Web Platform Threats: Automated Detection of Web Security Issues With WPT

Pedro Bernardo[*†], Lorenzo Veronese[*†], Valentino Dalla Valle[‡],
Stefano Calzavara[‡], Marco Squarcina[†], Pedro Adão[§], Matteo Maffei[†]
[†] *TU Wien*
[‡] *Università Ca' Foscari Venezia*
[§] *Instituto Superior Técnico, Universidade de Lisboa, and Instituto de Telecomunicações*

Abstract

Client-side security mechanisms implemented by Web browsers, such as cookie security attributes and the Mixed Content policy, are of paramount importance to protect Web applications. Unfortunately, the design and implementation of such mechanisms are complicated and error-prone, potentially exposing Web applications to security vulnerabilities. In this paper, we present a practical framework to formally and automatically detect security flaws in client-side security mechanisms. In particular, we leverage Web Platform Tests (WPT), a popular cross-browser test suite, to automatically collect browser execution traces and match them against Web invariants, i.e., intended security properties of Web mechanisms expressed in first-order logic. We demonstrate the effectiveness of our approach by validating 9 invariants against the WPT test suite, discovering violations with clear security implications in 104 tests for Firefox, Chromium and Safari. We disclosed the root causes of these violations to browser vendors and standard bodies, which resulted in 8 individual reports and one CVE on Safari.

1 Introduction

Writing secure Web applications is notoriously hard, due to the heterogeneity, complexity and open-ended nature of the Web. To mitigate the challenges of secure Web application development, browsers integrate a growing list of client-side security mechanisms to assist Web developers. Examples of such mechanisms include cookie security attributes (HttpOnly, Secure and SameSite), security headers like Origin and Sec-Fetch-Data, mechanisms to secure mixed content (e.g., to avoid that HTTPS-served webpages fetch content in clear over HTTP), and sophisticated client-side protection mechanisms like Content Security Policy (CSP).

The design of such mechanisms is very delicate, as witnessed by the long list of design shortfalls (e.g., unexpected interactions with other browser components) or implementation flaws, which led to breaking well-established Web security invariants [18, 37]. *Formal methods proved to be an essential tool to rigorously analyze client-side security mechanisms*, allowing for the identification of bugs and formulation of formal security proofs in such a complex environment. All state-of-the-art techniques, however, be they manual [24], machine-checked [21], or automated [18, 37], apply to *browser models*, which suffer from two fundamental drawbacks. First, client-side security mechanisms evolve over time and new ones are being proposed on a regular basis, which makes browser models extremely hard to maintain. Second, even if specifications are correct, security-critical bugs often affect the implementations [27, 33, 34, 40]. Correctly integrating client-side security mechanisms within browsers is challenging and error-prone for various reasons. Browsers are incredibly complicated software artifacts: for instance, the Chromium codebase contains roughly 35 million lines of code, i.e., it is larger than the Linux kernel. Furthermore, browser vendors are required to translate natural language specifications, e.g., from the World Wide Web Consortium (W3C), into new code to be pushed into an already complicated codebase. Even worse, client-side security mechanisms often cannot be specified in isolation: most of them interact with core browser components like Fetch, which defines requests, responses, and the process which eventually binds them. This means that the implementation of client-side security mechanisms often requires changes to existing browser components which were not developed with such an integration in mind.

We thus tackle the following research question: *can we design a practical framework to formally and automatically detect security flaws in the implementation of client-side security mechanisms?*

In this paper, we answer in the affirmative, putting forward a novel, formally-grounded and lightweight technique. In particular, we leverage existing community efforts in the development of Web Platform Tests (WPT) [16], a cross-browser test suite designed to give browser vendors confidence that they are shipping software which is compliant with specifications and compatible with other implementations. WPT includes

[*] Shared first authorship

more than 50K tests covering a wide range of browser components, including Web security mechanisms, thus representing the largest benchmark of the intended browser behavior to date. Our approach consists in abstracting the test executions into sets of *traces* (i.e., sequences of relevant browser events), which are then matched against *Web security invariants* (i.e., intended security properties expressed in first-order logic). This way, we automatically identify traces breaking important security properties and thus pinpoint browser behaviors requiring immediate attention by browser vendors, due to their clear security implications. Furthermore, WPT is continuously updated as Web standards and new features are introduced to the Web platform, which makes our verification pipeline automatically applicable to the latest browser versions.

Contributions. More concretely, we contribute as follows:

- We formalize 9 Web invariants regarding core components of the Web platform such as Cookies and Mixed Content, encoding them in first-order logic to allow for efficient verification of browser execution traces using an automated theorem prover (Sec. 3).
- We present an automated pipeline designed to identify security-critical inconsistencies in browser implementations. Our approach leverages the WPT test suite to acquire browser execution traces, which are then matched against Web security invariants in order to identify any traces that violate Web security properties (Sec. 4).
- We demonstrate the effectiveness of our approach by validating our 9 invariants against the WPT test suite, discovering violations with clear security implications in 104 tests (Sec. 5). In particular, we discuss 10 attacks against Chromium, Firefox, and Safari concerning cookies and Mixed Content policy violations (Sec. 5.2). We responsibly disclosed all the new findings to affected browser vendors and standard bodies, which resulted in 8 individual reports and one CVE on Safari.

We publish all the artifacts developed during this research, including the definition of the Web invariants in SMT-LIB format, and our trace verification pipeline [20].

2 Background

We assume familiarity with the basic functionality of the Web platform, e.g., the HTTP protocol, HTML and JavaScript.

Web Security Primer. The traditional threat model of Web applications considers both *Web attackers* and *network attackers* [22]. A Web attacker is the owner of a malicious host, which is used to mount attacks against other Web applications. Traditional examples of Web attacks include Cross Site Scripting (XSS) and Cross Site Request Forgery (CSRF). A network attacker extends the capabilities of a Web attacker with full control of the network traffic, i.e., everything which is unencrypted can be read and modified by a network attacker. Encryption can be enforced through the use of HTTPS, which provides a secure transport protocol for the Web. Browsers rely on the notion of *secure context* to identify pages satisfying minimal confidentiality and integrity requirements [38].

The baseline defense mechanism of Web browsers is the *Same Origin Policy* (SOP), which is intended to enforce the intuitive invariant that content owned by a Web application should not be read or written by other Web applications. The notion of *origin* defines the security perimeter of SOP: an origin is a triple including a scheme (HTTP, HTTPS...), a host (e.g., www.foo.com) and a port (defaulting to 80 for HTTP and 443 for HTTPS). This way, a Web page at https://evil.com cannot access content served by https://foo.com. Since the fine-grained isolation of SOP is too restrictive for specific settings, another common Web security concept is the notion of *site*, i.e., one domain part plus the *effective top-level domain* as defined in the Public Suffix List [30] – also called *registrable domain* or *eTLD+1*. For example, foo.com and foo.github.io (as github.io is in the PSL) are sites, and a.foo.com and b.foo.com are two subdomains of the same site foo.com. Although https://a.foo.com and https://b.foo.com are two different origins, their same-site position might relax some security checks enforced by browsers (see below). The W3C Secure Contexts specification [38] also defines the notion of *potentially trustworthy* origins as those that the browser can trust sending data securely. In particular, in addition to origins whose protocol is `https` or `wss`, the localhost IP address and all subdomains of `localhost` are considered potentially trustworthy even for unencrypted connections.

Cookies. Cookies are a client-side storage mechanism based on the name-value paradigm and can be set through JavaScript or using the `Set-Cookie` header of HTTP responses. In their default configuration, cookies are accessible by JavaScript using the `document.cookie` property and are attached by the client to all the requests sent to the host which set them, using the `Cookie` header. The scope of cookies can be extended to other subdomains by using the `Domain` attribute; this allows cookie sharing across sibling domains, e.g., a.foo.com can set cookies with the `Domain` attribute set to foo.com, which makes them available to b.foo.com. Since cookies may store sensitive data, e.g., session identifiers that must be protected to prevent session hijacking, clients offer a plethora of defensive options deployed in terms of cookie *attributes* and *prefixes*.

Cookies marked with the `Secure` attribute are only attached to requests sent over secure channels, e.g., over HTTPS, which is important to ensure their confidentiality against network attackers. The `HttpOnly` attribute makes cookies inaccessible to JavaScript, which is useful to prevent cookie theft in the presence of injection vulnerabilities like XSS. Finally, the `SameSite` attribute can be used to restrict the attachment of cookies to same-site requests, thus mitigating CSRF. If `SameSite` is set to `Strict`, no cross-site request will ever attach the cookie; if `SameSite` is set to `Lax`, top-level navigation

requests with a safe method (e.g., GET) can attach the cookie even though they are fired from a cross-site position.

Since cookies have weak integrity guarantees in their default configuration, Web developers can qualify their names with special prefixes to improve protection. The `__Secure-` prefix requires the cookie to be set over secure channels with the `Secure` attribute activated. The `__Host-` prefix extends the protection of the `__Secure-` prefix by also forcing the deactivation of the `Domain` attribute, thus scoping the cookie to a specific host rather than to its site.

Mixed Content. When a document is loaded via a secure channel, all its subresources, i.e., frames, scripts, etc, must also be received securely to not compromise the integrity of the page. If any of such resources is loaded via a non-secure channel, i.e., HTTP, a network attacker can tamper with the content of the reply, opening the possibility for, e.g., executing malicious JavaScript code within a secure context.

The W3C Mixed Content specification [39] regulates the fetching of subresources within documents loaded via a secure channel, defining as *mixed content* any insecurely-loaded subresource. Mixed content is categorized based on the corresponding security risks. Mixed content is *upgradeable* when the risk of allowing its usage is outweighed by the risk of breaking significant portions of the Web. Image, audio, and video content are all classified as upgradeable because the usage of such resource types is sufficiently high, while their loading is generally considered as low-risk. Upgradeable mixed content goes through protocol autoupgrading: the URL is rewritten to use the HTTPS protocol and an attempt is made to fetch the subresource securely. If the resource is not available via the new URL, it will not be loaded in the page.

Any mixed content that is not upgradeable is classified as *blockable*. Examples of blockable content are scripts, frames, XHR, and fetch requests. The risk of loading such content is much higher: for example, allowing insecurely-loaded scripts within a secure context would allow a network attacker to read or modify data accessed therein. Blockable mixed content is filtered and the subresource is not loaded in the document.

3 Web Invariants

A *Web invariant* is an intended security property of a Web security mechanism that should never be violated by Web browsers, i.e., any counter-example might reveal a security-relevant bug. In this paper, we define 9 Web invariants concerning two core components of the Web Platform: cookies and Mixed Content. The selection and definition of these invariants is based on the following methodology. First, we focus on Web components with clear security implications and relatively compact specifications. For each selected mechanism, we abstract the expected security properties by thoroughly analyzing the specification. We then review the existing literature to identify invariants already defined in prior research. In cases where specifications prove to be ambiguous, we encode as a Web invariant the community security expectations that emerge from previous research or from our discussion with the specification maintainers. For each of these cases, we provide a bibliographic reference or a link to the GitHub discussion. Finally, we express the invariants as first-order logic formulas. Table 1 presents an intuitive natural language description of the invariants we encode in this work. In particular, we define 6 new Web invariants (I.4–I.9) and propose an encoding of 3 invariants from the literature (I.1–I.3). In this section, we focus on the 6 new Web invariants we propose, presenting their expected security property and encoding. We first define a model to represent browser execution traces and show how security properties can be encoded in this model. We then proceed with the discussion of the invariants. Due to space constraints, the encoding of the remaining invariants is dicussed in Appendix A.

3.1 Traces and Events

We define Web invariants in terms of browser execution traces. A *trace* is represented as a list of browser events, each mapping to a concrete browser action. Events are encoded as shown in Fig. 1 and capture JavaScript API calls (*js*), network requests and responses (*net*), and hooks into the browser internals, e.g., ***cookie-jar-set*** triggers when a cookie is stored in the cookie jar. JavaScript events store a reference to the browsing context, i.e., the Window or Worker, in which the API call was executed. For each browsing context, we store a unique identifier, its location URL, and a flag indicating whether it is a secure context [38] or not.

