, use duplicate headers or multiple values in the same header, and mutate the names and values to generate our complete set of responses. We use the following mutations: we change the casing (all-upper, all-lower, random-case); we enclose the value in spaces, double quotes, and single quotes; we remove all spaces, double all spaces, or convert all spaces to tabs; we insert characters (all ASCII bytes including control characters, double space, non-breaking space, and full-width comma) at the beginning, the end, or the middle of the string; we replace all instances of a semicolon, comma, colon, equal sign, single quotation mark, double quotation mark, dash, and underscore with all other values, remove them, or replace them with a space, a backtick, a tick, or eight different UTF-8 quotation mark symbols. We chose these mutations to cover common typos and confusion with other headers. The complete list of mutations is available in our repository [40].

4.5 Root Cause Analysis Process

The above described testing pipeline allows us to run millions of tests and record thousands of differences. However, analyzing every detected difference individually is inefficient, as many differences have the same underlying root cause.

Consider the example from Figure 2. Firefox blocked cross-origin framing for `X-Frame-Options: SAME ORIGIN` whereas Chrome allowed it. The *same* difference is also visible for `DE NY` and `D\tENY`. The underlying root cause is that Firefox removes whitespace from anywhere within an XFO value instead of only stripping leading and trailing whitespace as required by the specifications.

4.5.1 Outcome-Browser Mapping Clustering. We perform an initial clustering step, grouping tests that are similar. For each test function, we cluster the tests by the mapping of recorded outcomes to browsers. For example the test function *iframe-direct* can have two outcomes `Message received` and `No message`. A first cluster, containing 80 tests, could have the mapping `Message received: {Chrome, Safari}, No message: {Firefox}`. A second cluster, containing 128 tests, could have the mapping `Message received: {Firefox, Safari}, No message: {Chrome}`.

We then use a manual analysis step to identify the semantic similarities between the tests in a cluster and to decide which browsers are wrong and whether they intentionally behave differently. We also sample the tests and verify the recorded outcomes to confirm that our results are correct.

For example, in the above example, the first cluster could contain responses with XFO headers with spaces such as `DE NY` and `SAME ORIGIN` that only result in blocking the frame in Firefox resulting in no message being send. The second cluster could contain header names that contain a NULL byte, such as `X-Frame\x00-Options: DENY` where Chrome blocks the entire response, and Firefox and Safari only ignore the invalid header.

After identifying the *root cause* of a difference, we name it and note down all details. Then, we search the specifications for the correct behavior and look for bug reports and browser documentation to decide which browser is wrong, if there is an issue with the specification, and whether it is already known. If it is not already known, we create a bug report against the affected browsers or the specifications. Lastly, we reason about the potential consequences of the detected root cause and classify it as mainly affecting security, privacy, or compatibility.

4.5.2 Comparing Browser Updates. After the root cause analysis has been done once, there is an easier way to check for expected and unexpected browser changes in new versions. Instead of comparing all tested browsers, it is possible to compare only two browser configurations, for example, Chrome version 122 and Chrome 131. Here, it is possible to quickly glance at the clusters and see whether the differences between the versions are due to known changes

or unrelated changes that might be new root causes. Additionally, it is possible to only look at all the tests that result in a unique outcome in a single browser configuration. Unique outcomes mean the browser's behavior changed but is still not aligned with other browsers, hinting at incomplete fixes or new root causes.

5 Experimental Settings

Based on our above-described methodology, we instantiated our test infrastructure with 16 browsers, twelve features, and 177,146 tests, as described in the following.

5.1 Tested Browsers

Our goal is to find differences in browsers used in the real world. As of 2025, there are only three main browser engines available, which all major browsers are built on top of: Blink (used in Chrome, Brave, Edge, and many more), Gecko (used in Firefox and Tor), and WebKit (used in Safari and iOS browsers) [35]. According to StatCounter, the market share of these three engines is over 97%, and the market share of Chrome, Firefox, and Safari alone is over 86% [46].

Table 1 shows the browser configurations we tested for this paper. We expect most of the security header parsing and enforcement code to be deep in the browser engine code, usually untouched by derivatives of these engines. Thus, we tested each engine's flagship browser in its most stable version as of February 2, 2024: Chrome 121, Firefox 122, and Safari 17.3.1. During our initial pre-studies, we discovered no differences between the macOS and Linux browser versions regarding our tests, and we expect the same to hold for Windows. Thus, we decided to run Chrome and Firefox on Linux, where we can efficiently parallelize browser instances on our Linux cluster. Additionally, our tests behaved identically in the headless and the headful modes of browsers. Thus, we decided only to use headless versions if available, as they run more reliably in parallel.

To investigate whether security header parsing code is currently under development, we also tested the prior and following versions of Chrome and Firefox. We only tested a single version for Safari (17.3.1) as the version is bound to the operating system.

In addition to the three flagship browsers, we test Brave (Blink-derivative). This derivative explicitly states that it changes the core browser engine code to provide better security and privacy [7]. We also used Edge and other Blink derivatives during initial testing but could not discover any differences between them and Chrome, so we dropped them from the final runs.

As mobile browsers might behave differently due to performance optimizations, we test the same browsers on mobile. However, Firefox on Android users cannot permanently allow pop-ups. Thus, we instead tested the same version of Firefox Beta, where we could disable pop-up blocking, which is necessary to run our tests in an automated manner. A similar issue occurs on iPadOS, where one cannot disable manual pop-up verification in Safari. Thus, we instead tested Chrome, as all browsers on iPadOS use the underlying system WebKit [2].[1]

5.1.1 Updated Browsers. We rerun our framework several months later on the newest browser versions as of January 6, 2025: Chrome 131, Brave v1.73, Firefox 133, and Safari 18.2. We only rerun our framework on the desktop browsers as our prior results showed only minor variations between the mobile and desktop versions of the browsers. We performed this rerun to verify whether the fixes implemented in response to our bug reports are complete and to highlight that browser vendors could easily use the framework as part of their continuous integration system.

[1] Recently (iOS 17.4) Apple allowed alternative browser engines in Europe [3]. However, none is available as of August 2025.

Operating System	Browser	Versions
Ubuntu 22.04	Chrome	120, 121, 122, 131
	Firefox	121, 122, 123, 133
	Brave	v1.62.156 (Chromium 121), v1.73.101 (Chromium 131)
Android 11	Chrome	121
	Firefox Beta	123
	Brave	v1.62.165 (Chromium 121)
macOS 14.3.1	Safari	17.3.1
macOS 15.2	Safari	18.2
iPadOS 17.3.1	Chrome	122 (WebKit 17.3.1)

Table 1: List of tested browser configurations.

