
Lorenzo Veronese, Benjamin Farinier, Pedro Bernardo, Mauro Tempesta, Marco Squarcina, Matteo Maffei

► To cite this version:

HAL Id: hal-04316042
https://inria.hal.science/hal-04316042
Submitted on 30 Nov 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Lorenzo Veronese*, Benjamin Farinier†, Pedro Bernardo*, Mauro Tempesta*, Marco Squarcina*, Matteo Maffei*
*TU Wien
†Univ Rennes, Inria, CNRS, IRISA

Abstract—The complexity of browsers has steadily increased over the years, driven by the continuous introduction and update of Web platform components, such as novel Web APIs and security mechanisms. Their specifications are manually reviewed by experts to identify potential security issues. However, this process has proved to be error-prone due to the extensiveness of modern browser specifications and the interplay between new and existing Web platform components. To tackle this problem, we developed WebSpec, the first formal security framework for the analysis of browser security mechanisms, which enables both the automatic discovery of logical flaws and the development of machine-checked security proofs. WebSpec, in particular, includes a comprehensive semantic model of the browser in the Coq proof assistant, a formalization in this model of ten Web security invariants, and a toolchain turning the Coq model and the Web invariants into SMT-lib formulas to enable model checking with the Z3 theorem prover. If a violation is found, the toolchain automatically generates executable tests corresponding to the discovered attack trace, which is validated across major browsers.

We showcase the effectiveness of WebSpec by discovering two new logical flaws caused by the interaction of different browser mechanisms and by identifying three previously discovered logical flaws in the current Web platform, as well as five in old versions. Finally, we show how WebSpec can aid the verification of our proposed changes to amend the reported inconsistencies affecting the current Web platform.

I. INTRODUCTION

Web browsers are considered among the most complex software in use today, and the number of Web platform components, i.e., browser functionalities and security mechanisms, is constantly increasing. These are typically proposed by browser vendors in the form of a W3C Editor’s Draft and discussed within the community. If enough consensus is reached, the standardization process has to progress through several maturity levels before becoming a W3C recommendation.

While the implementation of new Web platform components is subject to extensive compliance testing (see, e.g., the Web Platform Tests project [6]), their specifications undergo a manual expert review to identify potential issues: this is a continuous and extremely complex process that has to consider the interplay with legacy APIs and should, in principle, be revised whenever new components land on the Web platform.

Unfortunately, manual reviews tend to overlook logical flaws, eventually leading to critical security vulnerabilities. For example, the HttpOnly flag was introduced by Internet Explorer 6 [21] as a way to protect the confidentiality of cookies with this attribute by not exposing them to scripts. Eight years after its launch, Singh et al. discovered that this property could be trivially violated by any scripts accessing the response headers of an AJAX request via the getResponseHeader function [48]. Security vulnerabilities at the level of Web specifications have also affected CORS [9], CSP [49], and Trusted Types [5], to name a few.

We argue that this dire situation stems from several concurring factors: (i) Web platform components are specified informally and therefore their analysis, albeit conducted by expert eyes, may easily overlook corner cases; (ii) there is no precise understanding of which security properties should be seen as invariants in the Web and, thus, be preserved by updates of the Web platform; (iii) Web platform components are typically evaluated in isolation, without considering their interactions, that is, the entangled nature of the Web platform.

Our Contributions. In this work, we advocate a paradigm shift, letting Web platform components and their interplay undergo a formal security analysis as opposed to a manual expert review. In particular, we introduce WebSpec, the first formal framework for the security analysis of browser security mechanisms that supports the automated detection of logical flaws as well as machine-checked security proofs. As outlined in Figure 1, WebSpec includes:

- a formal browser model in Coq (Section [1]), which (i) formalizes a core set of Web platform components, including well-established (cookies, SOP, CORS, etc.) as well as recent ones (e.g., CSP level 3 and Trusted Types), and (ii) supports the definition of Web invariants, i.e., properties that are expected to hold in the Web;
- the WebSpec verification toolchain (Section [III]), encompassing a compiler and a trace verifier. The compiler translates the browser model and the Web invariants into SMT-lib formulas to enable model checking by the Z3 automated theorem prover. A salient feature of WebSpec is the support for both bug finding and proof generation.
We demonstrate the effectiveness of WebSpec by:

- defining ten Web invariants against which we identify (i) a new attack on cookies caused by the interaction with legacy APIs, (ii) a new inconsistency between CSP and a planned change to the HTML standard, as well as (iii) three previously reported logical flaws in the current Web platform (Section IV);
- validating all five Web inconsistencies against the latest versions of Chrome and Firefox;
- adjusting the model to reflect past states of the Web platform in order to identify five previously published attacks, with the goal of showing that automated security analysis would have prevented these vulnerabilities;
- writing the proofs in the Coq model for the correctness of application-specific assumptions that attackers could violate.
- conducting an experimental evaluation to demonstrate the effectiveness of the WebSpec toolchain and the optimizations we integrated therein (Section V).

II. Browser Model

This section provides an overview of the main components of our browser model written in Coq. The model focuses on Web platform components, i.e., browser functionalities and security mechanisms, abstracting away from the network and Web servers. Our formalization enables reasoning about all possible sequences of events leading to an inconsistent state without necessarily having to model a specific Web application. We are indeed interested in proving and disproving Web invariants, i.e., properties of the Web platform that are expected to hold across its updates and independently on how its components can interact with each other [9]. Web invariants are supposed to hold for all Web applications, irrespectively of application-specific assumptions that attackers could violate. For instance, scripts in our model can, in principle, execute arbitrary sequences of any of the API calls we support, as this would be the case in presence of cross-site scripting attacks. The model also includes configuration flags that enable reasoning on former states of the Web platform or testing new proposals prior to their implementation.