Invariants are encoded as first-order logic formulas, which should be true for all possible traces.[1] As an example, consider our encoding of the *Confidentiality of HttpOnly Cookies* (I.2) defined in [37].

$$
\begin{aligned}
&\textsc{http-only-invariant}(tr) := \\
&\quad t_1 > t_0 \land \\
&\quad \textbf{\textit{cookie-jar-set}}(name, value, \{http\text{-}only, secure, domain, path\})@_{tr}t_0 \land \\
&\quad \textbf{\textit{js-get-cookie}}(ctx, cookies)@_{tr}t_1 \land \\
&\quad name +\!\!+ \texttt{"="} +\!\!+ value \in split\text{-}cookie(cookies) \land \\
&\quad cookie\text{-}match(path, domain, secure, ctx\text{-}location(ctx)) \rightarrow \\
&\qquad http\text{-}only = false
\end{aligned}
$$

The invariant is defined as an implication, requiring the *http-only* flag to be equal to false if a set of hypotheses is satisfied. We use the $e@_{tr}t$ predicate to check if event e is present in trace tr at timestamp $t \in \mathbb{N}$. Intuitively, this invariant says that if a script successfully uses the `document.cookie` getter (***js-get-cookie*** at time t_1) to obtain the *cookies* string, and if *cookies*, after splitting on the cookie separator `";"`, contains the string composed of the concatenation of *name*, the literal string `"="`, and *value*, then the *http-only* flag present when the cookie was set (***cookie-jar-set*** at time $t_0 < t_1$) needs to

[1]For readability, all variables are implicitly ∀-quantified when no quantification is specified.

	Name	Invariant	Description	References
Cookies	**I.1**	Integrity of Secure cookies	*Cookies with the `Secure` attribute can only be set over secure channels.*	[37]
	I.2	Confidentiality of HttpOnly cookies	*Scripts can only access cookies without the `HttpOnly` attribute.*	[37]
	I.3	Integrity of __Host- cookies	*A `__Host-` cookie set for domain d can only be set by d or by scripts included in pages on d.*	[37]
	I.4	Integrity of SameSite cookies	*A `SameSite=Lax/Strict` cookie can only be set for domain d through HTTP responses to requests initiated by domains which are same-site with d or by top-level navigations.*	[23, §4.1.2.7]
	I.5	Isolation of SameSite cookies	*If a `SameSite=Lax/Strict` cookie should not be attached to a request to load a page p, then it is not attached to that request, it is not accessible by scripts in p nor attached to requests initiated by p.*	[13]
	I.6	Cookie serialization collision resistance	*A cookie with name n and value v is serialized to the string "n=v" when attached to requests or accessed via `document.cookie`.*	[35]
	I.7	Confidentiality of Secure cookies	*`Secure` cookies are only attached to requests (resp. accessible by scripts) to potentially trustworthy URLs.*	[11]
Mixed Content	**I.8**	Blockable mixed content filtering	*Every request performed by the browser is either a toplevel request, its URL is potentially trustworthy, or the request context does not prohibit mixed content.*	[39, §4.4]
	I.9	Upgradeable mixed content filtering	*For every non-toplevel request performed by the browser where the URL is not potentially trustworthy, the request context does not prohibit mixed content and the request type is not upgradeable.*	[39, §4.1]

Table 1: Web Invariants

Trace := *List Event*	execution trace
Ctx := $\langle$*id*, *location*, *secure-context*$\rangle$	browsing context
Event :=	browser event
js-set-cookie(*Ctx*, *arg*, *ret*)	`document.cookie` setter
js-get-cookie(*Ctx*, *ret*)	`document.cookie` getter
cookie-jar-set(*name*, *value*, *attributes*, *deleted*)	cookiejar hook on set/delete cookie
net-request(*id*, *url*, *method*, *type*, *origin*, *doc-url*, *frame-ancestors*, *headers*, *body*)	network request
net-response(*id*, *url*, *headers*, *body*)	network response
js-fetch(*Ctx*, *url*)	`window.fetch` API call

Figure 1: Syntax of traces: event types.

be set to *false*. We use the *split-cookie* function to split a cookie header on the separator character `;`, returning a list, and the *cookie-match* predicate to consider the case in which the cookie set at time t_0 is readable by the browsing context *ctx* where `document.cookie` is accessed. In particular, given a URL and the path, domain and security attributes of a cookie, *cookie-match* is true when the domain matching and path matching algorithms defined in the specification [23] return true and when, if the `Secure` attribute is set, the URL uses a secure protocol. That is, when *cookie-match* is true for a URL and a cookie, we should expect that cookie to appear in the request headers and `document.cookie` for that URL.

Invariants are expressed in quantified first-order logic using the theories of uninterpreted functions, integer arithmetic, algebraic datatypes, and strings. In particular, events are defined as a datatype, the $@_{tr}$ predicate is implemented as a recursive function, and auxiliary predicates can be defined as macros or functions. This combination of theories gives us flexibility in the definition of Web invariants, e.g., allowing us to encode properties about parsing and serialization, while allowing for automated verification using the Z3 theorem prover.

3.1.1 Integrity of `SameSite` Cookies

The cookie specification explicitly forbids setting SameSite cookies (either `Lax` or `Strict`) in response to non-top-level cross-site requests [23, §4.1.2.7]. For instance, assume that https://good.com embeds a page at https://evil.com as an iframe. If the iframe includes subresources from https://good.com, the browser should discard SameSite cookies set in responses to those requests. This behavior defines additional integrity guarantees to SameSite cookies and corresponds to the following invariant.

Invariant (I.4)**.** *A cookie whose `SameSite` attribute has value `Strict` or `Lax` can only be set for domain d through HTTP responses to requests initiated by domains which are same-site with d or by top-level navigations.*

We encode this invariant as follows:

$$\begin{aligned}
&\textsc{SameSite-Cookies-Integrity}(tr) :=\\
&\quad t_1 < t_2 < t_3 \land\\
&\quad \textbf{\textit{net-request}}(id, url, _, type, origin\text{-}url, _, _, _, _, _)@_{tr}t_1 \land\\
&\quad \textbf{\textit{net-response}}(id, url, \{set\text{-}cookie\text{-}headers\}, _)@_{tr}t_2 \land\\
&\quad set\text{-}cookie \in set\text{-}cookie\text{-}headers \land\\
&\quad name \mathbin{+\!\!+} \texttt{"="} \mathbin{+\!\!+} value \in split\text{-}cookie(set\text{-}cookie) \land\\
&\quad \texttt{"SameSite="} \mathbin{+\!\!+} SS \in split\text{-}cookie(set\text{-}cookie) \land\\
&\quad (SS = \texttt{"Lax"} \land same\text{-}site = SS\text{-}Lax \lor\\
&\qquad SS = \texttt{"Strict"} \land same\text{-}site = SS\text{-}Strict) \land\\
&\quad \textbf{\textit{cookie-jar-set}}(name, value, \{same\text{-}site, path, domain\})@_{tr}t_3 \land\\
&\quad cookie\text{-}match(path, domain, _, url) \land\\
&\quad url\text{-}site(url, site) \rightarrow\\
&\qquad (type = main_frame \lor url\text{-}site(origin\text{-}url, site))
\end{aligned}$$

For every ***net-response*** event that successfully sets a cookie, i.e., that is followed by a ***cookie-jar-set*** whose parameters match the value of the response `Set-Cookie` header; if the `SameSite` attribute is set to `Lax` or `Strict`, then either the request type is *main_frame*, i.e., it is a top-level request, or the initiator of the request is same-site w.r.t the target url of the request, i.e., *origin-url*, the url of the request initiator, is in

the same site of *url*. Here, the *url-site* predicate is true when its second argument is the site of the url in the first argument.

3.1.2 Isolation of `SameSite` Cookies

SameSite cookies, especially when set with the `Strict` attribute, are widely considered a robust defense against cross-site attacks such as CSRF [23] and, more recently, XS-Leaks [3, 32, 36]. The protection is effective as long as these cookies are not attached to requests initiated by an attacker operating from a cross-site page. For instance, the specification mandates browsers to not include SameSite cookies in requests to load cross-site iframes, nor make them available to JavaScript APIs in that context [23, §5.2.1].

We verified instead that cross-site top-level navigations can cause same-site navigations to be executed, thus attaching SameSite cookies to requests initially started by the attacker. This is the case of a pop-up window opened by a cross-site page, which executes a same-site JavaScript-based redirection via, e.g., `window.location`. Browsers consider the first request as cross-site but the second as same-site, thus attaching SameSite cookies to the second request, as captured by the specification [23, §8.8.5]. Similarly, subresources loaded in a top-level cross-site context are considered same-site and are loaded with SameSite cookies attached.

By carefully examining public discussions between browser vendors [1, 10, 12], we found that the current behavior is the result of a bottom-up threat modeling process, with security implications that extend beyond what is declared in the specification: "*same-site navigations and submissions can certainly be executed in conjunction with other attack vectors such as cross-site scripting*". Indeed, SameSite `Strict` cookies can be bypassed using JavaScript-based same-site redirectors (i.e., no XSS required) [31], and loading authenticated subresources can introduce observable user-dependent state in the opened page, thus enabling XS-Leaks attacks, as we discuss in Sec. 5.2. We are currently engaging with browser vendors and specification maintainers to harmonize the specification and the implementations, and to clarify the security properties that should be expected from SameSite cookies based on the principle that high-sensitive resources (e.g., cookies and authenticated resources) should not flow into low-sensitive contexts (e.g., pages loaded from cross-site requests) [13].

Invariant (I.5)**.** *If a cookie set for domain d with the SameSite attribute set to "Lax" or "Strict" should not be attached to a request that loads a page p, then the cookie is not attached to that request, it is not accessible to scripts running in p and it is not attached to network requests initiated by p.*

We encode the invariant as:

$$
\begin{aligned}
&\textsc{SameSite-Cookies-Confidentiality}(tr) := \\
&\quad t_1 < t_2 < t_3 \land \\
&\quad \boldsymbol{cookie\text{-}jar\text{-}set}(name, value, \{secure, same\text{-}site, path, domain, host\text{-}only\})@_{tr}t_1 \land \\
&\quad (same\text{-}site = SS\text{-}Lax \lor same\text{-}site = SS\text{-}Strict) \land \\
&\quad \boldsymbol{net\text{-}request}(_, url, method, type, origin, _, _, redirs, \{cookies\}, _)@_{tr}t_2 \land \\
&\quad cookie\text{-}match(path, domain, secure, host\text{-}only, url) \land \\
&\quad \neg cookie\text{-}match\text{-}samesite(same\text{-}site, type, origin, method, redirs, url) \land \\
&\quad (\\
&\qquad (\boldsymbol{js\text{-}get\text{-}cookie}(ctx, cookies')@_{tr}t_3 \land url = ctx\text{-}location(ctx)) \lor \\
&\qquad (\boldsymbol{net\text{-}request}(_, url', method', type', origin', doc\text{-}url', _, redirs', \{cookies'\}, _)@_{tr}t_3 \land \\
&\qquad\quad doc\text{-}url' = some(url) \land \\
&\qquad\quad cookie\text{-}should\text{-}be\text{-}sent(\\
&\qquad\qquad path, domain, secure, same\text{-}site, host\text{-}only, type', origin', url', method', redirs')) \\
&\quad) \rightarrow \\
&\qquad (name \mathbin{+\!\!+} \texttt{"="} \mathbin{+\!\!+} value \notin split\text{-}cookie(cookies) \land \\
&\qquad\quad name \mathbin{+\!\!+} \texttt{"="} \mathbin{+\!\!+} value \notin split\text{-}cookie(cookies'))
\end{aligned}
$$

Assume that there is a SameSite cookie set for a specific domain, that is, the trace contains a ***cookie-jar-set*** event at time t_1, and that the browser then loads a page at time t_2 for which this cookie would have been sent if it was not SameSite (i.e., for which *cookie-match* is true but *cookie-match-samesite* is not). If there is a subsequent event at time t_3, be it a ***js-get-cookie*** where the browsing context location matches the URL of the request at t_2, or a ***net-request*** to which the cookie should be attached (i.e., for which the *cookie-should-be-sent* predicate is true), then the value of the cookie header (or the return value of `document.cookie`) $cookies'$ should not contain the cookie set at t_1, and that cookie was not attached to the request at t_2.