5.2 Tested Features

Our methodology can be used to test any header. For this work, we implemented test functions for the eight non-legacy security response headers from MDN [33]. Additionally, we implemented test functions for the five CORS headers, Timing-Allow-Origin, and Referrer-Policy, which are related to privacy and security but are not listed in the security section on MDN. Lastly, we added the Feature-Policy header, the precursor of the Permission-Policy header. From these 16 headers, we infer twelve features for which we created test functions (see Table 2 for a list of headers, features, and test functions and the number of corresponding tests).

The tested security headers can depend on the origin relations of the involved responses. An origin relation describes the relation between the origin of the top-level context A and the origin of the subsequent context B. All our test functions use one of the following inclusion chains: $A \rightarrow B$, $A \rightarrow B \rightarrow A$, $A \rightarrow B \rightarrow A \rightarrow A$. We test same-origin, same-site, and cross-site origin relations. For same-site relations, we include both a child and a parent domain to test headers such as `CSP: frame-ancestors *.project`. All tested domains are used with both HTTP and HTTPS.

Table 2 shows the implemented test functions and the number of tests that belong to them. We have a total of twelve features and 33 test functions. For example, the test function `Script Execution (CSP)/Iframe sandbox` tests the CSP script-control functionality by evaluating whether scripts are allowed to execute in a sandboxed iframe that responds with the specified CSP header. When executing the test functions, they also require a concrete response and an origin relation. The created responses are specific to each feature. We created a total of 188 basic responses and 43,416 parsing responses. By instantiating each test function with the corresponding responses and all origin relations, we create 9,992 basic response tests and 167,154 parsing response tests.

As we generated thousands of responses and running each test function for each origin relation and response would be too costly, we only run all test functions and origin relations for the basic responses. For the parsing responses, we aim to find header parsing-related differences. Thus, we only run a subset of the test functions.

Feature (Headers)	Test Function	Basic Response Tests	Parsing Response Tests
Fetch-CORS (Access-Control-, -AO, -AC, -AM, -AH, -EH)	GET simple	96	16,118
	GET custom headers	96	16,118
	GET credentials	96	16,118
	TEST custom method	96	16,118
Framing (XFO, CSP)	Iframe direct	560	17,560
	Object direct	560	-
	Embed direct	560	-
	Iframe sandbox	560	-
	Object sandbox	560	-
	Embed sandbox	560	-
	Iframe nested	560	17,560
	Object nested	560	-
	Embed nested	560	-
MIME Sniffing (XCTO)	Script direct	208	1,778
Permissions: FullscreenAPI (PP, FP)	Iframe direct	384	7,760
	Iframe child	384	-
	Iframe child allow	384	7,760
	Iframe child sandbox	384	-
PerformanceAPI (TAO)	IMG direct	208	3,202
Referrer Access (RP)	Iframe	256	5,594
	Window.Open	256	-
Script Execution (CSP)	Iframe direct	272	5,452
	Iframe sandbox	272	-
Subresourceloading (COEP)	IMG direct	176	5,274
	IMG sandbox	176	-
Subresourceloading (CORP)	IMG direct	176	6,714
	IMG sandbox	176	-
	IMG nested	176	-
	Object direct	176	-
Subresourceloading (CSP)	IMG direct	272	5,420
Upgrade (HSTS)	Direct	28	6,648
	Subdomain	28	6,648
Window References (COOP)	Window.Open	176	5,312
Total		9,992	167,154

Table 2: List of features and headers with corresponding test functions and their number of basic and parsing tests.

For example, we expect parsing of the XFO header to be independent of whether the resource is framed as `iframe` or `object`, as there is only one network parser that passes the parsed headers to the browser process. Thus, we only use `iframe` for the parsing responses. The basic responses already cover enforcement issues such as XFO not correctly applied for `object`.

5.3 Test Run Infrastructure and Settings

Running thousands of tests on various operating systems and browsers requires a robust infrastructure. We used an iterative design process to develop the test run infrastructure and determine the adequate test settings. Based on preliminary tests, we determined test page timeouts of five seconds for seven features and ten seconds for five features that require more processing time. This timeout proves a good compromise in that all tests are executed without waiting too long. For Android, we used doubled timeouts as the emulators introduced additional delays.

We require an HTTP server to serve invalid responses and template pages at all the specified origins. For that, we use a modified wptserve HTTP server [13]. Additionally, we require test runners to navigate to the test URLs on Ubuntu, macOS, Android, and iPadOS. We implement the test runners using Selenium for Ubuntu and macOS. On Android, we use a custom script that uses the Android Intent framework to open our test pages in the corresponding browsers. On iPadOS, we manually visit one coordinator website that executes all the tests in new tabs using `window.open`. In addition, we restart the browsers from time to time (e.g., after a maximum of 100 test URLs on Ubuntu) and monitor for errors while running many browsers in parallel.

It could be that two runs of the same test in the same browser do not result in the same outcome due to either noise in the test infrastructure (e.g., a timeout) or a test being inherently indeterministic. We visited each test page five times to detect such issues and perform a stability analysis. After each test URL is visited five times, we determine which tests do not have five outcomes (e.g., due to crashes in the test infrastructure) and restart them until each test has at least five outcomes in all browsers.

The tests on Ubuntu were performed on one x86_64 server with 192 CPUs. Identical servers were used to run the test on Android Emulators using Android Virtual Devices (AVD) and the Android Debug Bridge (ADB). The tests for macOS were performed on two MacBooks and one iMac, and the tests for iPadOS on two iPads.

6 Evaluation

Here, we present our framework's test run statistics and stability analysis. We present the numbers of the first run with twelve browser configurations and the second run with four browser configurations together, as the only difference were the used browsers.

6.1 Test Run Statistics

Each of the 177,146 tests was executed at least five times, and we collected between 885,730 and 1,558,656 test entries for each browser for a total of 15,010,037 test entries.

For two reasons, we collected more than the minimum number of 885,730 test results in some browsers. First, as we used intents to open URLs on Android, we cannot close the test page after execution; sometimes, the browser would re-execute an already finished test page when receiving a new intent, resulting in additional results. Second, when repeating missing tests, we revisited the original test page. During that, we also re-executed tests that already had results. As we perform majority voting, the exact number of collected outcomes for a test does not matter as long as it is high enough to identify that a test has stable results.