A. Core Abstractions

The browser is modeled as a transition system in which a state evolves from an initial to a final configuration following a list of events and according to an inductive relation named \( \text{Reachable} \) parameterized by a global environment:

\[
\text{Inductive Reachable : } \text{Global} \rightarrow \text{list Event} \rightarrow \text{State} \rightarrow \text{Prop.}
\]

Intuitively, given a global environment \( gb \), a list of events \( evs \), and a state \( st \), \( \text{Reachable gb evs st} \) means that, starting from a given initial state, \( st \) is reachable by executing sequentially the events in \( evs \) under environment \( gb \).

The \text{Global} environment contains concrete values (e.g., the browser configuration) or symbolic variables (e.g., a set of pages) which are constant through the evolution of the browser state. An \text{Event} represents an atomic action that modifies the state, e.g., sending a network request or updating the DOM, which may originate from different sources, such as the browser itself, a script, or a service worker. A \text{State} (Figure 2) is a collection of datatypes used to model browser components.

Based on these ingredients, we formalize Web invariants within our model as follows, where \text{hypothesis} and \text{conclusion} are predicates that may refer to the global environment, past events, or the current state of the browser:

\begin{align*}
1 & \quad \text{Parameter} \; \text{hypothesis} : \text{Global} \rightarrow \text{list Event} \rightarrow \text{State} \rightarrow \text{Prop}. \\
2 & \quad \text{Parameter} \; \text{conclusion} : \text{Global} \rightarrow \text{list Event} \rightarrow \text{State} \rightarrow \text{Prop}. \\
3 & \quad \text{Definition} \; \text{Invariant} (gb : \text{Global}) \; (evs : \text{list Event}) \; (st : \text{State}) \\
& \quad \Rightarrow \text{Reachable gb evs st} \rightarrow \text{hypothesis gb evs st} \rightarrow \text{conclusion gb evs st}.
\end{align*}

In Section IV we introduce ten Web invariants and discuss their security implications caused by the interplay of different Web platform components.

B. Page Rendering

The main component used to model the rendering functionality is the \text{Window} datatype. \text{Window} represents a window in terms of browsing context [62 §7.1], i.e., an environment in which the browser displays a document. The field \text{wd_location} is the URL being visited and \text{wd_document} contains the displayed document. Since a \text{Window} can represent either a top-level window or a frame, \text{wd_parent} contains an optional index which, if empty, denotes a top-level window or points to the parent frame otherwise. Similarly, \text{wd_initiator} contains an optional index which is used to track the source browsing context of this window [62 §7.11] by storing a reference to the window responsible for starting the navigation.

The \text{Document} datatype represents a Web page loaded and rendered in a browser window. When a page is loaded, \text{dc_html} represents the HTML code of the response, while \text{dc_dom} contains the rendered elements of the page. Static elements, e.g., forms and possibly other markup tags, are rendered immediately. Subresources of the page, such as frames and scripts, require an additional request to be included in \text{dc_dom}. For instance, the presence of a \text{HTMLFrame} in \text{dc_html} might cause three additional events to be executed in sequence: a request (\text{EvRequest}), a response (\text{EvResponse}), followed by the update of the DOM (\text{EvDOMUpdate}) resulting in \text{DOMFrame} being added to \text{dc_dom}. This approach enables fine-grained modeling of the rendering process of the browser. In particular, our model captures the order in which resources are loaded and the presence or absence of specific elements.

WebSpec currently supports forms with the \text{method} and \text{action} attributes, images, scripts, and frames. Rendered
frames in dc_dom contain a reference to the corresponding Window, reflecting the tree-like structure of the DOM. Finally, Document also includes the list of headers (dc_headers) in the HTTP response used to render the page and dc_domain, an optional field used to model domain relaxation via the document.domain API (Section II-E).

C. Networking and Cookies

The main component used to model the networking functionality is the FetchEngine, which abstracts network access and is responsible for sending requests and receiving responses. ft_request contains the last emitted request, and ft_emitter maps to the originator of the request, i.e., whether the request is top-level or generated by the inclusion of subresources, issued by a script, a form, a worker, or it is a CORS preflight. ft_response is a field that either contains the corresponding response or is empty if the request is still pending. Finally, we store Emitter * Request * Response triples in ft_history to track previous network accesses.

The modeling of Request and Response is rather straightforward, as shown in Figure 2. We support requests and responses through HTTP and HTTPS protocols. For requests, we model the HTTP methods GET, POST, PUT, DELETE, and OPTION. Concerning responses, the following HTTP status codes are supported: (i) 200 OK, successful response, (ii) 204 No Content, successful response with an empty body, (iii) 302 Found, redirection with no integrity guarantees in the redirected request over the HTTP method and the body of the original request [43], (iv) 307 Temporary Redirect, redirection enforcing that the method and body of the original request are preserved in the redirected one.

Supported headers are Origin, Cookie and Referer for requests, and Content-Type, Set-Cookie, Location and Referrer-Policy for responses. To support CSP and CORS, we include the Content-Security-Policy and Access-Control-Allow-Origin headers.

Cookies are stored in the CookieJar as a list of triples in the form Domain * Cookie * CookieAttribute, where Domain represents the host setting the cookie, Cookie is a pair corresponding to the name of the cookie and its value, and CookieAttribute is a record containing the attributes. Modeled cookie attributes are Domain, Path, Secure, HttpOnly, and SameSite. We also support cookie prefixes which enforce additional constraints [29]: (i) __Secure- cookies must be set with the Secure attribute and from a page served over HTTPS; (ii) __Host- cookies have all the constraints of the __Secure- attribute, plus the Path attribute must be set to the value "\"/\"" (ensuring that cookies will be attached to all requests) and must not contain a Domain attribute, thus restricting the scope to the host that set it.

D. Additional Features

Starting from the core functionalities discussed in the previous sections, our model can be extended to support other Web components, including novel security mechanisms that could benefit from the automated formal analysis enabled by WebSpec. We discuss in the following six additional Web components supported by our model.