3.1.3 Cookie Serialization Collision Resistance

In 2020, *nameless cookies* were introduced in the cookie RFC [9] to standardize the legacy behavior adopted by major browsers. According to the standard, cookies with an empty name and a non-empty value must be serialized in the `Cookie` request header using only their value, without the `=` separator. To exemplify, a nameless cookie with value `foo` is serialized by compliant browsers as `Cookie: foo`. This serialization strategy is known to introduce collisions, which can be leveraged to perform cookie tossing attacks [35]. For example, a cookie set via `Set-Cookie: =foo=bar`, with empty name and value `foo=bar`, is attached to outgoing requests as `Cookie: foo=bar` resulting indistinguishable to a server from a cookie with name `foo` and value `bar` [23, §5.5, item 3].

Browsers can prevent cookie collisions by removing support for nameless cookies altogether, as in the case of Safari [35], or simply by including the `=` separator in the serialized cookie irrespectively of the content of the name or the value fields. Building on the previous example, the nameless cookie with value `foo=bar` would be serialized as `Cookie: =foo=bar`, allowing servers to distinguish it from a standard named cookie. This is captured by the following invariant.

Invariant (I.6)**.** *A cookie with name n and value v set for domain d is serialized to the string "n=v" when attached to requests or accessed via `document.cookie`.*

The invariant is encoded as:

$$
\begin{aligned}
&\textsc{cookie-serialization-invariant}(tr) := \\
&\quad t_2 > t_1 \wedge \\
&\quad \textbf{\textit{cookie-jar-set}}(name, value, \{secure, same\text{-}site, path, domain\})@_{tr}t_1 \wedge \\
&\quad (\\
&\quad\quad (\textbf{\textit{net-request}}(_, url, method, type, origin\text{-}url, _, _, redirs, \{cookies\}, _)@_{tr}t_2 \wedge \\
&\quad\quad\quad cookie\text{-}should\text{-}be\text{-}sent(\\
&\quad\quad\quad\quad path, domain, secure, same\text{-}site, type, origin\text{-}url, url, method, redirs)) \vee \\
&\quad\quad (\textbf{\textit{js-get-cookie}}(ctx, cookies)@_{tr}t_2 \wedge url = ctx\text{-}location(ctx) \wedge \\
&\quad\quad\quad cookie\text{-}match(path, domain, secure, url)) \\
&\quad) \wedge \\
&\quad is\text{-}effective\text{-}cookie(t_2, tr, name, value, domain, path, \texttt{""}) \rightarrow \\
&\quad\quad name +\!\!+ \texttt{"="} +\!\!+ value \in split\text{-}cookie(cookies)
\end{aligned}
$$

For every request (or access to the `document.cookie` property) at time t_2, where a cookie stored previously in the cookie jar at time t_1 should be sent (resp. retrieved), the cookie header (or the return value of `document.cookie`) should contain the string $name +\!\!+ \texttt{"="} +\!\!+ value$ after splitting on the separator `";"`. This invariant uses the three predicates *cookie-should-be-sent*, which is true if a cookie should be attached to a request, *cookie-match*, which is true if a cookie should be readable in a specific browsing context URL, and *is-effective-cookie*, which makes sure that the ***cookie-jar-set*** at t_1 we consider is the event that set the cookie in the cookie jar. Specifically, the predicate makes sure that there was no ***cookie-jar-set*** between t_1 and t_2 that overwrote the cookie stored in the cookie jar.

3.1.4 Confidentiality of `Secure` Cookies

The Cookies RFC delegates the decision of which protocols are denoted as *secure* to the specific user agent, requiring it to attach the cookies with the `Secure` attributes to URLs using such protocols [23]. Noticing this ambiguity in the RFC, we investigated how different browsers implement this behavior and discovered an inconsistency: Chromium and Firefox (behind a configuration flag) deem the `localhost` host, its subdomains, and its IP representation (`127.0.0.1`) as secure regardless of the protocol, and thus attach `Secure` cookies to local requests, whereas Safari does not. Similar inconsistencies apply to cookie prefixes, where only Firefox attaches prefixed cookies to localhost.

We contacted the HTTP Working Group [11], notifying them about the potential differences in handling of `Secure` cookies, suggesting to disambiguate the requirements on browsers by using the *potentially-trustworthy* origin definition for determining secure URLs, instead of a browser-dependent definition of secure protocol. Our proposal is currently being discussed in the Working Group. Initial feedback suggests that the specification editors are considering modifying the phrasing to include potentially trustworthy origins.

This change in the specification would align it to the de-facto standard behavior of the majority of the top browsers, which we formalize as follows:

Invariant (I.7)**.** *Cookies with the `Secure` attribute are only attached to requests sent to potentially trustworthy origins and are only readable by scripts running in browsing contexts whose origin is potentially trustworthy.*

The invariant is encoded as:

$$
\begin{aligned}
&\textsc{secure-cookies-confidentiality}(tr) := \\
&\quad t_1 > t_0 \wedge \\
&\quad \textbf{\textit{cookie-jar-set}}(name, value, \{secure = true, same\text{-}site, path, domain\})@_{tr}t_0 \wedge \\
&\quad (\\
&\quad\quad (\textbf{\textit{net-request}}(id, url, method, type, origin\text{-}url, _, _, _, \{cookies\}, _)@_{tr}t_1 \wedge \\
&\quad\quad\quad cookie\text{-}should\text{-}be\text{-}sent(\\
&\quad\quad\quad\quad path, domain, false, same\text{-}site, type, origin\text{-}url, url, method, redirs)) \vee \\
&\quad\quad (\textbf{\textit{js-get-cookie}}(ctx, cookies)@_{tr}t_1 \wedge url = ctx\text{-}location(ctx)) \\
&\quad) \wedge \\
&\quad cookie\text{-}match(path, domain, false, url) \wedge \\
&\quad name +\!\!+ \texttt{"="} +\!\!+ value \in split\text{-}cookie(cookies) \wedge \\
&\quad is\text{-}effective\text{-}cookie(t_1, tr, name, value, domain, path, \texttt{""}) \rightarrow \\
&\quad\quad is\text{-}origin\text{-}potentially\text{-}trustworthy(url)
\end{aligned}
$$

Assume that there is a cookie in the cookie jar with the `Secure` attribute set, i.e., the trace contains a ***cookie-jar-set*** event at t_0. If there is a network request (or an access to the `document.cookie` property) at t_1 where the cookie should be sent (resp. retrieved) and it is actually part of the attached cookies (resp. present in the return value of `document.cookie`), i.e., $name +\!\!+ \texttt{"="} +\!\!+ value \in split\text{-}cookie(cookies)$, then the origin of the URL of the request (or the browsing context where `document.cookie` is called) is potentially trustworthy.

3.1.5 Blockable Mixed Content

For each request, the browser determines whether it should be blocked by applying the steps defined in the *Should fetching request be blocked as mixed content* algorithm [39, §4.4]. In particular, a request is allowed when either its URL is *potentially trustworthy*, the context in which the request is performed does not restrict mixed content requests (e.g., a page loaded via HTTP making a fetch request), or when the request is top-level. We can define the following invariant.

Invariant (I.8)**.** *For every network request performed by the browser, either: (i) the context does not prohibit mixed content requests; or (ii) the request URL is potentially trustworthy; or (iii) the request is top-level.*

The encoding of the invariant is:

$$
\begin{aligned}
&\textsc{blockable-mixed-content-filtered}(tr) := \\
&\quad \textbf{\textit{net-request}}(_, url, _, type, origin, doc\text{-}url, ancestors, _, _, _)@_{tr}t1 \rightarrow \\
&\quad\quad (\neg does\text{-}settings\text{-}prohibits\text{-}mixed\text{-}security\text{-}contexts(\\
&\quad\quad\quad\quad origin, doc\text{-}url, ancestors) \vee \\
&\quad\quad\quad is\text{-}url\text{-}potentially\text{-}trustworthy(url) \vee \\
&\quad\quad\quad (type = main_frame \wedge nil = ancestors))
\end{aligned}
$$

The invariant uses the predicates *is-url-potentially-trustworthy*, which is true if the request URL is potentially trustworthy according to the respective algorithm of the secure context specification, and *does-settings-prohibits-mixed-security-contexts*, that is the implementation of the respective algorithm defined by the Mixed Content specification [39, §4.3] and is true if the request initiator origin is potentially trustworthy, or if any ancestor of the navigation initiator has a potentially trustworthy origin. The invariant also uses the expression $type = main_frame \wedge nil = ancestors$ to check if a request is a top-level navigation.

3.1.6 Upgradeable Mixed Content

For upgradeable mixed content requests, e.g., loading images over insecure channels, the browser should rewrite the URL of the request by changing its scheme from HTTP to HTTPS. The mixed content specification defines the conditions for applying this rewriting in the *Upgrade mixed content request to a potentially trustworthy URL* algorithm [39, §4.1]. This algorithm applies to every request by the Fetch specification, thus every successful request made by the browser for upgradeable mixed content should have been upgraded. That is, every non-top-level request whose URL is not potentially trustworthy should not be upgradeable or should be permitted by Mixed Content. This corresponds to the following invariant:

Invariant (I.9)**.** *For every non-toplevel network request performed by the browser whose URL is not potentially trustworthy, the request context does not prohibit mixed content or the request type is not upgradeable.*

The invariant is encoded as:

$$
\begin{aligned}
&\textsc{upgradeable-mixed-content-filtered}(tr) := \\
&\quad \boldsymbol{net\text{-}request}(_, url, _, type, origin, doc\text{-}url, ancestors, _, _, _)@_{tr}t1 \land \\
&\quad \neg is\text{-}url\text{-}potentially\text{-}trustworthy(url) \land \\
&\quad type \neq main_frame \rightarrow \\
&\qquad (\neg does\text{-}settings\text{-}prohibits\text{-}mixed\text{-}security\text{-}contexts(\\
&\qquad\qquad origin, doc\text{-}url, ancestors) \lor \\
&\qquad\ \ \neg is\text{-}mixed\text{-}content\text{-}upgradeable(type))
\end{aligned}
$$

where the presence of a request in the trace whose URL is not potentially trustworthy and whose type is different from *main_frame* (as upgradeable mixed content does not restrict toplevel requests) implies that both *is-mixed-content-upgradeable*, which checks if the request type is upgradeable (by implementing its definition in [39, §3.1]), and *does-setting-prohibits-mixed-security-contexts* are false.

4 Trace Verification Pipeline

In this section, we will first motivate with an example the importance of abstracting WPT tests into execution traces in order to automate the discovery of Web invariant violations, and then describe our verification pipeline in detail.

4.1 Motivating Example

We present a simple example to motivate why looking at failed WPT tests does not already enable reasoning about security. The WPT test `/mixed-content/gen/top.meta/unset/img-tag.https.html` is a set of test cases that check the mixed content behavior of browsers when fetching `img` tags. In particular, the test expects image requests to always be performed within an HTTPS browsing context (i.e., a window with a HTTPS URL as location). This is expected, as upgradeable mixed content requests should be allowed when the browser is able to rewrite the request URL to use the HTTPS scheme, i.e., performing the auto-upgrade. This test is successful on the stable versions of Firefox and Safari, but fails on Chromium, as some of the requests fail.