Only 134 test runs (<0.01%) were aborted, and another 28 test runs did not even start due to hitting the test page timeout before being executed. These low numbers show that the chosen test page timeout is adequate. Our test page runners start the following test as soon as the prior test is finished and only wait until the page timeout if necessary. Executing the basic response tests on Ubuntu with one parallel browser instance once takes approximately 40 minutes, and the parsing response tests take around 25 hours. Running all tests five times in all browsers using 50 parallel browser instances takes less than a day for Ubuntu and Android. For Safari, it is only possible to automate a single instance per device at a time. Thus, the tests on macOS and iPadOS are limited by the number of devices available, and it took us around one workweek to collect all results.

	Android			Ubuntu							iPadOS	macOS	
	Brave 1.62.165	Chrome 121	Firefox Beta 123	Brave 1.62.156	Brave 1.73.101	Chrome 120	Chrome 121-122	Chrome 131	Firefox 121-123	Firefox 133	Chrome 122/17.3.1	Safari 17.3.1	Safari 18.2
Android Brave (1.62.165)	-	88	2854	18	304	122	106	392	2868	2785	2525	2570	4011
Android Chrome (121)	88	-	2917	106	392	34	18	304	2931	2849	2589	2634	4074
Android Firefox Beta (123)	2854	2917	-	2872	2588	2951	2935	2651	14	127	2683	2626	3275
Ubuntu Brave (1.62.156)	18	106	2872	-	286	104	88	374	2858	2791	2531	2584	4029
Ubuntu Brave (1.73.101)	304	392	2588	286	-	390	374	88	2574	2507	2701	2754	3749
Ubuntu Chrome (120)	122	34	2951	104	390	-	16	302	2937	2871	2611	2664	4108
Ubuntu Chrome (121-122)	106	18	2935	88	374	16	-	286	2921	2855	2595	2648	4092
Ubuntu Chrome (131)	392	304	2651	374	88	302	286	-	2637	2571	2765	2818	3812
Ubuntu Firefox (121-123)	2868	2931	14	2858	2574	2937	2921	2637	-	129	2685	2636	3289
Ubuntu Firefox (133)	2785	2849	127	2791	2507	2871	2855	2571	129	-	2598	2567	3214
iPadOS Chrome (122/17.3.1)	2525	2589	2683	2531	2701	2611	2595	2765	2685	2598	-	61	1754
macOS Safari (17.3.1)	2570	2634	2626	2584	2754	2664	2648	2818	2636	2567	61	-	1799
macOS Safari (18.2)	4011	4074	3275	4029	3749	4108	4092	3812	3289	3214	1754	1799	-

Table 3: Pairwise comparison matrix showing the number of tests with different outcomes between all browsers excluding the test functions `Referrer Access/Window.Open` and `Subresourceloading (COEP)/IMG direct`.

6.2 Stability Analysis and Majority Voting

An important factor for the analysis of the outcomes is their stability. Do the outcomes of a test change due to noise in the test infrastructure or indeterministic behavior in the browser?

Across all repetitions, we only observed 2,074 tests (0.07%) with more than one outcome in one browser. This low percentage highlights our test infrastructure's robustness and our tests' deterministic nature. The vast majority of the non-unanimous tests are due to one indeterministic behavior on Firefox for CORP related to caching: up to 26.99% of tests have differing results for the test function `Subresourceloading (COEP)/IMG direct`.

We performed majority voting to ensure that only a single outcome for each test in each browser exists. Because most tests with more than one result had a one/four split, it is unlikely that much noise entered our results as it is unlikely that the wrong outcome was recorded in the majority of runs.

Later, during the manual analysis, we verified each discovered cluster. During this process we discovered a total of eleven tests (<0.01%, 10xCOOP, 1x HSTS) which recorded incorrect outcomes, and we changed the reported results to the ones manually verified. Additionally, we encountered the non-deterministic behavior in Firefox for `Subresourceloading (COEP)/IMG direct` and unstable behavior in our test infrastructure due to the HTTP Upgrade feature in Blink-based browsers for `Referrer Access/Window.Open`.

7 Browser Differences and Root Causes

Our goal was to find differences in security header parsing and enforcement across browsers. Many of our tests use responses that are luckily treated equally in all browsers. However, still a total of 5,606 tests (3.16%) had more than one outcome when comparing all 16 browsers configurations. There were two different outcomes for 5,485 tests, and we observed three different outcomes for 121 tests.

7.1 Test Differences

We could discover browser differences for 30 out of our 33 test functions. Table 3 shows the pairwise number of differences for 28 of the 30. We had to remove the two test functions `Referrer Access/Window.Open` and `Subresourceloading (COEP)/IMG direct` from the Table to make it easier to digest. There was some noise for the former due to the automatic upgrade-to-HTTPS functionality in Chromium-based browsers. This noise led to unstable results that we could not verify even after majority voting. The latter is inherently unstable on Firefox and thus would show random differences between the various tested Firefox versions, cluttering the Table.

When taking the bigger picture of Table 3 into account, we see a large difference between the three engines (e.g., 2,917 differences between Android Chrome (121) and Android Firefox Beta (123)) and only minor differences between the browser configurations of an engine (e.g., 106 differences between Android Chrome (121) and Ubuntu Brave (1.62.156)), with the exception of Safari 18, which had major changes in header parsing behavior compared to Safari 17.

These results indicate that code related to header parsing is not platform-, nor browser-specific but is integrated into the engine. They also show that the code is not under active development and that results are largely stable across versions.

7.2 Differences Root Cause Analysis

We analyzed all the differences to find their root causes. For the initial run of twelve browser configurations, we clustered the outcomes for each of the 30 test functions that had different outcomes using the output-browser mapping. This mapping groups all tests with the same outcomes for the same browsers. The minimum number of clusters for a test function was one (e.g., `Subresourceloading (COEP)/IMG sandbox`), and the maximum was 56 (`Subresourceloading (COEP)/IMG direct`). For the second run, we only analyzed the differences between the same browsers in the new and old versions and clustered these differences.