Content Security Policy. The Content Security Policy (CSP) allows Web developers to tighten the security of Web applications by controlling which resources can be loaded and executed by the browser. Originally, the CSP was designed to mitigate content injection vulnerabilities. Subsequently, it was extended to restrict browser navigation (e.g., form-action, frame-ancestors) and protect DOM XSS sinks (via trusted-types). A CSP policy consists of a set of directives and source expressions specifying an allow-list of actions the page is allowed to perform. Currently modeled CSP directives are: script-src, which defines the allow-list for JavaScript sources, and trusted-types together with require-trusted-types-for for Trusted Types [38] support, as explained later in this section.
Service Workers and Cache API. A service worker acts as client-side proxy between Web applications and the network. Web applications are required to register a service worker, binding it to a specific origin and a scope. When enabled, a service worker can intercept and modify HTTP requests and corresponding responses initiated by its origin. Furthermore, service workers are activated also by cross-origin requests towards their registering origin. Using the Cache API, a service worker can store HTTP responses and serve them even when the network is unreachable. We reflect these capabilities in our model by considering a specific kind of service worker that can perform fetch requests, serve synthetic responses, and cache pairs of requests and responses, regardless of the scope. To this end, we also model a lightweight Cache API and assume that service workers have arbitrary access to it.

Local Scheme URLs. We model requests to local scheme URLs [61], i.e., URLs with a local scheme such as data: and blob:, as virtual requests that do not generate a response from the network. We partially support the File API [56] by enabling the creation of blob URLs via the URL.createObjectURL JavaScript method. We assume that local URLs are accepted interchangeably with remote URLs, meaning that they can be navigated by frames, or included as a script in a page. We thoroughly discuss the interaction between Content Security Policy and local scheme URLs in Section IV-C.

Local Storage. The Web Storage API [62, §12] enables JavaScript to store and retrieve key/value pairs in the browser. The API provides two mechanisms to store data: sessionStorage, an ephemeral storage that expires when the browser or the page is closed, and localStorage, which persists in the browser unless cleared explicitly. As we are interested in capturing single browser sessions, the difference between the two mechanisms is irrelevant. For this reason, we model only localStorage, providing methods to read and write data in the browser storage from any scripts.

Web Messaging. Cross-origin communication is enabled by the postMessage API [62]. As we are interested in modeling messages that are sent and received—while we ignore messages that do not reach the destination—we encode the sending and receiving of a message as a single action. We also model the origin validation process performed in the receiving script. This way, we can capture potential security issues due to cross-origin messages processed without validating the sender’s origin [52].

Trusted Types. Trusted Types are an experimental security mechanism designed to prevent DOM XSS by restricting injection sinks to accepting only non-spoofable typed values in place of strings [38]. These types can be created based on application-defined policies, allowing developers to specify via JavaScript a set of rules to protect injection sinks. Trusted Types are controlled by two CSP directives: require-trusted-types-for ’script’, which enables the enforcement of Trusted Types, i.e., instructs the browser to only accept Trusted Types for all DOM XSS injection sinks, and trusted-types, optionally followed by the name of one or more policies, which specifies the policies that are allowed to create Trusted Types objects. When no name is specified or when the special value ’none’ is used, no policy, and thus no Trusted Types, can be created, effectively disabling all DOM XSS sinks. We model the enforcement of Trusted Types on a page by mandating scripts to invoke the Trusted Type API to create a TrustedHTML object, and use it to modify the DOM via the Element.innerHTML property. Although we do not model the content of policies, we encode the ability to disallow the creation of any Trusted Types via the CSP directive trusted-types ’none’.

E. JavaScript

Contrary to previous works [17], instead of modeling scripts with an internal state and precise small-step semantics, we model them in terms of actions that the browser can perform. Since we are not interested in application-specific behavior, our abstraction captures the execution of sequences of Web API calls and the evolution of the browser’s state. For example, we model the fact that a script can set a cookie, or add a request-response pair to a cache using the Cache API, but we do not model how cookie data, requests, or responses are built. Instead, we introduce symbolic variables with constraints following the API specification.

Scripts in our model can update the DOM, set and get cookies using Document.cookie, or navigate frames using the Window.location setter. They can use the Fetch API to perform network requests and read the corresponding responses. We also model the legacy Document.domain API, which allows for cross-origin communication between windows in the same site by relaxing their document.domain property to a common ancestor. Although this API has been deprecated due to security concerns [42], and Google announced that it will be disabled by default starting from Chrome 109 [37], it is still supported by all major browsers. Scripts can also use the APIs described in Section II-D: they can update a cache from page context using Cache.put, communicate with other windows using Window.postMessage, create blob URLs with URL.createObjectURL, and create Trusted Types.

III. WebSpec Toolchain

In the following, we present the WebSpec toolchain (Figure 1). This toolchain comprises (i) a compiler (Section III-A) which translates the browser model and the Web invariants written in Coq into a query that can be automatically checked by an SMT solver – Z3 in the current implementation (Section III-B) and (ii) a verifier which reconstructs from a SMT solution an attack trace enjoying correctness and minimality, and validates it against real Web browser (Section III-C). Finally we discuss the advantages of our approach as compared to prior work (Section III-D).

A. Compilation

To automatically verify the (in)validity of our invariants, we developed a compiler that translates the Coq model and the invariants into SMT-lib formulas, which are then fed to the Z3 solver. Technically, we compile Coq inductive types into...
We encode this invariant in WebSpec as follows:

\[
\text{We encode this invariant in WebSpec as follows:}
\]

```
1 Definition SecureCookiesInvariant (gb: Global) (evs: list Event) (st: State): Prop :=
2  \forall rp corr _evs cookie,
3  Reachable gb evs st →
4  evs = (EvResponse rp corr _evs) →
5  rp_hd_set_cookie (rp_headers rp) = Some cookie →
6  sc_secure cookie = true →
7  url_protocol (rp_url rp) = ProtocolHTTPS.
```

This definition says that for every reachable state where the browser handles a network response, i.e., the state is Reachable and the current event is EvResponse (lines 2-4), if the response contains a Set-Cookie header (line 5) with a cookie that has the Secure attribute (line 6), then the protocol used to serve the response is HTTPS (line 7).