The execution trace of the test contains multiple ***net-request*** events, each corresponding to the requests performed by the browser during execution. Specifically, for each embedding of an `img` tag, the event includes the image URL, the request type (*image*), and additional fields characterizing the request, e.g., the origin of the request initiator and the URL of the document where the new image will be loaded. The I.8 invariant mandates that for every ***net-request*** event, at least one of three conditions must hold for it to be compliant with the Mixed Content specification. Since the request is not top-level, i.e., its type is *image*, and it originates from a page loaded via HTTPS, i.e., *does-setting-prohibit-mixed-content* is true, then its URL must be potentially trustworthy, i.e., its scheme must be HTTPS. In the traces produced during the execution of Firefox and Safari, the ***net-request*** event corresponding to the embedding of the image has an insecure URL, i.e., the image is fetched via HTTP, violating the requirement of I.8. In Chrome, on the other hand, the request is auto-upgraded and the corresponding ***net-request*** has a potentially trustworthy URL, thus I.8 is not violated.

Since the WPT test only checks for the images to be loaded, without explicitly testing their protocol, Firefox and Safari, which do not currently implement protocol auto-upgrading [2, 15] and perform the mixed content requests without blocking them, pass the test. Chromium, on the other hand, performs the auto-upgrading as mandated by the Mixed Content specification. However, since the image is served on a non-standard HTTP port (8000), the browser upgrades the protocol without changing the port causing a connection error.

This example highlights that the WPT test results alone may not always capture potential security concerns since failed tests do not necessarily break Web invariants, and, conversely, successful tests might break Web invariants. Tests can not only be unsuccessful because browsers implement new security features, as in the example above, but they can also fail if the execution relies on unimplemented APIs. This further emphasizes that observing a discrepancy across the WPT results of different browsers (i.e., simple WPT-based differential testing) is not a direct indication of security issues. By verifying browser traces obtained during the execution of WPT tests, irrespectively of test results, our approach provides a deeper insight into each test. In particular, violating an invariant is a clear indicator of potential security issues in the exercised browser behavior, pinpointing the specific Web components requiring immediate attention.

4.2 Methodology

Our methodology for detecting security-relevant issues in browser implementations leverages the WPT test suite and consists of two main stages, as shown in Fig. 2. First, the execution traces produced by executing the WPT tests on

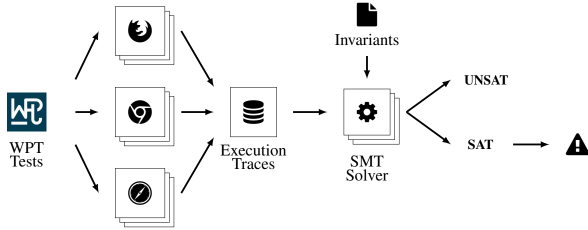

Figure 2: Trace Verification Pipeline.

the three major Web browsers (Chromium, Firefox, Safari) are collected into a database. Second, the obtained traces are post-processed, translated to SMT-LIB, and checked against the Web invariants we define in Sec. 3 using an SMT solver. When the solver cannot prove the validity of the invariant on a test trace (SAT, i.e., a counterexample exists), a violation is found on the specific browser. Our analysis pipeline is based on the Kubernetes container orchestration platform, allowing us to execute multiple instrumented browsers and the SMT solving in parallel. We detail in the following the main steps of the pipeline and our criteria for selecting the relevant tests.

Test Selection. The tests part of the WPT project can be classified into four main categories: (i) rendering tests, which test the graphical output of the browser (by, e.g., comparing it to screenshots) to verify that pages are displayed as expected; (ii) `testharness.js` tests, which test JavaScript interfaces available in browsing contexts, allowing to automatically check assertions about their behavior; (iii) `wdspec` test, which test parts of the WebDriver protocol and are written in the Python programming language; (iv) manual tests that require human interaction to determine their result. In this work, we focus on `testharness.js` tests, since our Web invariants cover JavaScript and browser internals behavior, ignoring most UI aspects. In particular, we consider all `testharness.js` tests of the April 2023 version of the WPT test suite. We detail our test selection in Table 6 (Appendix B), where we report the version (commit hash) of the test suite, the considered WPT subfolders, and the respective number of tests for each folder.

Trace Collection. We run each WPT test in its own isolated ephemeral container named *runner*. Each runner container includes a specific version of the tested browser, all its run-time dependencies, our patched version of the WPT tooling, and the browser instrumentation composed of a browser extension and a proxy (Sec. 4.3). For Safari, the runner container executes a MacOS virtual machine containing the instrumented browser. We build a runner container for Chromium (version 118.0.5961.0), Firefox (version 116.0.3) and Safari (version 16.4). Once the runner container terminates the execution of a WPT test, it stores the execution trace in JSON format in a centralized database. Note that we ignore test assertions, storing the captured trace regardless of the test results.

Verification. Upon completion of the runner container, the generated JSON file is post-processed and translated to SMT-LIB format. In particular, the events that were captured by our browser instrumentation are converted to execution traces following the format described in Sec. 3.1. It may be the case that multiple events recorded by the browser instrumentation happened simultaneously, i.e., the JSON stores multiple events with the same timestamp. This may occur when, for instance, a page containing multiple subresources is rendered: the browser may try to load all resources in parallel, thus resulting in multiple events of type *network-request* to be recorded at the same time. In such cases, the SMT-LIB translator generates multiple traces, each corresponding to a single permutation of the simultaneous events, allowing us to consider all possible orderings of the concurrent events. Note that, in practice, the number of concurrent events in WPT traces rarely exceeds four events, thus having a negligible impact on the pipeline performance.

Once execution traces are translated to SMT-LIB format, we use an SMT solver to query, for each trace, the *validity* of each Web invariant. That is, we check satisfiability of the negation of the invariant applied to each trace. This satisfiability checking may have three possible outcomes: (UNSAT) the invariant is valid, i.e., it is true for the current trace; (SAT) the invariant is not valid, i.e., the current trace is a counterexample for the invariant; (UNKNOWN) the solver was not able to prove nor disprove the invariant, hence in such cases we cannot draw any conclusion and we do not report any violation. Whenever the solver returns SAT, we obtain a model, i.e., an instantiation of the variables mapping them to the concrete values from the trace that make the invariant false. Being based on the standard SMT-LIB format, our pipeline supports all standard-compliant solvers that implement decision procedures for quantified string constraints, integer arithmetic and algebraic data types. Specifically, we currently support both the Z3 theorem prover and CVC5.

Violating an invariant may have several security implications, and for this reason, we manually inspect the execution trace of every SAT result and design a minimal proof of concept (PoC) attack to showcase the vulnerability in the affected browsers. We discuss the discovered attacks in Sec. 5.2.

4.3 Browser Instrumentation

Browser instrumentation and trace collection are essential components of our pipeline. Our main goal is to develop a browser instrumentation solution that provides a balance between observability and cross-browser support, while minimizing the implementation effort. Our instrumentation must be easily integrated into existing testing pipelines such as the Web Platform Tests and work across different browsers. We refer the reader to Appendix C for an analysis of the design space of browser instrumentation techniques.

4.3.1 Implementation

Based on our design space analysis, we implemented a browser instrumentation solution which combines a browser extension with an external proxy that improves on the limitations of the extension API with respect to its ability to inspect network traffic. Our solution provides the necessary hooks to monitor internal browser state, JavaScript API calls, and have a complete picture of the network activity when collecting browser execution traces.

Internal Browser State Monitoring. With extensions, we gain access to the internal browser state not available to regular scripts or external monitoring tools. This state includes the `CookieJar`, and network activity such as requests and responses. This internal state is accessible to extensions via background scripts, which have no access to the DOM but can make full use of the extension APIs. Network events are monitored by registering callback functions that run whenever a request is about to be sent, and when a request is deemed completed, i.e., it has a response or it was dropped. These callbacks provide access to the request and response headers, and additional information added by the browser, like the tab and frame IDs of the request initiator. The `CookieJar` can also be monitored via `onChange` callbacks, whose execution can be delayed depending on the state of the JavaScript event loop. Due to these inherent delay inconsistencies, we opted for polling the state of the `CookieJar` instead of registering callback functions, which gives us higher precision timestamps for `CookieJar` events.

JavaScript API Call Monitoring. In addition to monitoring network events and internal browser state, we focus on JavaScript API calls as another category of relevant events for our analysis. We proxy the relevant JavaScript functions, logged as events used in the invariants, to record function calls in a centralized structure located in the extension's background script. This proxying is done through `Proxy` objects and method overriding, enabling us to collect all the relevant data associated with each API call, such as its arguments and the respective browsing context. Our instrumentation logs calls to the setter and getter of `document.cookie`, but more JavaScript methods could be supported in the future using similar techniques. We adopt a dynamic approach for our instrumentation using content scripts, which are extension scripts that run in the context of webpages. Each webpage is injected with a content script that installs the proxy functions according to the extension configuration. This dynamic instrumentation is more versatile and scalable compared to code rewriting methods and allows us to efficiently track and analyze JavaScript API calls as they occur in real-time.

External Proxy. We incorporate an external proxy into our framework to overcome two main issues: (i) the restrictions imposed by browsers over network content deemed sensitive and, hence, inaccessible to background scripts but visible through a network proxy (e.g., request and response bodies); (ii) the inconsistent delay between network events and the execution of their corresponding callback event handlers. When a network request leaves the browser, the callback corresponding to its event handler is queued in the extension's JavaScript event loop and eventually executed. If the proxy intercepts the request before the callback is executed, the proxy event's timestamp is more accurate and is used as the request event timestamp in the trace.

4.3.2 Limitations

While our browser instrumentation technique based on extensions and proxies offers a powerful means to monitor internal browser state, JavaScript API calls, and network events, enabling comprehensive browser security analysis with cross-browser compatibility and minimal code modification or rewriting, it is essential to acknowledge the inherent limitations of this approach. These limitations include:

Browser Discrepancies. In our instrumentation, we strive to use only browser extension APIs that are compatible across browsers. However, browser behavior varies across implementations, which can introduce limitations to our approach. These inconsistencies are detected by manual inspection of our results, and whenever possible, we implement specific workarounds. But, some issues require changes to the browsers' source code and bug fixes. For example, a bug in Firefox's URL matching prevents the content scripts from being injected into opaque origins, such as data URL iframes. This limitation hinders our ability to monitor JavaScript API calls in these frames, negatively impacting the comprehensiveness of our analysis, and it cannot be circumvented without changes to Firefox's source code. For our current usage, this translates to missing events executed in iframes with opaque origins, which can lead to false negatives. Another example is Safari's resource isolation for WebDriver-controlled instances, which isolates resources such as the cookie jar. This isolation prevents our extension from effectively monitoring specific resources such as the `CookieJar` within Safari instances controlled by WebDriver, which translates to missing `CookieJar`-related events for Safari execution traces, leading to false negatives.

API Constraints. Currently our instrumentation is able to monitor the necessary components and collect the events required to reason about our invariants, i.e., `CookieJar`, network, and some JavaScript API call events. While some extensions to our instrumentation are possible, they are constrained by the availability of APIs in both JavaScript and the extension environment, and by the Same Origin Policy which is applied to the injected content scripts. For example, information such as the effective Content Security Policy (CSP) of a frame cannot be directly monitored as it is not accessible to scripts running in pages, nor to browser extensions. To support the analysis of the CSP mechanism with our approach, we must develop inference and heuristic techniques, which we could

use alongside other artifacts of our instrumentation, such as response headers, to infer the CSP enforced on a given frame. Another constraint to our approach is monitoring the DOM. Content scripts injected by the extension are still subject to the Same Origin Policy. Therefore, a full picture of the DOM may prove difficult to obtain without heuristics over other events such as network activity and DOM mutation.

In summary, while our browser instrumentation technique was proven effective in collecting security-relevant browser execution traces, these limitations underline the importance of developing better introspection and instrumentation mechanisms for browser testing. These mechanisms would benefit not only our approach but also testing frameworks like WPT, which currently uses incomplete workarounds to test features like cookies and the Content Security Policy.

4.4 Discussion: Extensibility

The methodology we propose is meant to enable specification maintainers and browser developers to check their security expectations, expressed as Web invariants, against multiple implementations. This way, security issues can be identified early during development and across Web platform or browser updates, e.g., for regression testing.