It is important to note that many differences are not specific to a single feature but are due to general browser differences. For example, Chrome does not allow NULL bytes in headers and resorts to network error in such cases. In addition, while many of the differences are due to the parsing of the response, others are due to general feature differences that are unrelated to the header received. For example, we discovered issues in the iframe sandbox logic in combination with CSP in both Firefox and Safari. Thus, we present

#	Title	Type	Affected Party*	Status with Hyperlinks (as of 2025-08-31)
			General Differences	
Related to header parsing				
1	LF in Header Block	Compat.		Confirmed
2	NULL in Header Name	Compat.	Fetch Standard	Confirmed
3	CR in Header Block	Compat.	Fetch Standard	Confirmed; Partially known
4	Whitespace Colon	Compat.	Fetch Standard	Confirmed
5	VT in Header Values	Compat.	,	Confirmed (), Confirmed ()
6	Empty + Non-Empty Header	Compat.		Confirmed
7+	Leading Colon	Compat.		Confirmed
8+	Leading Whitespace	Compat.		Confirmed
Not related to header parsing				
9	Status Code 300	Compat.		Known + unintended; Fixed
10	Mixed Content Images	Security	,	Known + unintended (), Known + unintended (); Fixed
11	HTTP Upgrade	Security	,	Known + intended (), Known + intended ()
12	Embed/Object URL Reliance	Compat.		Confirmed
			Feature-Specific Differences	
Related to header parsing				
13	CSP: Uppercase Scheme	Compat.		Fixed
14	CSP: Invalid Bytes	Compat.		Fixed
15	CSP: */	Compat.		Confirmed + Spec changed
16	CSP: Path in Frame-Ancestors	Security		Confirmed
17	XFO: Whitespace Everywhere	Compat.		Fixed
18	HSTS: Various Issues	Security	, ,	Fixed (), Fixed (), Confirmed ()
19	RP: FF and VT allowed	Compat.		Confirmed
20	LF in Fetch	Compat.	,	Confirmed (), Confirmed ()
21	PerformanceAPI and NULL	Compat.		Confirmed
22	NULL in Header Values (Fetch)	Compat.		Fixed
23	XCTO: Various Issues	Compat.		Known + unintended; Fixed
24+	XFO: FF allowed	Compat.	,	New (), Confirmed ()
Not related to header parsing				
25	Code 300 Cached (HSTS)	Compat.	,	Confirmed (), Fixed ()
26	CSP: Sandboxed Frames FA	Security		Confirmed
27	CSP: Sandboxed Frames 'self' Bypass	Security		Confirmed
28	CSP: Sandboxed Frames *.origin	Security		Confirmed
29	FP Header not supported	Compat.	,	Known + intended
30	PP Header not supported	Security	,	Known + intended
31	TAO and 302	Compat.		Fixed
32	Mixed-Content performanceAPI	Compat.		Confirmed
33	CORP and Object	Compat.		Known + unintended
34	RP Safer-Defaults	Privacy	, ,	Known + intended (,), Known + intended ()
35	RP Safer-Defaults Exception Top-Level	Privacy		Known + intended; Missing documentation
36	RP Safer-Defaults Exception Same-Site	Privacy	,	Known + intended (), Known + intended (); Might change
37	COEP Secure-Context	Security		Confirmed
38	CORP Random Caching	Security		Confirmed
39	Download Window Reference	Compat.	All	Different default settings for download behavior
40	Download Behavior Difference	Compat.	(Mobile only)	Confirmed
41	204 Not About:Blank	Compat.	(Mobile only)	Confirmed
42+	HSTS Race Condition	Security		Confirmed

+ Root cause discovered in the second run with the four new browser configurations

* All root causes affecting Chrome also affect Brave

Table 4: Identified root causes across two dimensions discovered in the initial run and the second run.

the results along two axes: whether an issue is generic or feature-specific and whether or not an issue relates to header parsing.

Table 4 shows all 42 difference *root causes* we identified while analyzing all clusters grouped by the four categories. 38 of them were identified in the initial run, and four more were identified in the second run with the new browser versions. 31 of the 42 root causes were previously unknown (see status column), and we reported 36 bugs to affected browsers and specifications.

The Table includes a short descriptive name, an impact type, the affected party, and the issue's status. We always attributed the differences to the browsers that act against the current specifications, and if this was not possible against the specification itself. The status for newly discovered issues can be *new*, *confirmed*, i.e., we submitted a bug report that got acknowledged, or *fixed*, i.e., additionally, the browser vendor already changed their codebase and fixed this issue. Some of the discovered issues were already *known* to the vendors; here, we distinguish between *intended* differences, such as Brave intentionally not supporting privacy-invasive referrer policies, and *unintended* differences, i.e., the browser vendor was already aware of the issue but did not fix it yet.

7.3 Browser Differences

In the following, we first list the differences between browser configurations belonging to the same engine in the first run. Then, we

list the additional differences discovered by our rerun. Concrete example root causes are discussed in Section 7.4.

7.3.1 First Run. **Chrome 120 vs. Chrome 121:** The 16 differences are all due to incorrect parsing of XCTO headers prior to Chrome 121 (root cause 23).
Brave vs. Chrome: All differences are caused by the fact that Brave does not support privacy-invasive values such as `unsafe-url` for Referrer-Policy (root cause 34).
Safari desktop vs. Chrome iPadOS: Most differences (52/62) are caused by Chrome on iPadOS setting a stricter mixed-content preference (root cause 10). Additionally, some differences are caused by varying download behavior between WebKit mobile and desktop and a parsing difference for status code `204` (root causes 40 and 41).
Desktop vs. Android (Brave, Chrome, Firefox): The differences are due to non-automated downloads in the case a file with the same name already exists on Mobile (root cause 39) and to caching of status code `300` for HSTS (root cause 25).

7.3.2 Updated Browser Versions. Following our rerun, we compared each browser configuration with its previous version and report on the results here. In general, most differences are due to fixes to our bug reports following the initial run; however, some other changes exist.
Chrome 131 vs. Chrome 122: All the 286 differences are due to fixes for our bug reports for root causes 13 and 18.
Firefox 133 vs. Firefox 123: The 129 differences are due to fixes for root causes 17 and 18. Additionally, root causes 9 and 10 were fixed (known previously, no new bug reports by us), and the behavior of root causes 32 and 39 changed.
Safari 18.2 vs. Safari 17.3.1: Safari changed significantly and has 1,799 differences compared to the earlier version. Some changes are due to fixes to our reports for root causes 31 and 22. Additionally, blocking of mixed content images (root cause 10) was implemented, and its version of automated HTTP Upgrades was implemented (root cause 11). However, most differences are due to new general header parsing differences. For root cause 4, Safari changed the behavior from Chrome's to Firefox's. For root cause 3 and the new one *leading colon* (root cause 7), Safari now throws network errors instead of only ignoring such invalid rows. Additionally, Safari now accepts *leading whitespace* (root cause 8). Lastly, we discovered one race condition in the HSTS implementation of Safari responsible for many differences (root cause 42).
Firefox 133 vs. Safari 18.2/Chrome 131: Due to the fix of root cause 17, we discovered a new difference in whitespace behavior where earlier all engines behaved incorrectly and Firefox is now following the specifications (root cause 24).