We encode a query for finding a counterexample to this invariant with the following Coq inductive type:

```
1 Inductive SecureCookiesQuery (gb: Global) (evs: list Event) (st: State): Prop :=
2  \forall Query_state: \forall rp corr _evs cookie,
3  Reachable gb evs st →
4  evs = (EvResponse rp corr _evs) →
5  rp_hd_set_cookie (rp_headers rp) = Some cookie →
6  sc_secure cookie = true →
7  url_protocol (rp_url rp) ≠ ProtocolHTTPS →
8  SecureCookiesQuery gb evs st.
```

This inductive type definition is essentially identical to SecureCookieInvariant, except the negation of the conclusion (line 7, we require the protocol to be ≠ HTTPS). The following theorem formalizes that inhabitants of SecureCookiesQuery are indeed counterexamples of SecureCookiesInvariant:

```
1 Theorem secure_cookies_query_invalidate_invariant :
2  \forall gb evs st, SecureCookiesQuery gb evs st →
3  not (SecureCookiesInvariant gb evs st).
```

B. SMT Solving and Trace Reconstruction

The compilation of the inductive Reachable relation results in a recursive CHC formula, which cannot be handled by standard SMT solvers. Therefore we use the \( \mu Z \) extension (satisfiability modulo least fixed-points [32]) of the Z3 theorem prover. More precisely, we run in parallel the Spacer engine of \( \mu Z \), a generalized property-directed reachability (GPDR) model checker suitable for finding proofs [31], and the bounded model checking (BMC) engine of \( \mu Z \), designed to find counterexamples. The four possible outcomes are:

**SAT** \( \mu Z \) finds a counterexample, hence the invariant does not hold. We discuss in Section [V] the security implications of violating an invariant.

**UNKNOWN** \( \mu Z \) fails to find a counterexample or to prove its absence. In such a case, which never happened in our case studies, we cannot draw any conclusion.

**UNSAT** \( \mu Z \) proves that there is no counterexample. Although this does not formally suffice to conclude that the invariant holds in our model since neither our compiler nor \( \mu Z \) are formally verified, this gives us strong confidence that this is the case. A formal proof in Coq can be manually produced, if stronger confidence is needed.

**LOOP** \( \mu Z \) does not terminate. Due to the way the BMC engine works, this means that \( \mu Z \) did not find a counterexample after exploring a certain number of steps. When this number becomes high enough, though it is not a proof, it gives us a good intuition that the invariant is likely to hold, and hints it is worth starting a formal proof, as shown in Section [V].

When running WebSpec on SecureCookiesQuery, Spacer proves that there is no counterexample within 2min, while in the same time, BMC reaches a trace size of 50 events without detecting any attack. Moreover, we constructed a formal proof in Coq that the invariant indeed holds [7].

In the case \( \mu Z \) finds a counterexample (SAT), we first automatically extract an attack trace from it. It is worth noting that this attack trace enjoys the property of being minimal. This property is due to the decision procedure implemented in the BMC engine of \( \mu Z \), which ensures that the list of events in the counterexample is the smallest one that leads to a contradiction of the invariant. Then, we verify its correctness by automatically translating it back into a Coq term and checking whether the trace produces an inhabitant of the query. We take this precaution because, as mentioned previously, neither our compiler nor \( \mu Z \) are formally verified. Since \( \mu Z \) instantiates all symbolic variables, the proof is straightforward and mostly automatic. WebSpec then renders this trace as a sequence diagram, making the representation of the counterexample accessible to users unfamiliar with formal verification. Examples of such diagrams are given in Section [IV].

C. Trace Validation

In addition to rendering sequence diagrams, WebSpec includes a verifier to validate the discovered attack traces against real-world browser implementations (Chrome, Firefox). Our verifier consists of approximately 3500 lines of OCaml
code and leverages the Web Platform Tests (WPT) cross-browser framework to map attack traces to tests, enabling verification against all major browsers. WPT is the standard test suite for browser and specification developers. It allows browser vendors to write tests modeling the expected behavior of Web standards and test their implementations for compliance. Using the common format specified by the WPT framework makes WebSpec tests compatible with the test suite. This allows for easy triage of the attack traces extracted from counterexamples and the inclusion of our cross-component tests into WPT.

The generated tests translate trace events to browser actions and server responses. These actions modify the browser state to match the model’s state after a given event. To maintain the browser and model states consistent throughout test execution, the verifier ensures these actions execute in the correct order. The effects of these actions are collected (directly or indirectly) and verified via WPT assertions. If all assertions succeed, the test is passing and the trace is considered valid, whereas the test is failing if one assertion fails. We map each event to a <Setup, Action, Verification> (SAV) tuple:

Setup. A set of pre-conditions necessary to execute an event, e.g., for EvWorkerCacheMatch to happen, a service worker must be installed in the matched URL’s scope.

Action. Any browser or server action, from top-level navigations and JavaScript API calls to server responses. Actions can be implicit or explicit: Explicit actions require an explicit API call or client-originated event, like window navigation; Implicit actions are actions triggered by other actions (implicit or explicit), like subresource loading.

Verification. A means to ensure that an action succeeded. Explicit verifications map to WPT assertions and are the primary mechanism for verifying attack traces. Asserting the value of a cookie after an EvScriptSetCookie event is an example of an explicit verification. Implicit verifications do not require a corresponding WPT assertion. Server responses are an example of actions with implicit verifications, as they are not browser-dependent behavior and are known before runtime.