In this paper, we encode 9 invariants, as discussed in Sec. 3, showing that the verification pipeline is not bound to a single security mechanism and can be extended to support additional Web features. Although we do not consider the required expertise to develop new invariants a limiting factor, given that specification maintainers already possess this knowledge, the expressiveness of the invariants may be limited by the introspection capabilities of our instrumentation. Specifically, every JavaScript API or property access that can be wrapped with Proxy objects can easily be traced, encoded as an event (as in Fig. 1), and used in the definition of new invariants. Instead, monitoring internal browser state which is not exposed to pages or extensions, e.g, CSP, may prove to be difficult to trace without relying on heuristics or a different instrumentation approach (e.g., browser code patching [26]).

Automated generation. The definition of new Web invariants relies on the manual effort of understanding the security requirements of a specification and encoding them into a logical proposition. Automation could be beneficial for aiding the process, allowing more properties to be covered. Previous work on Web invariants identifies the importance of clearly defining the security properties of the Web as a way to have a *sound* scientific understanding of Web security [18]. Thus, the generation of Web invariants presents the challenge of retaining soundness while characterizing the relevant Web mechanisms. We leave the development of a methodology to automatically extend the set of invariants as future work.

Invariant						
	SAT	UNK.	SAT	UNK.	SAT	UNK.
I.1	0	0	0	0	–	–
I.2	0	0	0	0	–	–
I.3	0	0	0	0	–	–
I.4	0	0	1	0	–	–
I.5	10	0	6	0	–	–
I.6	15	0	9	0	–	–
I.7	0	0	0	0	–	–
I.8	0	448	24	643	21	692
I.9	0	355	18	509	0	628

Table 2: Trace verification results.

5 Evaluation Results

We evaluate our methodology by verifying, using our pipeline, the 9 Web invariants we define in Sec. 3 against the execution traces of the 24896 testharness tests from the April 2023 version of the WPT suite. Note that every browser is executed 24896 times, totaling 74688 traces. We use the Z3 theorem prover as the SMT solver component since it proved to be the best performing for our invariants. We set a timeout of 10 minutes for the execution of the browser for each test, and 10 minutes for each Z3 query. When Z3 is not able to return an answer within the timeout it returns UNKNOWN. All our experiments have been conducted on a cluster with 132 VCPUs (AMD EPYC 2.0GHz) and 382GB of RAM.

5.1 Preliminary Results

Table 2 reports the outcome of our analysis of the three major browsers on the WPT test suite, showing the number of tests for which Z3 found violation of a Web invariant (SAT). Additionally, we report the number of UNKNOWN results, for which our pipeline could not generate a definitive answer. Note that, given the limitations of Safari instrumentation (see Sec. 4.3.2), invariants about cookies are expected to always return UNSAT there (marked as – in Table 2), since the Safari traces never contain the ***cookie-jar-set*** event, which is used in the premises of our cookie invariants.

Five invariants have at least one violation. The results confirm our expectation that different implementations may exhibit different behaviors with respect to the implemented security mechanisms. In particular, although there is overlap in some of the SAT traces between different browsers, Table 2 highlights that some SAT results are browser-specific. We discuss in Sec. 5.2 the security implications of violating each invariant, where we group SAT results into concrete attacks against specific browsers that we present as case studies.

For four invariants our pipeline does not report any violation, so they are valid on the entirety of the execution traces produced by WPT. This may happen in the cases where the invariants are well-known and expected to hold by the literature (I.1, I.2). Additionally, we may obtain no violation if

	Trace Collection			Verification			Total
	avg	std	total	avg	std	total	
	28s	6s	23h 29m	19s	1m 42s	23h 05m	1d 22h 35m
	40s	8s	1d 07h 18m	27s	2m 06s	1d 08h 34m	2d 15h 52m
	27s	8s	1d 06h 34m	32s	2m 28s	1d 14h 33m	2d 21h 07m

Table 3: Trace verification execution times.

the traces generated by the test suite do not cover the specific preconditions for an attack to be performed. As an example, I.3 does not hold in the current Web platform [37] because of an attack that requires combining *domain relaxation*, i.e., assignment to the Document.domain property, with `__Host-` cookies. This invariant may have no SAT results because the WPT test suite never uses the two Web features together in the same test. A similar consideration applies to I.7, as the `localhost` URL is never used in cookie-related tests. We discuss these cases in Sec. 5.3, where we explore additional tests beyond what is included in WPT.

Z3 returned UNKNOWN during the verification of the Mixed Content invariants I.8 and I.9. These are caused by the complex checks that are mandated by the Mixed Content specification, in particular the recursive checking of the entire ancestor chain for each network request, which may negatively affect the solver speed and result in UNKNOWN if the execution time exceeds the verification timeout.

Performance. The performance of our trace verification pipeline is shown in Table 3. The total run-time for each of the three major browsers is reported together with the time required for executing the browser (collecting execution traces) and the Z3 verification time. Executing a single WPT test on each of the browsers consistently requires less than one minute, whereas the verification with the Z3 theorem prover shows more variability, while still requiring less than a minute on average. This confirms that verifying Web invariants on the traces generated by WPT does not add substantial overhead to the execution of the testing suite, but supplements the result obtained from each WPT test with an assessment of the security of the exercised browser functionality.

5.2 Attacks on Major Browsers

Every SAT result obtained as the output of the Z3 theorem prover corresponds to a violation of a Web invariant on the execution trace of a specific browser, as captured by our instrumentation. These results require a manual analysis to identify and aggregate similar issues, organizing them into concrete inconsistencies. This effort is supported by the model obtained from Z3, which provides the concrete values from the trace that violate the invariant, highlighting problematic events in the trace and allowing us to easily discern the cause of the violation. A goal of our analysis of SAT results is to determine the root causes underlying these inconsistencies and to quantify their security impact, and in particular, if they can lead to concrete real-world attacks. This step is also critical in identifying any false positives introduced by the observability limitations of our browser instrumentation (Sec. 4.3.2). For instance, the inability to correctly observe a specific browser event may lead to the generation of a violating trace for an otherwise compliant browser. For example, a missing cookie deletion event may result in a violating trace if we expect that cookie to be attached to a subsequent network request.

We now present all the attacks resulting from the analysis of the SAT results, discussing them in the form of case studies. In particular, we aggregated all 104 invariant violations into 10 confirmed attacks and 5 false positives as shown in Table 4. Due to space constraints, we refer to Appendix D for a discussion of each false positive and its causes.

Framed Pages Mixed Content Bypass

Z3 reported SAT for Safari for the trace of the `mixed-content/nested-iframes.window.html` test, where the browser successfully performs a fetch request to an insecure endpoint coming from a frame whose origin is potentially trustworthy, violating the I.8 invariant. After some investigation, we concluded that Safari incorrectly performs mixed content checks, i.e., secure pages embedded in insecure origins were not considered potentially trustworthy, and therefore, mixed content was not blocked except for requests to load scripts, stylesheets, or requests to insecure WebSocket. For example, if `https://bank.com` contains an authenticated mixed content request (i.e. via fetch), framing it over `http://attacker.com` will cause the request to not be filtered. This behavior might incorrectly expose non-Secure cookies in clear over the network to passive network attackers. Moreover, the integrity of the fetch request (and its response) would not be ensured against network attackers, meaning that attackers could tamper with its contents to, for example, alter the control flow of JavaScript execution on the target page.

Disclosure. We disclosed the attack to the Safari developers. The issue has been fixed in Safari 16.6.

Sandbox Attribute Mixed Content Bypass

The test `mixed-content/csp.https.window.html` consists in a webpage using the `sandbox allow-scripts` CSP directive. The page is loaded via HTTPS so mixed content should be prohibited, nevertheless, a fetch request targeting an HTTP endpoint is not blocked in Safari, violating I.8. In the trace, the CSP directive is effectively setting the origin of the page to `null`. Since the null origin is not potentially trustworthy, the requests are not filtered. This vulnerability can be combined with the previous one to obtain a complete bypass of the mixed content policy: the presence of the `sandbox` directive makes the browser allow mixed content requests to scripts, stylesheets, and insecure WebSockets, which are otherwise blocked. As a consequence, if `https://bank.com` contains a mixed content script, framing it with the `sandbox` attribute over `http://attacker.com` will allow the request to the script to be sent. A network attacker can tamper with its content

Invariant	Total SAT	SAT Traces (Chromium)	SAT Traces (Firefox)	SAT Traces (Safari)	Type	Description (causes of SAT)
I.4	1	–	1	–	🐞	SameSite cookie integrity violation
I.5	18⋆	1	2	–	🐞	SameSite cookies attached to (favicon, subresource, fetch) requests (*requests*)
		10	5	–	🐞	SameSite cookies accessible via Document.cookie (*non-HTTP*)
		1	–	–	🐞	SameSite cookies attached to location.reload() network requests (*reload*)
		1	–	–	⊗	Incorrect event ordering
I.6	16	2	3	–	🐞	Nameless cookies serialization collision
		2	1	–	⊗	Missing events from sandboxed iframes
		5	2	–	⊗	Missing delete cookie event
		1	–	–	⊗	Incorrectly tagged requests: missing request initiator origin
I.8	45	–	–	1	🐞	Framed pages mixed content bypass
		–	–	1	🐞	Sandbox attribute mixed content bypass
		–	–	7	🐞	Mixed content beacon requests not blocked
		–	11	–	🐞	Mixed content Websocket requests not blocked
		–	13	10	🐞	Mixed content autoupgrade not performed
		–	–	3	⊗	Incorrectly tagged requests: missing request type
I.9	18	–	18	–	🐞	Mixed content autoupgrade not performed

Table 4: Aggregated SAT results. (🐞: attack; ⊗: false positive; ⋆: the same trace may contain multiple attacks)

to obtain code execution on `https://bank.com`, in a context where the origin is `null`. In this scenario SOP prevents certain operations (e.g., cookies access) but other attacks, such as user input tracking, and DOM modifications can still be performed. For instance, an attacker embedding the login page of `bank.com` can track user inputs by registering new listeners through the injected script and exfiltrate user credentials whenever a user is tricked into logging in.

Disclosure. We disclosed this attack to the Safari developers. The issue has been fixed in Safari 16.6 and CVE-2023-38592 was assigned to this and the previous vulnerability.

Mixed Content Beacon Requests Not Blocked

A beacon request is a non-blocking POST request sent using the `navigator.sendBeacon` API. Mixed content beacon requests are *blockable* and therefore should be filtered. However, our pipeline SAT results show that Safari performs such requests, violating I.8. When a mixed content beacon request is not blocked, attached cookies and the data attached to the request are leaked even to passive network attackers.

Disclosure. We reported the problem to the Safari developers and we are waiting for confirmation.

Nameless Cookies Serialization Collisions

Part of the SAT results reported for I.6 are caused by the serialization of nameless cookies. Our invariant expects every cookie with name n and value v to be serialized as $n = v$. However, Chromium and Firefox serialize nameless cookies where n = "" simply as v. Consequently, our pipeline will report a violation whenever I.6 matches a trace where a nameless cookie is serialized. The higher number of SAT results related to nameless cookies in Firefox compared to Chromium stems from an inconsistency between the browsers: whenever Firefox encounters the JavaScript API call `document.cookie = ""`, a cookie with an empty name and value is set, unlike Chromium, which does not set any cookie. The serialization of nameless cookies enables attackers to shadow arbitrary cookies. This capability includes shadowing `Secure` cookies from insecure origins, relaxing an attacker's requirements to perform cookie tossing or eviction attacks on `Secure` cookies, which would typically require a secure origin [35].

Disclosure. The issue was already reported to the IETF HTTP Working Group by Squarcina et al. [35] during their study of cookie integrity.

SameSite Cookie Integrity Violation

Our pipeline returned SAT for Firefox in the trace of the `cookies/samesite/setcookie-navigation.https.html` test, where a cookie with the `SameSite` attribute set to `Strict` is successfully set in the response to a cross-site network request initiated from an iframe, violating the I.4 invariant. In particular, an iframe loading `https://attacker.com` within `https://bank.com` might navigate itself to some page at `https://bank.com`, which sets `SameSite` cookies in the response to the navigation request. Note that this applies to both `Strict` and `Lax` SameSite cookies. A gadget attacker [18, 19] can thus leverage this behavior to overwrite cookies to perform, e.g., de-authentication attacks.