7.4 Case Studies

The following presents root cause case studies across the four groups and their potential impact.

7.4.1 General Differences. The root causes in the general header parsing section apply to all or many headers. The first four and root causes 7 and 8 are triggered by severely broken HTTP messages, such as including a NULL byte in a header name. While such responses do not conform to the HTTP specification [15], it is usually unclear how browsers should handle such responses. Thus, we reported several bugs to the specification writers to clarify those ambiguities such that in the future they can be tested by usual expected-outcome tests in WPT. For example, Chrome decided to treat responses with NULL bytes in header names as network errors, whereas other browsers ignore only the malformed header. We added our findings to existing specification issues and tried reviving the discussion to specify browser behavior for HTTP/1.1 parsing of severely broken responses. The sixth issue is that Firefox incorrectly parses responses containing two headers of the same name if one is empty. Firefox ignores the empty header even though for headers defined as a single field such as CORP, COOP, or ACAO, such occurrences should be treated as an invalid list. A developer using Firefox and incorrectly sending two headers could believe their site is adequately protected. However, most Web users using other browsers would not be protected.

In the non-parsing section, we mainly discovered differences due to Mixed-Content and the new HTTP Upgrade feature. Firefox and Safari had yet to implement Level 2 of the mixed content specification [18, 1] in the initially tested versions and allowed mixed passive content instead of automatically upgrading it. The HTTP Upgrade feature in Chrome goes even further and automatically tries to upgrade every top-level HTTP request, only allowing HTTP if the HTTPS request fails. While some sites might break by stricter mixed-content settings, the benefits of only allowing HTTPS on the Web are positive for security and privacy. Thus, it is good that all browsers want to align to more aggressive settings here.

7.4.2 Feature-Specific Differences. In the feature-specific header parsing section, we discovered differences due to whitespace, casing, or non-ASCII letters in header values. For most of these, the specification clearly stated the correct behavior, and it was possible to detect the non-conforming browsers and report the issue to them. Several of the reports are already fixed. If developers use such invalid values in one of the affected browsers, they might incorrectly conclude that their header is correct and working, leaving users of other browsers unprotected.

A particular case to highlight is HSTS, where we found many differences between the browsers and divergences from the specifications for all browsers. A small number of these cases were previously discovered by Siewert et al. [45], but the browsers took no action. Our reports already caused Chrome engineers to rewrite their HSTS parsing implementation from scratch to adhere more closely to the specifications. Currently, no tests for HSTS exist in WPT due to fear of HSTS contaminating other test results [52]. That there are no cross-browser HSTS testing efforts seems to be a central factor in why we could detect many issues related to HSTS. Thus, in addition to reporting the individual issues to the browsers, we also suggested reconsidering how to run HSTS tests to the WPT maintainers. As invalid HSTS headers, such as duplicate headers, are not uncommon in the wild [41], browser divergence here can lead to users being vulnerable to network attackers.

Another case to highlight is root cause 15. We reported it to Chrome, as Chrome was not following the specifications. The Chrome developer assigned to the bug report was also an author of the specification and decided that the behavior of Chrome is preferable. Thus, instead of fixing it in Chrome, they changed the specification.

Most of the discovered differences in the last category relate to CSP enforcement and Referrer-Policy implementations. For CSP, several differences relate to sandboxed frames. Sandboxed frames should have an opaque origin that should not match anything. As we discovered three issues in two browsers here, sandboxing seems hard to implement. A CSP frame-ancestor of `*` should not match a sandboxed frame in the ancestor chain, which is not the case in Firefox. More severely, Safari has a major issue with sandboxed frames that themselves set a CSP. If the CSP allows scripts from `'self'`, Safari matches every possible origin instead of only allowing scripts from the origin specified in the src attribute, making it possible to bypass the CSP. Additionally, CSP patterns of the format `*.origin` do not work in Safari, potentially leading developers to deploy overly broad CSPs increasing the attack surface.

For the Referrer-Policy, it is known that privacy-friendly browsers such as Brave, Firefox, and Safari do not support privacy-invasive values such as `unsafe-url`. Brave additionally does not support values such as `origin`. We also discovered two undocumented exceptions in Firefox and Safari. In Firefox, the privacy-invasive values are only turned off for subresource loads such as iframes or images; they are still supported for user-initiated clicks on links and `window.open`. According to Mozilla, this is an intended behavior as it is required for web compatibility. However, they agreed that such exceptions should be explained in their documentation on MDN. A second exception is that insecure values are supported for same-site origins in Firefox and Safari, which has also not been documented. Mozilla answered that this is the intended behavior, whereas Apple replied that this was their intended behavior, but they are considering changing it.

8 Discussion

In this section, we first discuss the limitations and future extensions of our work, then summarize the main insights, and finish with ethics considerations of our work.

8.1 Limitations and Future Extensions

A core limitation of our work is that we only tested the HTTP/1.1 implementations of browsers. Additional issues could be hidden in the HTTP/2 or HTTP/3 implementations. The WPT HTTP/2 implementation does not allow sending malformed responses, such as multiple headers of the same name, and does not contain a general-purpose HTTP/3 server. To the best of our knowledge, no HTTP/2 or HTTP/3 implementation intentionally allows for invalid responses. A future extension would be to implement an HTTP/2 and HTTP/3 server that allows the sending of invalid responses to test these protocols. However, as HTTP/2 and HTTP/3 are specified in a much stricter manner, we do not expect many differences in the general parsing implementations and expect the feature-specific differences to remain the same.

Another limitation is that extending our work to future browser configurations and headers requires some human effort. We note that creating the tests is a one-time effort for each feature. A domain expert can easily create new tests if browsers decide to add new security headers in the future, which they usually do at a rate of less than one header per year. The analysis pipeline automatically outputs clusters that have to be condensed down to root causes. As many clusters were very similar, this process only took a couple of hours in the initial full comparison. When testing new versions of existing browsers, it is enough to compare the changes between the versions of the same browser; no full comparison across all browser configurations is necessary, and this took on average 30 minutes for each tested new browser version.