Since events are state transitions in the model, corresponding Actions must change the concrete state of the browser accordingly. By enforcing that Actions occur in the correct order, we guarantee that the model and the concrete browser’s state match throughout the test execution, reaching a final state that, e.g., violates a Web invariant. Precisely mapping trace events to SAV tuples allows us to generate executable tests that can perform and verify each event in a trace, thus ensuring the correspondence between test executions and traces. We refer to Appendix D for a detailed example of how an attack trace presented in Section IV is mapped to an executable test.

The following three paragraphs discuss some non-trivial implementation details necessary for trace verification:

Script Construction and Serialization. In our model, an EvScript* event is associated with a DOMP� (Figure 4) which identifies the script element performing the action in the current window’s DOM. The verifier keeps an ordered list of EvScript* events for each script element in a trace. Since each EvScript* event maps to a specific sequence of JavaScript methods, these events can be serialized. This serialization is also performed in order, adding verification and synchronization code between event actions when necessary.

Event Synchronization. To keep tests as close as possible to the attack traces, the verifier must ensure that the actions corresponding to trace events are executed in the correct order. For instance, assume that a script in a page caches a response to an HTTP request, and such entry in the cache is subsequently processed by a service worker after matching on the corresponding request. Without synchronization, the service worker matching could occur before the cache update, leading to inconsistent test results. Ordering is implemented by enforcing explicit dependencies between actions, e.g., action $a_j$ must be preceded by action $a_i$. This dependency relation is implemented by writing a nonce to a global key-value storage from $a_i$. Then, $a_j$ is allowed to execute only after successfully reading the same nonce from the storage.

Content-Security-Policy Inference. Invariants regarding the CSP impose a challenge on verification as browsers do not allow direct access to the CSP via JavaScript. Therefore, to verify the value of the CSP of a given browsing context, we must infer it. The verifier generates a set of URLs that the CSP should allow or block and calculates a signature representing this allow/block pattern. Pages updated via a EvScriptDOMUpdate then attempt to load scripts from these URLs. The verifier adds a report-uri field to the CSP for the server component to know both allowed and blocked requests. The server can then calculate the CSP signature and report it to the main page, which asserts its value. If it matches, we conclude that the CSP matches the one in the trace.

D. Discussion

The choice of Coq for the formalization of a Web browser brings two advantages: First, using a strictly-typed higher-order language as a specification language makes it possible to write expressive and parametrizable models which can easily and consistently be extended to new Web features. Second, having our model specified within a proof assistant allows us to write fully machine-checked proofs when the highest level of confidence is required. We developed such proof for the four changes we propose to fix the vulnerabilities that are currently affecting the Web platform.

In general, the main drawback of a model written in Coq is the lack of automation, which becomes particularly problematic when the model is constantly evolving to reflect the regular Web platform updates. Compiling our Web browser model from Coq to SMT queries circumvents the issue by providing not only automatic counterexample finding as in [9], but also automated proofs that counterexamples do not exist.

Finally, a common limitation of model-based security analysis is the lack of evidence of compliance between models and reality. In WebSpec, we avoid this pitfall by using of a verifier (Section III-C) which allows us to execute and validate the discovered attack traces against real browsers. In particular, we are able to validate the five vulnerabilities found in our
model of the current Web platform by running the attack traces produced by WebSpec against the latest versions of Chrome and Firefox. Additionally, the verifier enables automated testing of the semantic rules of the browser model, ensuring that our model matches the observable behavior of real browsers. For every Web component in our model, we query for a reachable state which makes use of the modeled feature. Then we validate, using the verifier, that the obtained state maps to a reachable concrete state across browser implementations. Although this does not correspond to a correctness proof, it provides empirical evidence that our modeling is consistent with browser behavior.

IV. WEB INVARIANTS AND ATTACKS

We define 10 Web invariants concerning the security properties of 5 core Web components: cookies, CSP, Origin header, SOP, and CORS. Table I presents an overview of the invariants that we formally encode in our model. For each invariant that does not hold in the current Web platform, WebSpec is able to find a counterexample that translates to a concrete attack. When the invariant holds, WebSpec can be configured to reflect a past state of the Web that was affected by a vulnerability, confirming that our approach can identify previously reported attacks. In this section, we focus on three invariants that do not hold in the current Web platform, showing how WebSpec is able to discover a new attack on the __Host- prefix for cookies as well as a new inconsistency between the Content Security Policy and a planned change in the HTML standard. We also present an attack against Trusted Types for which we propose a mitigation in Section V. Due to space constraints, we do not hold in the current Web platform in Appendix A, while we present the full set of invariants in the technical report [8].

A. Integrity of __Host- Cookies

The invariant stipulates that __Host- cookies ensure integrity against same-site attackers [51]. When a cookie whose name starts with __Host- is set, the browser verifies that the Domain attribute is not present and discards the cookie otherwise, thus marking all __Host- cookie as host-only.

Invariant 1.3. A __Host- cookie set for the domain d can be set either by d (via HTTP headers) or by scripts included by the pages on d.

We encode this invariant by splitting the two cases in which a host cookie can be set: (i) via HTTP headers, and (ii) via JavaScript. For space reasons we present only case (ii) below and refer to the technical report [8] for the full definition.

For every reachable state in which a script sc is setting a cookie on the top-level window (lines 3-6), ctx is the window (browsing context) in which the script sc is running (line 5). If the cookie has the __Host- prefix (line 8), we require (line 11) the domain on which the cookie was registered to be equivalent to the domain of the ctx browsing context. This corresponds to stating that a script running on a page of domain d can set a host-prefix cookie only for the domain d.

Attack. When we run the query, our toolchain discovers a novel attack that breaks the invariant using domain relaxation. A script running on a page can modify at runtime the effective domain used for SOP checks through the document.domain API. Indeed, the value of document.domain is taken into account only for DOM access. All remaining access control policies implemented in the browser, e.g., for cookie jar access, XMLHttpRequests, and the origin information reported when performing a postMessage, use the original domain value [48]. The mismatch between the access control policies in the DOM and the cookie jar allows a script running in an iframe to access the document.cookie property of the parent page when both pages set document.domain to the same value. Once the inner frame performs a set cookie of a host-prefix cookie through the parent page DOM, the browser uses the original domain value of the parent page to perform the host prefix checks, breaking the invariant.