Disclosure. We reported this vulnerability to Firefox developers [14] who confirmed the issue assigning it a severity rating of *Normal (blocks non-critical functionality)*, planning a fix for the next release.

SameSite Cookies Isolation

The SAT results returned from our pipeline for I.5 fall into three categories: *request*, *non-HTTP*, or *reload*. Traces in these categories all have a similar setup but differ in how the cookie is retrieved. The setup follows this structure: (i) a

cookie c with `SameSite` attribute set to `Strict` or `Lax` is set for domain d; (ii) a top-level request initiated by domain d', where c is not attached, opens page p with domain d, which is cross-site with d'. From this point, *request* traces perform a network request, initiated by d (from page p) that is considered same-site and attaches c, violating I.5. This request can be, for example, a subresource load, a request to load the `favicon`, or a request generated by a call to the `fetch` JavaScript API. *Non-HTTP* traces retrieve the cookie c through a call to `document.cookie` from p, violating I.5. Finally, *reload* traces perform a call to `location.reload`, triggering a same-site request that reloads page p and attaches c, which violates I.5. Note that *reload* traces are not SAT for Firefox. By manually investigating this inconsistency, we discovered that Firefox does not attach `SameSite` cookies to network requests initiated from calls to `location.reload`, as it considers these requests cross-site.

Setting the `SameSite` attribute of cookies to `Strict` is considered an effective defense against CSRF and XS-Leak attacks as these cookies are not attached to cross-site requests. However, attackers can exploit the browser behavior highlighted by I.5 SAT results to bypass these restrictions. In particular, attackers can forge same-site requests starting from a cross-origin position by abusing, e.g., redirection gadgets that trigger attacker-controlled same-site navigation requests, effectively enabling CSRF attacks. Another security implication is the possibility of performing XS-Leaks. Consider a page that loads a script depending on whether the subresource load request attaches `SameSite=Strict` cookies and that this script modifies the DOM of the target page, altering `window.length`. An attacker could navigate to this page through `window.open`, and even though `SameSite=Strict` cookies are not attached to the top-level request, they will be included in subresource loads in the target page. An attacker can then use the `length` property of the window handler to infer the authentication status of the victim.

Disclosure. We are currently engaging with the HTTP Working Group to clarify the security properties that should be expected from SameSite cookies [13].

Mixed Content WebSockets Requests Not Blocked

These SAT results refer to a set of tests for the following scenario: WebSocket requests sent from a Worker using the *ws* protocol. If the Worker is created from a secure page, so its origin is potentially trustworthy, we expect the request to be blocked as mixed content. However, in Firefox it is not, violating I.8. Investigating the issue uncovered that Firefox incorrectly implements the filtering for WebSocket requests. In particular, filtering is not performed if either the origin's scheme is *blob:* or the request is sent from a Worker created in a trustworthy origin using a *data:* URI.

Disclosure. We disclosed the problem to Mozilla. The issue has been fixed in Firefox 120.

Mixed Content Autoupgrade Not Performed

From the analysis of these SAT results, we observed how both Safari and Firefox do not perform protocol autoupgrading, and as a consequence, upgradeable mixed content requests are sent over the network, violating I.8 or I.9. When this happens, network attackers can tamper with the content of upgradeable requests to attempt phishing users by e.g. swapping the icons of two buttons tricking them into performing destructive operations (e.g., *delete message* instead of *send message*). To prevent these attacks, the latest revision of the specification forbids loading upgradeable mixed content, but, as of today, neither Firefox nor Safari are compliant. However, they are aware of the issue and are planning a fix [2, 15].

5.3 Comprehensiveness of Tests

In this section, we explore additional tests beyond those in WPT, to (i) show that our pipeline can generalize to different test suites without modifications, and (ii) to assess how the *comprehensiveness* of the individual tests, in terms of the usage of Web features, affects the discovery of inconsistencies. As mentioned in Sec. 5.1, the limited scope of tests may prevent our pipeline from discovering violations. This is the case when tests do not include actions that are preconditions for the attack, e.g., when a violation is enabled by the combination of multiple Web features.

We construct a separate test suite comprising 9 tests to exercise behavior not covered by WPT. The selected tests are shown in Table 5. The first group (1-5) corresponds to the violations discovered by Veronese et al. [37] affecting the current Web platform. These tests combine multiple features to reproduce the attack traces generated by WebSpec. For instance, the first test uses *domain relaxation* to allow a subframe to set a `__Host-` cookie for a different origin. The remaining `webspec_*` tests use a combination of CSP, Service Workers, and Trusted Types. Given that our invariants only focus on cookies and Mixed Content, these tests are not expected to reveal new violations. The second group of tests (6-7) reproduces the browser testing performed by Squarcina et al. [35]. In particular, the tests try to perform cookie tossing, eviction based on cookie jar overflow, and serialization collisions based on nameless cookies. Each test is composed of multiple sub-tests that correspond to various combinations of cookie properties, e.g., tossing of `Secure` cookies over insecure channels, or eviction of `__Host-` cookies. Note that these tests are actively abusing undefined behavior to perform eviction, as the RFC does not impose a specific limit to the number of entries in the cookie jar (although implementations are allowed to set one). Finally, the last two (8-9) tests use features that are not covered by WPT. The `localhost_cookies` test sets `Secure`, `__Secure-`, and `__Host-` cookies for the `localhost` domain, which is never used in WPT. The `multi_nested_frames` test sets cookies using mixed-content resources loaded across multiple levels of frames, as WPT does not include cookies in mixed-content tests and uses up to two levels of nesting.

	Test Name	SAT					
		I.1	I.3	I.6	I.7	I.8	I.9
1	`webspec_host_frames`	–	[Firefox]	–	–	–	–
2	`webspec_csp_sw`	–	–	–	–	–	–
3	`webspec_csp_sop`	–	–	–	–	–	–
4	`webspec_tt_frames`	–	–	–	–	–	–
5	`webspec_csp_blob`	–	–	–	–	–	–
6	`crumbles_tossing` (5)	–	–	[Chrome] [Firefox]	–	–	–
7	`crumbles_eviction` (8)	–	–	–	–	–	–
8	`localhost_cookies` (3)	[Chrome]	–	–	–	–	–
9	`multi_nested_frames`	–	–	–	–	[Firefox] [Safari]	[Firefox]

Table 5: Additional tests and new violations.

Table 5 reports the results of running our pipeline on the traces produced by the new test suite. The experiment confirms that new violations can be discovered using more comprehensive tests. In particular, I.3 does not hold for Firefox, where domain relaxation allows compromising `__Host-` cookies integrity. Interestingly, Chrome satisfies the invariant, since starting from version 115, the `document.domain` property is immutable [5], preventing pages from relaxing the SOP. The I.1 invariant does not hold for Chrome, as it is possible to set `Secure` cookies over an insecure connection when the URL is `localhost`. This matches the behavior we discuss in Sec. 3.1.4 and encode in I.7. Note that Firefox violates the invariant only when a specific setting flag is enabled. The new test suite, additionally, allows us to rediscover a violation for I.6, since the `crumbles_tossing` test uses nameless cookies. Similarly, I.8 and I.9 are SAT because upgradeable mixed content is not upgraded nor blocked in both Firefox and Safari. Safari also incorrectly loads mixed-content frames if the top-level window is loaded via HTTP, regardless of the protocol used to load any intermediate frame. Specifically, in `multi_nested_frames`, the test opens a window with three nested frames, where the top-level window is loaded via HTTP, the intermediate frames are over HTTPS, and the innermost frame is over HTTP, which should be blocked.

This experiment shows that employing a more comprehensive test suite has the potential to identify additional violations. While our focus for this paper is WPT, as it is currently the most complete and regularly updated browser testing suite available, our pipeline can be applied to any alternative testing suites, potentially improving its efficacy.

6 Related Work

Browser Testing. BrowserAudit is a test suite designed to assess the implementation of Web security mechanisms in Web browsers [25]. It includes more than 400 automated test cases for SOP, CSP, CORS, cookies and security headers. While the approach is undeniably useful to detect bugs, it suffers from significant limitations compared with our proposal. First, test cases in BrowserAudit were manually created by the authors. Our approach instead leverages WPT, which is an actively maintained existing test suite backed up by a large community (to date, its GitHub repository counts more than 1,500 contributors). Moreover, the security implications of failed BrowserAudit tests are also manually identified: failures are categorized by the authors as warning or critical, supposedly based on their security impact according to the authors' understanding. Our approach instead detects effective violations of Web security invariants, i.e., deviant behavior clearly contradicting existing specifications. Concretely, the latest versions of Chromium and Firefox pass all the tests in BrowserAudit except for a few warnings, showing that the current set of test cases cannot identify relevant bugs in existing browsers, as opposed to our pipeline.

Other work on the automated detection of security bugs in browsers targeted specific mechanisms or vulnerabilities. For example, DiffCSP can detect bugs in CSP implementations [40], while other work investigated incoherencies in the implementation of SOP [33, 34]. Automated testing has also been used to detect new cross-site leaks in browsers [32] and to study the support of Web security mechanisms in mobile browsers [29]. All these proposals proved effective to identify new bugs, yet they are tailored to specific needs and do not leverage general security notions like the concept of Web security invariant adopted in this paper.

Browser Instrumentation. VisibleV8 (VV8) [26] is a browser instrumentation framework, implemented as a set of patches for the Chromium browser, that allows for tracing JavaScript function calls and property access during navigation. The VV8 patches are designed to minimize the modified lines of code, so that they can be easily applied to updated browser versions. Browser instrumentation implemented as patches to the JavaSript engine, compared to in-band JavaScript instrumentation (e.g., prototype patching), has the unique advantage of being tamper-proof and impossible to detect by malicious scripts. However, it suffers from being tied to a specific browser implementation and requires additional manual work to be ported to new browser versions. For this reason, in this paper we opted for browser extensions, which allow, via the WebExtension API, cross-platform instrumentation that requires minimal to no effort to be applied to any extension-supporting browsers.

Similarly to VV8, JSgraph [28] is a patch to the Chromium source code that instruments the interface between Blink and V8, allowing for the recording of audit logs related to the execution of JavaScript in the browser. JSgraph aims to provide a detailed JS and DOM-related event log to aid in analyzing and reconstructing Web attacks. To this end, the tool includes a visualization component that shows the captured events in the form of a graph, highlighting causal relationship between events. JSgraph shares its main limitations with VV8, being tied to the specific implementation of the Chromium browser, requiring a substantial amount of manual work to

keep up with the constantly evolving browser code.

Formalization of Web Invariants. In their 2010 paper, Akhawe et al. [18] presented a formal model of the Web platform for the Alloy analyzer and used it to verify the security of Web mechanisms such as CORS, the Origin header and HTML5 forms, discovering three new vulnerabilities. The authors encode in the model a set of security goals which are grouped into *security invariants* and *session integrity*. In particular, they emphasize the importance to identify clear Web security invariants that define the desired security goals of the Web platform, proposing the definition of 4 invariants. More recently, Veronese et al. proposed WebSpec [37], a framework for the analysis of Web security mechanisms composed of a model of the browser in the Coq proof assistant and a toolchain for automated model-checking against Web security invariants. In particular, the authors define 10 Web invariants concerning cookies, the CSP and the CORS, discovering two new attacks and presenting a formal proof of the correctness of their proposed mitigations. Although our approach for the definition of new Web invariants presents some similarities to both works, previous research focused on models of the browser and not on specific implementations. By leveraging the WPT test suite, we can (i) automatically check the actual browser implementation behavior (i.e., execution traces) against Web invariants; and (ii) sidestep the issue of requiring to manually update a browser model to match the updates of the Web platform. Additionally, compared to previous works, we are the first to support Mixed Content, modeling its specification by defining two new Web invariants.