Lastly, the chosen approach can have false negatives as it cannot detect that all browsers equally misbehave.

8.2 Main Insights

Even though frameworks such as WPT have existed for years, our testing framework enabled us to detect many new differences between browsers in their header parsing and enforcement behavior caused by 42 root causes, out of which 31 were previously unknown. In the following, we discuss the implications of these differences and how to get rid of them in the future.

Foremost, the results show that compatibility between browsers engines is still far from perfect due to implementation mistakes and intentional differences. Browser differences can annoy, confuse, and mislead developers and, in the worst case, result in security and privacy issues for web users. As discussed in our threat model section, this implies that users of a website may get a degraded level of security if they use a browser that behaves differently from the browsers that the web developers tested the website on. If a developer implements a security header incorrectly, e.g., uses `SAME ORIGIN` instead of `SAMEORIGIN`, and only tests for the feature in Firefox, they might conclude that the feature works, leading to the vast majority of users using other browsers that ignore this value and not apply the XFO defense being unprotected. We discovered that at least two sites in the crawler.ninja dataset [25] use an X-Frame-Options header value of `SAME ORIGIN` on their landing page, and many more use other invalid values. In another vein, if a developer uses a correct value (according to the specifications) that is erroneously not supported in their used browser, they might assume the header is non-functional and abort using the security mechanism and deploy a website not protected in any browser.

Other compatibility issues can lead to broken code or functionality issues. For example, responses with an invalid linefeed in their header block do not load in Safari. As users on iOS have to use WebKit-based browsers [2], some legacy websites might be unusable for them. Differences in Referrer-Policy implementations and different default values result in various privacy levels for users of different browsers. Some browsers actively promote that they are using better defaults and users expect that to be the case. Thus, these users could be upset as several undocumented privacy exceptions exist in Firefox and Safari.

Additionally, several discovered issues are exploitable by any web attacker. For example, consider the CSP bypass for sandboxed frames in Safari. An attacker can frame a site that has an HTML injection and a CSP with `'self'` in the script-src and is otherwise secure and can bypass it and perform XSS on the frame, which is similar to CVE-2023-38592 discovered by Bernardo et al. [5].

However, the blame for differences should not only be on browser vendors. Our tests did not only reveal implementation mistakes, but they also revealed several instances in which the specifications are currently unclear or intentionally allow browsers to choose

how to behave. For some of these instances, we have seen related issue reports against the specifications opened more than five years ago without concluding. We hope the additional information we provided in these issues reignites the discussion and results in better and more concrete specifications and consequentially in consistent protections across browsers.

Most importantly, we have shown that current expected-outcome browser testing is not enough and that differential browser testing is necessary to thoroughly test security header implementations. Even though WPT contains several thousand tests, we discovered many differences that they did not cover. In an extreme case, WPT does not contain any tests for HSTS due to an architectural design issue. While our tests use the WPT infrastructure, they cannot be directly added to WPT due to their differential nature and the fact that they do not define the expected outcome, which is required for WPT. However, prototype instances of each root cause can be converted to standard WPT tests after deciding what is the expected outcome, and our reports have already resulted in many tests being added to WPT by browser vendors and us including a major effort to test and define browser parsing of highly invalid HTTP/1.1 responses.

Finally, we encourage browser vendors to integrate our framework into their development cycle. Using the analysis results vendors can verify that their bug fixes are complete and correct. Additionally, they can uncover new root causes in other browsers if after an update differences where previously all browsers misbehaved are now detectable.

8.3 Ethics Considerations

In our research, we discovered vulnerabilities and other issues in browsers and disclosed them to the affected vendors right after we identified them following the principles described in the Menlo Report [28]. For each discovered root cause, we searched for whether the issue had already been reported or was otherwise known, such as being described in a release note. We only reported the issues for which we could not find information to keep work for the bug triagers minimal. We used the responsible disclosure procedures available for each browser vendor for issues with direct security implications. We reported the issues with only minor or compatibility impacts as functionality bugs to avoid unnecessary work for the security teams. Additionally, all bug reports contained links to URLs exhibiting the reported behavior, enabling analysts to reproduce the issues without overhead easily.

The tested browsers and responses are entirely under our control in our internal network. We did not perform any requests to unrelated websites (live systems) and thus cannot cause harm by too much load or invalid responses.

9 Related Work

Here, we survey related work in the areas of security headers, differential browser testing, and mobile browsers.

9.1 Security Headers in Browsers and the Web

The adoption of security headers in the wild such as CSP or HSTS, has been widely measured both academically and non-academically [25, 51, 42, 30, 8, 20]. Many of these works also showed that developers have trouble using security headers and real-world headers often do not provide adequate protection or are straight up invalid.

Related to our work, Calzavara et al. [9] and Siewert et al. [45] studied browser implementation differences and specification issues in various security headers such as CSP, XFO, and HSTS. Our approach diverges by not starting with the specifications and a small set of handcrafted tests. Instead, we employ differential testing to extensively examine browser parsing behavior. Furthermore, we cover a broader range of headers. Specifically, we developed a response mutation algorithm and introduced invalid values such as newlines, NULL bytes, and inserted spaces. Additionally, we propose a methodology for analyzing the root causes of differences using outcome-browser mappings, enabling us to uncover numerous new root causes of browser discrepancies.

9.2 Browser Testing

Various works tested security features in browsers. Hothersall-Thomas et al. created a testing suite for many different features [27]. Schwenk et al. compared the SOP in browsers [43]. Nguyen et al. compared caching in browsers [34]. Other works showed differences in regards to Cookies, CSP, CORB, and more [55, 44, 29, 5].

Similar to our work, Wi et al. used differential testing to find CSP enforcement differences for script execution [55]. In contrast to them, we are not focusing on the correct enforcement of valid headers but instead investigate parsing issues for invalid headers. Additionally, we cover twelve features and not only one feature (CSP script-execution).

Apart from scientific research, the most important project in browser testing is the cross-browser web-platform-tests (WPT) project that maintains an extensive collection of tests that are regularly run against major browsers [12].

We based our framework on them. The main difference between our project and the WPT tests is that they require defining the correct behavior up-front. In contrast, we record differences in behavior and later analyze the results.