The trace generated by WebSpec is shown in Figure 4 and detailed below. In the following, expressions of the form DOMPath [ _ _ ] _ _ represent a unique path in the DOM. In particular, the first argument of DOMPath is the nesting level. For instance, we refer to the window loaded inside two nested iframes as DOMPath [1,3] [1,2], where 1 and 3 are the indexes of the DOM elements representing the frames. The second argument is used to refer to a specific DOM object (DOMIndex) or to the whole document loaded in the frame (DOMTopLevel). An example is shown in Figure 3 the path to an image at index 3 loaded inside two nested iframes (respectively at index 2 and 1) is represented as DOMPath [1,2] (DOMIndex 3), while the path of the window containing the image is DOMPath [1,2] DOMTopLevel.

The attack trace describes the following scenario: (steps 1-3) a page from origin_1 is loaded in the top-level window of the browser. Note that origin_1 is the subdomain 16162 of the host 13, loaded via HTTPS; (4-6) an iframe element is loaded from origin_4 at index 0 of the DOM in the main window (in the path DOMPath [ ] (DOMIndex 0)). origin_4 is another subdomain of the same host; (7-9) a script is loaded in the main window at index 1; (10-12) a script is loaded in the
The authorization to perform a non-simple request towards a certain origin

The DOM of a page protected by CSP can be read/modified only by the scripts allowed by the policy. The DOM XSS protection despite the fact that the document was downloaded using a secure connection. When

Access control on Trusted Types DOM sinks

Interaction with SOP

If a request from the server contains a security policy, then the browser enforces that specific policy. This was part of an effort of browser vendors to restrict all new APIs to secure contexts to help advance the Web platform to default to the HTTPS protocol. The restriction, however, enabled attackers to bypass Trusted Types by framing the protected page from a non-secure context. This silently disabled the DOM XSS protection despite the fact that the document was downloaded using a secure connection. When

Origin Header

Authenticity of request initiator

If a request r includes the header Origin: o (with o ≠ null), then r was generated by origin o.

SOP/CORS

Authorization of non-simple requests (i)

A non-simple cross-origin request must be preceded by a pre-flight request.

Authorization of non-simple requests (ii)

The authorization to perform a non-simple request towards a certain origin o should come from o itself.

TABLE I

<table>
<thead>
<tr>
<th>Web Feature</th>
<th>Invariant</th>
<th>Description</th>
<th>Holds</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cookies</td>
<td>I.1</td>
<td>Integrity of Secure cookies (network)</td>
<td>Cookies with the Secure attribute can only be set (using the Set-Cookie header) over secure channels.</td>
<td>☐</td>
</tr>
<tr>
<td></td>
<td>I.2</td>
<td>Confidentiality of HttpOnly cookies (Web)</td>
<td>Scripts can only access the cookies without the HttpOnly attribute.</td>
<td>☐</td>
</tr>
<tr>
<td></td>
<td>I.3</td>
<td>Integrity of __Host--cookies</td>
<td>A __Host--cookie set for the domain d can be set either by d (via HTTP headers) or by scripts included by the pages on d.</td>
<td>☐</td>
</tr>
<tr>
<td></td>
<td>I.4</td>
<td>Interaction with SOP</td>
<td>The DOM of a page protected by CSP can be read/modified only by the scripts allowed by the policy.</td>
<td>☐</td>
</tr>
<tr>
<td></td>
<td>I.5</td>
<td>Integrity of server-provided policies</td>
<td>If a page has both trusted-types; and require-trusted-types-for ‘script’ directives in the CSP then no script in the page can modify the DOM using a Trusted Types sink.</td>
<td>☐</td>
</tr>
<tr>
<td></td>
<td>I.6</td>
<td>Access control on Trusted Types DOM sinks</td>
<td>Documents loaded from a local scheme inherit the policy of the source browsing context.</td>
<td>☐</td>
</tr>
<tr>
<td></td>
<td>I.7</td>
<td>Safe policy inheritance</td>
<td></td>
<td>☐</td>
</tr>
<tr>
<td></td>
<td>I.8</td>
<td>Authenticity of request initiator</td>
<td>If a request r includes the header Origin: o (with o ≠ null), then r was generated by origin o.</td>
<td>☐</td>
</tr>
<tr>
<td></td>
<td>I.9</td>
<td>Authorization of non-simple requests (i)</td>
<td>A non-simple cross-origin request must be preceded by a pre-flight request.</td>
<td>☐</td>
</tr>
<tr>
<td></td>
<td>I.10</td>
<td>Authorization of non-simple requests (ii)</td>
<td>The authorization to perform a non-simple request towards a certain origin o should come from o itself.</td>
<td>☐</td>
</tr>
</tbody>
</table>

I.6. If a page has both trusted-types; and require-trusted-types-for ‘script’ directives in the CSP then no script in the page can modify the DOM using a Trusted Types sink.

We encode the invariant in our model as follows:

```
1 Definition TTInvariant (gb: Global) (evs: list Event) (st: State): Prop = 
2 ∀ pt target_pt target_ctx ssrc ttty types, 
3 Reachable gb evs st → 
4 url_protocol (wd_document_target_ctx) = ProtocolHTTPS → 
5 rp_ha_csp (dc_headers (wd_document_target_ctx)) = Some 
6 { csp_script_src = ssrc; csp_trusted_types := Some 
7 ttypes |} → 
8 tt_policy types = Some None → 
9 tt_require_for_script ttypes = true → 
10 (No script can update the dom using innerHTML) → 
11 (No script can update the dom using innerHTML) → 
```

Here we assert that there cannot be an html update event (using a DOM XSS sink, e.g., innerHTML) for the window target_ctx, if the aforementioned directives are used to define the policy for target_ctx.