7 Conclusion

This paper presents a novel methodology for formally and automatically detecting security issues in browser implementations of client-side Web security mechanisms. Leveraging the WPT test suite, our framework collects browser execution traces and validates them using the Z3 theorem prover against Web security invariants. We formalized and encoded a total of 9 Web invariants and discovered violations within WPT, resulting in 10 unique attacks. We reported all our findings to the affected parties and kickstarted discussions with standardization bodies to address shortcomings at the specification level. This research positively answers our initial research question, showing that the proposed automated approach can provide valuable guidance to browser vendors in identifying vulnerable Web components requiring immediate attention.

Acknowledgments

The project leading to this publication has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101034440. Additionally, this work has been partially supported by the European Research Council (ERC) under the European Union's Horizon 2020 research (grant agreement 771527-BROWSEC); by the Vienna Science and Technology Fund (WWTF) and the City of Vienna (Grant ID: 10.47379/ICT22060); by the Austrian Research Promotion Agency (FFG) through the COMET K1 SBA; by DAIS - Università Ca' Foscari Venezia within the IRIDE program and by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU; by Fundação para a Ciência e a Tecnologia (FCT) under project UIDB/50008/2020 (Instituto de Telecomunicações).

References

[1] Bug 1459321 - treat loads the result from location.reload() as samesite. https://bugzilla.mozilla.org/show_bug.cgi?id=1459321.

[2] Bug 247197 - upgrade requests in mixed content settings. https://bugzilla.mozilla.org/show_bug.cgi?id=1811787.

[3] Bypassing samesite restrictions using on-site gadgets. https://portswigger.net/web-security/csrf/bypassing-samesite-restrictions.

[4] Chrome devtools protocol. https://chromedevtools.github.io/devtools-protocol/.

[5] Chrome will disable modifying document.domain to relax the same-origin policy. https://developer.chrome.com/blog/immutable-document-domain/.

[6] chrome.declarativenetrequest extension api. https://developer.chrome.com/docs/extensions/reference/declarativeNetRequest/.

[7] chrome.webrequest extension api. https://developer.chrome.com/docs/extensions/reference/webRequest/.

[8] Remote protocol (cpd) - firefox. https://firefox-source-docs.mozilla.org/remote/cdp/.

[9] [RFC6265bis] Accept nameless cookies. https://github.com/httpwg/http-extensions/commit/0178223.

[10] [RFC6265bis] Clarify behaviour on page refresh for samesite cookies. https://github.com/httpwg/http-extensions/issues/628.

[11] [RFC6265bis] Inconsistent browser behavior with secure and prefix cookies on localhost. https://github.com/httpwg/http-extensions/issues/2605.

[12] [RFC6265bis] Refactor cookie retrieval algorithm to support non-http apis. https://github.com/httpwg/http-extensions/pull/1428.

[13] [RFC6265bis] SameSite=Strict cookie isolation on cross-site windows. https://github.com/httpwg/http-extensions/issues/2644.

[14] Samesite cookies are set by cross-site iframe navigations. https://bugzilla.mozilla.org/show_bug.cgi?id=1844827.

[15] Ship mixed content level 2 upgrading of passive mixed content. https://bugzilla.mozilla.org/show_bug.cgi?id=1811787.

[16] The Web Platform Tests project. https://web-platform-tests.org/.

[17] Webkit web inspector. https://webkit.org/web-inspector/.

[18] D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song. Towards a Formal Foundation of Web Security. In *CSF*, 2010.

[19] A. Barth, C. Jackson, and J. C. Mitchell. Securing Frame Communication in Browsers. In *USENIX Security*, 2008.

[20] P. Bernardo, L. Veronese, V. D. Valle, S. Calzavara, M. Squarcina, P. Adão, and M. Maffei. Web platform threats: Automated detection of web security issues with WPT – artifacts and source code. https://github.com/SecPriv/web-platform-threats, 2023.

[21] A. Bohannon and B. C. Pierce. Featherweight firefox: Formalizing the core of a web browser. In J. K. Ousterhout, editor, *USENIX Security*, 2010.

[22] S. Calzavara, R. Focardi, M. Squarcina, and M. Tempesta. Surviving the web: A journey into web session security. *ACM Comput. Surv.*, 2017.

[23] L. Chen, S. Englehardt, M. West, and J. Wilander. Cookies: HTTP State Management Mechanism (IETF Draft). RFC 6265bis, 2022.

[24] D. Fett, R. Küsters, and G. Schmitz. A Comprehensive Formal Security Analysis of OAuth 2.0. In *CCS*, 2016.

[25] C. Hothersall-Thomas, S. Maffeis, and C. Novakovic. BrowserAudit: automated testing of browser security features. In *ISSTA*, 2015.

[26] J. Jueckstock and A. Kapravelos. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In *IMC*. ACM, 2019.

[27] S. Kim, Y. M. Kim, J. Hur, S. Song, G. Lee, and B. Lee. FuzzOrigin: Detecting UXSS vulnerabilities in browsers through origin fuzzing. In *USENIX Security*, 2022.

[28] B. Li, P. Vadrevu, K. H. Lee, and R. Perdisci. Jsgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser javascript executions. In *NDSS*, 2018.

[29] 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 *NDSS*, 2019.

[30] Mozilla. Public Suffix List. https://publicsuffix.org/.

[31] PortSwigger. Bypassing SameSite cookie restrictions. https://portswigger.net/web-security/csrf/bypassing-samesite-restrictions.

[32] J. Rautenstrauch, G. Pellegrino, and B. Stock. The leaky web: Automated discovery of cross-site information leaks in browsers and the web. In *S&P*. IEEE, 2023.

[33] J. Schwenk, M. Niemietz, and C. Mainka. Same-origin policy: Evaluation in modern browsers. In E. Kirda and T. Ristenpart, editors, *USENIX Security*, 2017.

[34] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the Incoherencies in Web Browser Access Control Policies. In *S&P*, 2010.

[35] M. Squarcina, P. Adão, L. Veronese, and M. Maffei. Cookie crumbles: Breaking and fixing web session integrity. In *USENIX Security '23*, 2023.

[36] A. Sudhodanan, S. Khodayari, and J. Caballero. Cross-origin state inference (cosi) attacks: Leaking web site states through xs-leaks. In *NDSS*, 2019.

[37] L. Veronese, B. Farinier, P. Bernardo, M. Tempesta, M. Squarcina, and M. Maffei. Webspec: Towards machine-checked analysis of browser security mechanisms. In *S&P*. IEEE, 2023.

[38] W3C. Secure Contexts. https://w3c.github.io/webappsec-secure-contexts/, 2021.

[39] W3C. Mixed Content. https://www.w3.org/TR/mixed-content, 2023.

[40] S. Wi, T. T. Nguyen, J. Kim, B. Stock, and S. Son. Diffcsp: Finding browser bugs in content security policy enforcement through differential testing. In *NDSS*, 2023.

A Encoding Known Web Invariants

We report in the following our encoding in first-order logic of the 3 invariants which were previously defined in the literature. For each invariant, we provide the natural language version of the property and its encoding in our model.

A.1 Integrity of Secure Cookies

The RFC dictates that it should not be possible to set cookies with the `Secure` attribute from insecure channels [23, §5.5]. This invariant has been previously formalized as part of the WebSpec framework [37] as follows.

Invariant (I.1). *Cookies with the `Secure` attribute can only be set over secure channels.*

The invariant is encoded in our model as follows:

$$
\begin{array}{l}
\textsc{secure-cookies-invariant}(tr) := \\
\quad t_2 > t_1 \land \\
\quad \textbf{\textit{net-response}}(_, \mathit{url}, \{\mathit{set\text{-}cookie\text{-}headers}\}, _)@_{tr}t_1 \land \\
\quad \mathit{set\text{-}cookie} \in \mathit{set\text{-}cookie\text{-}headers} \land \\
\quad \mathit{name} +\!\!+ \texttt{"="} +\!\!+ \mathit{value} \in \mathit{split\text{-}cookie}(\mathit{set\text{-}cookie}) \land \\
\quad \texttt{"Secure"} \in \mathit{split\text{-}cookie}(\mathit{set\text{-}cookie}) \land \\
\quad \textbf{\textit{cookie-jar-set}}(\mathit{name}, \mathit{value}, \{\mathit{Secure{=}true}\}), \mathit{false})@_{tr}t_2) \Rightarrow \\
\qquad (\mathit{url\text{-}proto}(\mathit{url}, \texttt{"wss"}) \lor \quad \mathit{url\text{-}proto}(\mathit{url}, \texttt{"https"}))
\end{array}
$$

For every network response at time t_1 that leads to a cookie being set in the cookie jar (at time t_2) that has the `Secure` attribute set to $true$, then the protocol of the response url is either $https$ or wss (i.e., it is a secure channel).

A.2 Confidentiality of HttpOnly cookies

The `HttpOnly` cookie attribute informs browsers that accesses to cookies with this attribute set to true by non-HTTP APIs, i.e., `document.cookie`, should not be allowed. This property was formalized in the literature [37] as:

Invariant (I.2). *Scripts can only access cookies without the `HttpOnly` attribute.*

We encode the invariant as:

$$
\begin{array}{l}
\textsc{http-only-invariant}(tr) := \\
\quad t_2 > t_1 \land \\
\quad \textbf{\textit{cookie-jar-set}}(\mathit{name}, \mathit{value}, \{\mathit{http\text{-}only}, \mathit{secure}, \mathit{domain}, \mathit{path}\})@_{tr}t_1 \land \\
\quad \textbf{\textit{js-get-cookie}}(\mathit{ctx}, \mathit{cookies})@_{tr}t_2 \land \\
\quad \mathit{name} +\!\!+ \texttt{"="} +\!\!+ \mathit{value} \in \mathit{split\text{-}cookie}(\mathit{cookies}) \land \\
\quad \mathit{cookie\text{-}match}(\mathit{path}, \mathit{domain}, \mathit{secure}, \mathit{ctx\text{-}location}(\mathit{ctx})) \Rightarrow \\
\qquad \mathit{http\text{-}only} = \mathit{false}
\end{array}
$$

For every access to `document.cookie` in the domain *domain* at time t_2 that successfully returns a cookie previously stored in the cookie jar for the same domain (at time t_1) then the cookie's `HttpOnly` attribute has the value $false$.

A.3 Integrity of __Host- cookies

Browsers should enforce that cookies with a name prefix of `__Host-` are set with an empty *domain* attribute, making these cookies host-only. Effectively, these cookies can only be set by responses to the domain that created them or by scripts running in that domain. Veronese et al. [37] discuss this property of the `__Host-` prefix and propose the following natural language formalization:

Invariant (I.3). *A `__Host-` cookie set for domain d can only be set by d or by scripts included in pages on d.*

We encode the invariant in our model as:

$$
\begin{array}{l}
\textsc{host-invariant}(tr) := \\
t_2 > t_1 \land \\
(\quad \textbf{\textit{net-response}}(_, \mathit{url}, \{\mathit{set\text{-}cookie\text{-}headers}\}, _)@_{tr}t_1 \land \\
\quad \mathit{set\text{-}cookie} \in \mathit{set\text{-}cookie\text{-}headers} \land \\
\quad \texttt{"__Host-"} +\!\!+ \mathit{cname} +\!\!+ \texttt{"="} +\!\!+ \mathit{cvalue} \in \mathit{split\text{-}cookie}(\mathit{set\text{-}cookie}) \land \\
\quad \mathit{url\text{-}domain}(\mathit{url}, \mathit{host}) \quad) \lor \\
(\quad \textbf{\textit{js-set-cookie}}(\mathit{ctx}, \mathit{set\text{-}cookie}, _)@_{tr}t_1 \land \\
\quad \texttt{"__Host-"} +\!\!+ \mathit{cname} +\!\!+ \texttt{"="} +\!\!+ \mathit{cvalue} \in \mathit{split\text{-}cookie}(\mathit{set\text{-}cookie}) \land \\
\quad \mathit{url\text{-}domain}(\mathit{ctx\text{-}location}(\mathit{ctx}), \mathit{host}) \quad) \\
\mathit{cookie\text{-}jar\text{-}set}(\texttt{"__Host-"} +\!\!+ \mathit{cname}, \mathit{cvalue}, \{\mathit{domain}\}, \mathit{false})@_{tr}t_2 \Rightarrow \\
\quad \mathit{domain} = \mathit{host}
\end{array}
$$

For every network response or access to Document.cookie property at t_1 that causes a `cookie-jar-set` event at t_2 which sets a `__Host-`-prefixed cookie, the effective domain of the cookie must be equal to the domain of the *url* of the network response or to the browsing context where the access to Document.cookie was performed.