9.3 Mobile Browser Security

Luo et al. performed a longitudinal analysis on the support of security mechanisms in mobile browsers from 2011 to 2018 and discovered many issues [31]. More recently, mobile browsers were analyzed in their privacy behaviors [56, 37].

While these works discovered many differences between Android and desktop browsers, Pradeep et al. revealed that the vast majority of Android browsers in 2023 are WebView browsers [37] that nowadays automatically update via Google System Services [21]. We showed that mobile browsers nowadays behave almost identically to their desktop counterparts of the same version in the case of security headers.

10 Conclusion

Modern websites deploy a plethora of security headers to enable protection against various attacks. Consistent implementations of security header parsing and enforcement code across browsers are required. Otherwise, users' security and privacy depend on their

chosen browser, and compatibility issues could lead developers to opt out of deploying security headers altogether.

By running 177,146 tests for twelve features and 16 headers in 16 browser configurations, we observed 5,606 tests (3.16%) with a different outcome across the browsers, which we traced back to 42 root causes out of which 31 were previously unknown. These results show that the current amount of cross-browser testing is insufficient, and many browser differences concerning security headers exist. In the context of this work, we reported dozens of bug reports to browser vendors, specifications, and shared testing platforms. We hope to improve the state of security header parsing and enforcement across browsers going forward such that all users are consistently protected.

Four of the root causes were only discovered by rerunning our framework several months later on newer browser versions. Performing such a rerun of our framework involves little manual involvement, and we encourage browser vendors to integrate it into their development workflow.

Availability

We are committed to open science and made our full source code, including our response generation, testing pipeline, and analysis scripts available at Zenodo: https://zenodo.org/records/16890358 [40]. Additionally, we released the collected dataset at Zenodo: https://zenodo.org/records/16996058 [39]. We also contributed many tests and several other changes to the open-source WPT project [12].

Acknowledgments

We thank the reviewers for their valuable feedback. This work was conducted in the scope of a dissertation at the Saarbrücken Graduate School of Computer Science. This paper uses icons from Font Awesome, which is licensed under the Creative Commons Attribution 4.0 International License. Font Awesome icons can be found at: https://fontawesome.com/.

References

[1] Apple. 2022. 247197 – Upgrade requests in mixed content settings. https://bugs.webkit.org/show_bug.cgi?id=247197.

[2] Apple. 2024. App Review Guidelines. Apple Developer. https://developer.apple.com/app-store/review/guidelines/.

[3] Apple. 2024. Apple announces changes to iOS, Safari, and the App Store in the European Union. Apple Newsroom. https://www.apple.com/newsroom/2024/01/apple-announces-changes-to-ios-safari-and-the-app-store-in-the-european-union/.

[4] Apple. 2019. Preventing Tracking Prevention Tracking. WebKit. https://webkit.org/blog/9661/preventing-tracking-prevention-tracking/.

[5] Pedro Bernardo, Lorenzo Veronese, Valentino Dalla Valle, Stefano Calzavara, Marco Squarcina, Pedro Adão, and Matteo Maffei. 2024. Web Platform Threats: Automated Detection of Web Security Issues With WPT. In *USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity24/presentation/bernardo.

[6] Mike Bishop. 2022. HTTP/3. Request for Comments RFC 9114. Internet Engineering Task Force. doi:10.17487/RFC9114.

[7] Brave. 2025. Deviations from Chromium (features we disable or remove). GitHub. https://github.com/brave/brave-browser/wiki/Deviations-from-Chromium-(features-we-disable-or-remove).

[8] William J. Buchanan, Scott Helme, and Alan Woodward. 2018. Analysis of the adoption of security headers in HTTP. *IET Information Security*, 2. doi:10.1049/iet-ifs.2016.0621.

[9] Stefano Calzavara, Sebastian Roth, Alvise Rabitti, Michael Backes, and Ben Stock. 2020. A Tale of Two Headers: A Formal Analysis of Inconsistent Click-Jacking Protection on the Web. In *USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity20/presentation/calzavara.

[10] WPT Contributors. 2024. Testharness.js. https://web-platform-tests.org/writing-tests/testharness-api.html.

[11] WPT Contributors. 2025. Web Platform Tests. https://github.com/web-platform-tests/wpt.

[12] WPT Contributors. 2024. Web-Platform-Tests. https://web-platform-tests.org/.

[13] WPT Contributors. 2024. Wptserve. https://web-platform-tests.org/tools/wptserve/docs/index.html.

[14] Dave Crocker and Paul Overell. 2008. Augmented BNF for Syntax Specifications: ABNF. Request for Comments RFC 5234. Internet Engineering Task Force. doi:10.17487/RFC5234.

[15] Roy T. Fielding, Mark Nottingham, and Julian Reschke. 2022. HTTP Semantics. Request for Comments RFC 9110. Internet Engineering Task Force. doi:10.17487/RFC9110.

[16] Roy T. Fielding, Mark Nottingham, and Julian Reschke. 2022. HTTP/1.1. Request for Comments RFC 9112. Internet Engineering Task Force. doi:10.17487/RFC9112.

[17] Firefox. 2019. 1531012 - (permissions-policy) [meta] Permissions Policy tracking. https://bugzilla.mozilla.org/show_bug.cgi?id=1531012.

[18] Firefox. 2023. 1811787 - Ship Mixed Content Level 2 upgrading of passive mixed content. https://bugzilla.mozilla.org/show_bug.cgi?id=1811787.

[19] Gertjan Franken, Tom Van Goethem, Lieven Desmet, and Wouter Joosen. 2023. A Bug's Life: Analyzing the Lifecycle and Mitigation Process of Content Security Policy Bugs. In *USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity23/presentation/franken.

[20] Gertjan Franken and Vik Vanderlinden. 2024. The 2024 Web Almanac: Security. 11. HTTP Archive. https://almanac.httparchive.org/en/2024/security.

[21] Google. 2024. Google System Release Notes. https://support.google.com/product-documentation/answer/14343500?hl=en#.

[22] HackerOne. 2024. Top Ten Vulnerabilities. Hacker-Powered Security Report. http://hackerpoweredsecurityreport.com/the-top-ten-vulnerabilities/.

[23] Robert Hansen and Jeremiah Grossman. 2008. Clickjacking. https://www.sectheory.com/clickjacking.htm.

[24] Florian Hantke and Ben Stock. 2022. HTML violations and where to find them: a longitudinal analysis of specification violations in HTML. In *ACM Internet Measurement Conference*. doi:10.1145/3517745.3561437.