Attack. An earlier version of the Trusted Types draft [38] Editor’s Draft, Feb. 3, 2021] restricted Trusted Types to Secure Contexts only. This was part of an effort of browser vendors to restrict all new APIs to secure contexts to help advance the Web platform to default to the HTTPS protocol. The restriction, however, enabled attackers to bypass Trusted Types by framing the protected page from a non-secure context. This silently disabled the DOM XSS protection despite the fact that the document was downloaded using a secure connection. When

iframe at index 0 (DOMPath [0] (DOMIndex 0)); (13) the script in the main window sets its document.domain to its parent domain; (14) the script in the iframe sets its document.domain to its parent domain. The two pages are now effectively same origin, having performed domain relaxation to the same domain; (15) the script in the iframe (DOMPath [0] (DOMIndex 0)) sets a cookie using document.cookie of the top-level window (DOMPath [] DOMTopLevel). The cookie has the __Host--prefix and has been set by origin_1 for origin_4, breaking the invariant.

Although the current Web platform is still vulnerable to

Fig. 4. Host Cookies Inconsistency
we enable the secure context restriction in our model, WebSpec is able to rediscover the bypass.

We can disable the secure context restriction with the _c_restrict_tt_to_secure_contexts_ (config gb)=false configuration option. However, when we rerun the solver with this configuration, our toolchain can still find a counterexample for which the invariant does not hold. The trace is shown in Figure 5 (steps 1-3) a page protected with Trusted Types is loaded from origin_1. In particular, no policy is allowed, so no Trusted Type can be created; (4-6) the page contains a same-origin iframe which specifies a Trusted Types policy (trusted-types 25809), allowing scripts loaded in this iframe to create Trusted Types using a policy named 25809; (7-9) a script that is loaded in the iframe modifies (10) the DOM of the parent frame using a Trusted Types sink. This is possible because the inner frame is able to create Trusted Types that are accepted by all DOM XSS sinks and because, being same origin, the inner frame can access the DOM of the parent. A similar attack on related domains is possible if the parent page performs domain relaxation, as the value of document.domain is used for DOM access control.

The Trusted Types draft [33, §5.1] includes a brief discussion of a similar attack in which cross-document import of nodes would bypass the enforcement of the policy. However, the current specification does not provide any solution and suggests that other mechanisms like Origin Policy [25] might be used to ensure that the same policy is deployed across the whole origin. Instead, we propose a different solution based on non-transferable Trusted Types, and prove the correctness of our approach within our model in Section [V].

C. Safe policy inheritance

The Content Security Policy specification [65, §7.8] mandates that every document loaded from a local scheme must inherit a copy of the policies of the source browsing context, i.e., the browsing context that was responsible for starting the navigation. This corresponds to the following invariant.

**Invariant I.7.** Documents loaded from a local scheme inherit the policy of the source browsing context.

We encode the invariant in our model as follows:

1. Definition LSInvariant (gb: Global) (evs: list Event) (st: State): Prop :=
2. ∀ evs pt _evs rm fhtml fwd ctx lv pt_idx init_idx, take get_csp wd :=
3. let get_csp wd :=
4. rp_hd_csp (dc_headers (wd_document wd)) in
5. reachable gb evs st →
6. (∀ A document has just been loaded in a frame *)
7. evs = (EvDOMUpdate pt :: _evs) →
8. is_frame_in_dom_path qb (st_window st) pt fhtml fwd ctv →
9. is_local_scheme (wd_location fwd) →
10. (∗ A document has just been loaded in a frame *)
11. pt = DOMPath lv (DOMIndex pt_idx) →
12. is_wd_initiator_of_idx ctv pt_idx (Some init_idx) →
13. (∗ The csp is equal to the req. initiator *)
14. get_csp fwd = get_csp (windows gb.init_idx).

When a frame has just loaded a document from a local scheme (lines 7-9), we require that the CSP of the navigation initiator (i.e., the source browsing context) is equal to the policy of the document loaded in the frame window (line 14).

The goal of this Web invariant is to ensure that a page cannot bypass its policy by navigating to content that is completely under its control. One such bypasses [2] was caused by the behavior defined for the inheritance of policies in a previous version of the CSP specification [55, Oct. 15, 2018]: documents loaded from local schemes would inherit the policies of the embedding document or the opener browsing context.

Recently, the concept of policy container was added to the HTML specification [62, §7.9]. A policy container is a collection of policies to be applied to a specific document and its purpose is to simplify the initialization and inheritance of policies. The introduction of the policy container in the specification allowed for clarifying the inheritance behavior for local schemes, which might differ depending on the specific scheme or URL that is used. The policy container explanation [3] stipulates the following behavior:

**about::srcdoc** An iframe element with the srcdoc attribute inherits the policies from the embedding document, i.e., the parent frame. Note that srcdoc iframes are in the same origin of the embedding document but their location URL is about::srcdoc.

**about:: data**: A document loaded from the data: or about: schemes inherits the policies of the navigator initiator (as mandated by the CSP specification).

**blob**: A document loaded from a blob: URL inherits the policies from the document that creates the URL, i.e., the document that calls the URL.createObjectURL function.

Note that in the current version of the HTML specification [62, §7.11.1] the inheritance behavior for blob: URLs matches the one for about: and data:, thus following the CSP specification. We contacted the editors of the HTML specification [4] asking for a clarification on the correct behavior for blob: URL and they confirmed that, because of the wrong ordering of a clause in the policy container.
construction for blobs, the initiator policy container was always replacing the creator policy container. The correct inheritance rule is to inherit the policy container of the creator of the URL \( \text{blob} \), thus introducing an inconsistency between the CSP specification and the HTML specification (as \( \text{blob} \) is a local scheme that is handled differently from the others).