B Test Selection

Table 6 reports the considered tests for our evaluation. In particular, we execute all `testharness.js` tests from the `d888ebb` version of WPT (Apr 2023).

C Design Space Analysis

We consider several approaches to browser instrumentation:

Browser extensions are software modules that extend browser functionality. Extensions are powerful as they can access internal browser structures such as the cookie jar, monitor and intercept network traffic, and access and modify the DOM.

Service workers are scripts that run in the background and control the behavior of Web pages. They act as proxy servers between Web applications, the browser, and the network. However, they run on a different JavaScript context and have no access to the DOM.

WebDriver is a Web standard that describes a remote interface that allows the control and introspection of browsers.

Chrome Devtools Protocol (CDP) is a remote debugging protocol [4] which provides access to the DOM, network activity, and a JavaScript debugger.

External proxies act as intermediaries between a Web server and the browser and allow the monitoring and intercepting of network traffic.

Browser source code patching allows access to all internal browser structures and events, enabling the most comprehensive monitoring and trace collection.

html	6404	compute-pressure	30	webvr	7
referrer-policy	1301	web-bundle	29	remote-playback	7
content-security-policy	821	focus	29	pointerlock	7
fetch	754	domparsing	29	mediasession	7
dom	473	soft-navigation-heuristics	28	mediacapture-fromelement	7
IndexedDB	454	cors	27	keyboard-lock	7
svg	448	payment-request	26	fledge	7
xhr	391	shape-detection	25	x-frame-options	6
navigation-api	375	webrtc-encoded-transform	24	webrtc-stats	6
workers	321	credential-management	24	shared-storage	6
service-workers	296	animation-worklet	24	gamepad	6
websockets	276	reporting	23	file-system-access	6
streams	251	mediacapture-image	23	close-watcher	6
webaudio	247	import-maps	23	badging	6
wasm	246	domxpath	23	webrtc-svc	5
bluetooth	230	worklets	22	wai-aria	5
encoding	215	orientation-event	21	push-api	5
upgrade-insecure-requests	197	inert	20	delegated-ink	5
shadow-dom	169	requestidlecallback	19	content-index	5
webrtc	168	longtask-timing	19	clear-site-data	5
mixed-content	163	visual-viewport	18	webrtc-identity	4
webmessaging	154	storage-access-api	18	vibration	4
mathml	140	long-animation-frame	18	ua-client-hints	4
webxr	137	hr-time	18	proximity	4
custom-elements	132	screen-wake-lock	17	payment-method-basic-card	4
pointerevents	124	quirks	17	mimesniff	4
speculation-rules	123	notifications	17	merchant-validation	4
resource-timing	122	mediacapture-record	17	lifecycle	4
WebCryptoAPI	119	js-self-profiling	17	device-memory	4
web-animations	119	battery-status	17	virtual-keyboard	3
scheduler	108	urlpattern	16	trust-tokens	3
encrypted-media	106	orientation-sensor	16	top-level-storage-access-api	3
client-hints	104	measure-memory	16	timing-entrytypes-registry	3
scroll-animations	102	geolocation-API	16	screen-details	3
eventsource	100	screen-orientation	15	periodic-background-sync	3
editing	98	old-tests	15	parakeet	3
infrastructure	91	browsing-topics	15	netinfo	3
trusted-types	88	beacon	15	mst-content-hint	3
FileAPI	87	web-share	14	generic-sensor	3
layout-instability	81	resize-observer	14	autoplay-policy-detection	3
media-source	78	input-events	14	webrtc-priority	2
permissions-policy	76	imagebitmap-renderingcontext	14	webhid	2
performance-timeline	76	background-fetch	14	savedata	2
encoding-detection	75	secure-payment-confirmation	13	png	2
web-locks	73	presentation-api	13	permissions-revoke	2
webcodecs	73	picture-in-picture	13	permissions-request	2
webvtt	71	payment-handler	13	managed	2
fullscreen	70	console	13	intervention-reporting	2
intersection-observer	69	scroll-to-text-fragment	12	installedapp	2
cookies	69	is-input-pending	12	html-media-capture	2
selection	63	font-access	12	direct-sockets	2
user-timing	62	accelerometer	12	deprecation-reporting	2
largest-contentful-paint	61	web-nfc	11	density-size-correction	2
signed-exchange	60	speech-api	11	background-sync	2
cookie-store	60	page-visibility	11	window-placement	1
compression	59	network-error-logging	11	webrtc-ice	1
serial	58	idle-detection	11	web-otp	1
webidl	55	geolocation-sensor	11	webmidi	1
url	54	forced-colors-mode	11	webdriver	1
event-timing	54	server-timing	10	subresource-integrity	1
paint-timing	53	screen-capture	10	private-click-measurement	1
navigation-timing	53	sanitizer-api	10	payment-method-id	1
mediacapture-streams	51	pending-beacon	10	page-lifecycle	1
webnn	50	mediacapture-insertable-streams	10	media-playback-quality	1
preload	50	media-capabilities	10	mediacapture-region	1
webusb	49	magnetometer	10	mediacapture-handle	1
webstorage	49	gyroscope	10	mediacapture-extensions	1
feature-policy	49	compat	10	input-device-capabilities	1
fs	48	audio-output	10	eyedropper	1
loading	47	ambient-light	10	entries-api	1
clipboard-apis	47	webrtc-extensions	9	ecmascript	1
element-timing	46	touch-events	9	custom-state-pseudo-class	1
uievents	43	permissions	9	contenteditable	1
portals	42	webgl	8	content-dpr	1
webtransport	37	video-rvfc	8	contacts	1
webauthn	36	subapps	8	apng	1
js	35	secure-contexts	8	acid	1
document-policy	33	keyboard-map	8	accname	1
storage	32	document-picture-in-picture	8		

Table 6: Considered WPT tests.
Total: 24896, WPT Version: `d888ebb`

Instrumentation	Ease of Implementation	Cross-Browser Compatibility	Observability
Browser Extensions	↑	↑	↑
Service Workers	↑	↑	→
WebDriver	→	↑	↑
Chrome Devtools Protocol	→	↓	↑
External Proxies	↑	↑	→
Source Patching	↓	↓	↑

Table 7: Design space analysis: comparison of browser instrumentation methods. (↑ : High, →: Medium, ↓ : Low)

We evaluate these options according to three criteria: (i) ease of implementation, (ii) cross-browser compatibility, and (iii) observability, i.e., how many events the instrumentation method can collect, and summarize the results in Table 7.

Ease of Implementation. *Browser extension* implementations generally depend on the complexity of the extension, which, for our use case, is directly proportional to the number of different events we mean to collect in the traces. However, since the extension API is well documented, and extensions are written in JavaScript, which is not verbose, we deem the ease of implementing our browser instrumentation using *browser extensions* high. *Service Workers* and the *remote protocols* would also be high in the *ease of implementation* scale for our use-case, for the same reasons as browser extensions, if not for the fact that *Service Workers* and *WebDriver* are either testing target or testing mechanisms of WPT, which means that the WPT framework and testing suite would require extra modifications, making its implementation more complex. *External proxies* also require low implementation effort for the same reasons as browser extensions, making use of expressive programming languages and well-documented APIs, with a small set of potential events to monitor. On the opposite end of the spectrum, *source code patching* requires extensive manual effort in understanding browser implementations, which are generally written in lower-level languages like C++ and are very extensive.

Cross-Browser Compatibility. *Browser extension* features are dictated by the manifest versions supported by a given browser. However, all major browsers support manifest versions v2 (Firefox and Safari) or v3 (Chromium), which significantly overlap in the supported APIs, making it possible to write powerful cross-browser extensions. The *Service Worker API* is a standard supported by all major browsers. The *WebDriver* protocol is also a standard, and its core set of functionality is supported across browsers. The *Chrome Devtools Protocol* is not a standard and, therefore, not supported consistently across different browsers [8], [17]. *External proxies* are completely browser agnostic and score high on compatibility. *Source code patching* concerns only a given browser, significantly hindering its cross-browser compatibility

Observability. *Browser extensions* provide access to browser internal structures and events, like network activity and the cookie jar, and allow the dynamic inclusion of arbitrary JavaScript in pages using only the overlapping APIs between manifest v2 and v3, completely cross-browser and independent from WPT. However, this approach also has some limitations. For instance, manifest v3 does not allow the inspection of network request and response bodies for privacy reasons [6, 7], and Chromium-based browsers no longer support manifest v2 extensions. *Service workers* are similar to *external proxies* in observability, as they are limited to monitoring network traffic since they do not have access to the DOM. Remote protocols like WebDriver and CDP also provide access to browser internal structures and events, like network activity and the cookie jar, and allow the dynamic inclusion

of JavaScript in the target pages, similar in observability to browser extensions. Source code patching provides the highest possible level of observability since it grants access to every internal structure in the given browsers.

D False Positives

In this section, we examine the false positives we obtained during our evaluation of the Web invariants against WPT traces and discuss their causes.

Incorrect event ordering
For one trace, our pipeline returned SAT for I.5 due to out-of-order events. Since our monitoring of network events is based on callbacks, which are subject to scheduling delays, and our monitoring of `CookieJar` events is polling-based, the order in which these events are collected may not match the concrete browser execution. Invariant I.5 matches a specific order of events, i.e., a `cookie-jar-set` event setting cookie *c*, followed by a cross-site network request that opens a page *p* where cookie *c* is not attached, and an access to cookie *c* from page *p*. Consider a concrete browser execution where a first network request leads to a cookie being set, which is then attached to a subsequent request. If the first two events are swapped in the trace, this incorrect trace can be matched by invariant I.5, leading to a violation.

Missing events from sandboxed iframes
Our pipeline reported SAT for the traces of the test `cookies/samesite/sandbox-iframe-subresource.https.html` on Chromium and Firefox for I.6. In this trace, a previously set cookie is expected to be attached to a network request from an iframe. However, since the `iframe` has the `sandbox` attribute, it cannot attach existing cookies to network requests. Since our instrumentation cannot observe events originating from sandboxed `iframes`, nor detect whether an iframe is sandboxed, invariant I.6 cannot account for this behavior. In this trace, browsers correctly withheld a cookie that I.6 expects to be attached to a network request, leading to an invariant violation.

Missing delete cookie event
In some cases, our browser instrumentation is unable to detect cookie deletion events. Missing cookie deletion events cause some of the SAT results for I.6. Consider an execution where a previously set cookie *c* is deleted before a network request that would attach *c*, but the cookie deletion event is missing from the trace. I.6 will expect the cookie to be attached to the network request since, according to the trace, that cookie is still in the `Cookiejar`. However, since in the browser execution the cookie no longer exists, it is not attached to the network request, leading to an invariant violation.

Incorrectly tagged requests
For three traces, Z3 returned SAT for I.8 on Safari. These are caused by the lack of the request-type field in the Request object returned by the instrumentation for network events. In particular, a toplevel request to a URL which is not potentially trustworthy should be allowed. However, the absence of the request type makes the expression *type = main_frame* false in I.8, violating the invariant. Similarly, one Chromium trace violates I.6 since a network event in the trace is missing the *origin* field, i.e., the origin of the request initiator. Since the *origin* field is used by *cookie-should-be-sent* to determine if a SameSite cookie should be attached to a request, a request missing the initiator origin information and containing no cookie can be incorrectly tagged as violating when *cookie-should-be-sent* incorrectly (because of the missing origin) determines that cookies should be present.