[25] Scott Helme. 2024. Crawler.Ninja. https://crawler.ninja/.

[26] Jeff Hodges, Collin Jackson, and Adam Barth. 2012. HTTP Strict Transport Security (HSTS). Request for Comments RFC 6797. Internet Engineering Task Force. doi:10.17487/RFC6797.

[27] Charlie Hothersall-Thomas, Sergio Maffeis, and Chris Novakovic. 2015. BrowserAudit: Automated Testing of Browser Security Features. In *International Symposium on Software Testing and Analysis*. doi:10.1145/2771783.2771789.

[28] Erin Kenneally and David Dittrich. 2012. The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research. *SSRN Electronic Journal*. doi:10.2139/ssrn.2445102.

[29] Soheil Khodayari and Giancarlo Pellegrino. 2022. The state of the SameSite: Studying the usage, effectiveness, and adequacy of SameSite cookies. In *IEEE Symposium on Security and Privacy*. doi:10.1109/SP46214.2022.9833637.

[30] Arturs Lavrenovs and F. Jesús Rubio Melón. 2018. HTTP security headers analysis of top one million websites. In *International Conference on Cyber Conflict*. doi:10.23919/CYCON.2018.8405025.

[31] Meng Luo, Pierre Laperdrix, Nima Honarmand, and Nick Nikiforakis. 2019. Time Does Not Heal All Wounds: A Longitudinal Analysis of Security-Mechanism Support in Mobile Browsers. In *Network and Distributed System Security Symposium*. doi:10.14722/ndss.2019.23149.

[32] Microsoft. 2009. Happy 10th birthday Cross-Site Scripting! https://learn.microsoft.com/en-us/archive/blogs/dross/happy-10th-birthday-cross-site-scripting.

[33] Mozilla Developer Network. 2024. HTTP headers - Security. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers#security.

[34] Hoai Viet Nguyen, Luigi Lo Iacono, and Hannes Federrath. 2018. Systematic Analysis of Web Browser Caches. In *International Conference on Web Studies*. doi:10.1145/3240431.3240443.

[35] David Nield. 2022. Which Browser Engine Powers Your Web Browsing and Why Does It Matter? Gizmodo Australia. https://gizmodo.com.au/2022/08/which-browser-engine-powers-your-web-browsingand-why-does-it-matter/.

[36] Mark Nottingham and Poul-Henning Kamp. 2021. Structured Field Values for HTTP. Request for Comments RFC 8941. Internet Engineering Task Force. doi:10.17487/RFC8941.

[37] Amogh Pradeep, Álvaro Feal, Julien Gamba, Ashwin Rao, Martina Lindorfer, Narseo Vallina-Rodriguez, and David Choffnes. 2023. Not Your Average App: A Large-scale Privacy Analysis of Android Browsers. *Proceedings on Privacy Enhancing Technologies*, 1. doi:10.56553/popets-2023-0003.

[38] Jannis Rautenstrauch. 2024. XFO ignores whitespace everywhere (not only leading and trailing whitespace). https://bugzilla.mozilla.org/show_bug.cgi?id=1891467.

[39] Jannis Rautenstrauch, Trung Tin Nguyen, Karthik Ramakrishnan, and Ben Stock. 2025. Dataset for: Head(er)s Up! Detecting Security Header Inconsistencies in Browsers. https://zenodo.org/records/16996058.

[40] Jannis Rautenstrauch, Trung Tin Nguyen, Karthik Ramakrishnan, and Ben Stock. 2025. Software for: Head(er)s Up! Detecting Security Header Inconsistencies in Browsers. https://zenodo.org/records/16890358.

[41] Jannis Rautenstrauch and Ben Stock. 2024. Who's Breaking the Rules? Studying Conformance to the HTTP Specifications and its Security Impact. In *ACM AsiaCCS*. doi:10.1145/3634737.3637678.

[42] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. 2020. Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies. In *Network and Distributed System Security Symposium*. doi:10.14722/ndss.2020.23046.

[43] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. 2017. Same-Origin Policy: Evaluation in Modern Browsers. In *USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schwenk.

[44] Chaofan Shou, Ismet Burak Kadron, Qi Su, and Tevfik Bultan. 2021. CorbFuzz: Checking browser security policies with fuzzing. In *IEEE/ACM International Conference on Automated Software Engineering*. https://ieeexplore.ieee.org/abstract/document/9678636/.

[45] Hendrik Siewert, Martin Kretschmer, Marcus Niemietz, and Juraj Somorovsky. 2022. On the Security of Parsing Security-Relevant HTTP Headers in Modern Browsers. In *SecWeb*. doi:10.1109/spw54247.2022.9833880.

[46] statcounter. 2024. Browser Market Share Worldwide. StatCounter Global Stats. https://gs.statcounter.com/browser-market-share.

[47] Chromium team. 2025. HSTS Preload List Submission. https://hstspreload.org/.

[48] Martin Thomson and Cory Benfield. 2022. HTTP/2. Request for Comments RFC 9113. Internet Engineering Task Force. doi:10.17487/RFC9113.

[49] W3C. 2024. Content Security Policy Level 3. W3C Working Draft. https://www.w3.org/TR/CSP3/.

[50] W3C. 2024. Content Security Policy Level 3: Frame-Ancestors. W3C Working Draft. https://w3c.github.io/webappsec-csp/#frame-ancestors.

[51] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. 2016. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *ACM SIGSAC Conference on Computer and Communications Security*. doi:10.1145/2976749.2978363.

[52] Mike West. 2017. How to test origin-wide impacts of things like HSTS? · Issue #4880. https://github.com/web-platform-tests/wpt/issues/4880.

[53] WHATWG. 2024. HTML Standard X-Frame-Options. https://html.spec.whatwg.org/multipage/document-lifecycle.html#the-x-frame-options-header.

[54] WHATWG. 2025. MIME Sniffing Standard. https://mimesniff.spec.whatwg.org/.

[55] Seongil Wi, Trung Tin Nguyen, Jihwan Kim, Ben Stock, and Sooel Son. 2023. DiffCSP: Finding Browser Bugs in Content Security Policy Enforcement through Differential Testing. In *Network and Distributed System Security Symposium*. doi:10.14722/ndss.2023.24200.

[56] Ahsan Zafar and Anupam Das. 2023. Comparative Privacy Analysis of Mobile Browsers. In *ACM Conference on Data and Application Security and Privacy*. doi:10.1145/3577923.3583638.