**Attack.** When we configure our model to reflect a past state of the Web platform in which policies were inherited from the embedding frame and not from the navigation initiator \( \text{c_csp_inherit_local_from_initiator (config gb)=false} \), our toolchain is able to rediscover the attack trace that allows an attacker to strip the CSP policies by navigating to a local scheme URL. The trace is shown in Figure 6 (steps 1-6) with no Content Security Policy loads an iframe with a restrictive CSP; (7-9) the iframe contains a script which navigates (10) the frame itself (e.g., using the \( \text{window.location} \) setter) to a local scheme URL; (11-13) the iframe renders the content of the local scheme URL and inherits the CSP from the embedding document, which contains no policy. The resulting document has no CSP, effectively removing the iframe’s previous policy.

We can configure our model to reflect the current state of the Web platform by inheriting the policies from the navigation initiator for all local schemes:

\[
\text{c_csp_inherit_local_from_initiator (config gb)=true \land \\
\text{c_csp_inherit_blob_from_creator (config gb)=false.}
\]

We can verify (up to a finite size, see Section VI) that with this configuration the invariant holds. However, when we configure our model to reflect the planned modification of inheriting the policies of the URL creator when rendering a blob: URL \( \text{c_csp_inherit_blob_from_creator (config gb)=true} \), Z3 is able to find a new counterexample. The trace is similar to the one depicted in Figure 6 with an additional script loaded in the top-level window executing an \( \text{EvScriptCreateBlobUrl} \) event: (i) a page in origin_1 with no CSP loads a same-origin iframe with a restrictive policy; (ii) a script running on the embedding document creates a new blob URL \( \text{EvScriptCreateBlobUrl} \); (iii) a script running on the inner frame navigates the frame itself (i.e., setting \( \text{window.location} \)) to the previously created URL. The frame loads the content of the blob and inherits the CSP from the embedding document, which does not have any policy. Similarly to the previous attack trace, the policy that was defined for the iframe has been removed by navigating to local-scheme content.

Hence, the planned modification on CSP inheritance in the HTML standard would introduce an inconsistency with the CSP specification. We responsibly reported the issue to the working group of the HTML standard [4], who initially deemed the security implications of the attack as low. However, at the time of writing (August 2022), our invariant still holds since no final decision has been taken.

V. Verification of Security Properties

WebSpec supports both machine-checked and automated Web invariant proof generation. In particular, we wrote four machine-checked proofs in Coq and were able to automatically derive two further Web invariant proofs through the Spacer engine of \( \mu Z \). Due to space constraints, we show here how to formally verify the security of the fix we present against the attack on Trusted Types (Section IV-B) through a machine-checked proof in WebSpec, referring to the long version [8] and to the online repository [7] for the remaining proofs.

According to the current draft of the Trusted Types specification, a Trusted Type object created by a page can be assigned to DOM XSS sinks belonging to different pages. This allows for bypassing the protection if a restricted document colludes with an unrestricted one. This can happen, e.g., in case of same-origin iframes (see Section IV-B). The specification acknowledges the issue and suggests the usage of the Origin Policy [25] to address the problem, which unfortunately is not currently supported by any browser.

For this reason, we propose an alternative solution that we label *non-transferable* Trusted Types, which consists in labeling each Trusted Type with the JavaScript realm (window or worker) that created it and ensuring that a type can only be assigned to DOM XSS sinks from the same realm. Therefore, our fix prevents the cross-document usage of Trusted Types. We implemented this behavior in WebSpec, which can be activated by setting the configuration option \( \text{c_tt_strict_realm_check=true} \) when our fix is enabled:

1. \( \text{Theorem strict_realm_check_implies_invariant :} \)
2. \( \forall \text{gb evs st,} \)
3. \( \text{c_tt_strict_realm_check (config gb)=true} \to \)
4. \( \text{c_restrict_tt_to_secure_contexts (config gb)=false} \to \)
5. \( \text{TTInvariant gb evs st.} \)
We recall that, according to the invariant, if a page is shipped with a CSP containing the directives trusted-types and require-trusted-types-for 'script', then the list of events evs cannot contain a EvScriptUpdate-HTML event that updates the contents of the page.

We can prove the theorem by induction on the Reachable relation where all the cases except EvScriptUpdateHTML are trivial. In this latter case we show that, by enabling strict realm checking, it is impossible to generate the correct Trusted Type for the update, since (i) the trusted-types directive disallows the creation of Trusted Types for the realm in which the directive is used; and (ii) the only Trusted Types that are accepted by a page with require-trusted-types-for 'script' are only those labelled with the realm of the page. This suffices to prove the correctness of the proposed solution within our model. The whole proof in Coq spans just over 54 LOC. For comparison, the longest of the four proofs we conducted in Coq is just 348 LOC, which demonstrates the feasibility of writing machine-checked proofs in our model.

VI. EVALUATION

In our experimental evaluation, we used WebSpec to automatically discover the attacks reported in Table I. Additionally, when we implemented a fix to an attack, we ran WebSpec again to confirm that the issue had been addressed. Since the μZ solver may not terminate (see Section III), we use the length of the previously discovered attack trace plus one as the maximal search size, thus obtaining confidence that the previous attack is not reachable anymore.

We report the time required by WebSpec to find the attacks and describe various optimization techniques that allowed us to drastically improve the performance of our approach. All our experiments have been conducted on a virtual machine with 32 VCPUs (2GHz AMD EPYC) and 128GB of RAM.

The baseline performance is displayed on the third column of Table I. We can observe a clear correlation between the size of the attack trace and the time required to find an attack, which is caused by the unrolling technique employed by the BMC engine of the μZ solver used in WebSpec. In particular, time increases exponentially with respect to the size of the attack trace, leading to running times of several days or weeks for traces with 10 or more events. To tackle these performance issues, we have implemented various optimizations that consist in (i) defining additional rules (or lemmas) representing common configurations (e.g., loading of a frame containing a script) that can be used by the SMT solver instead letting it rediscover the right list of events leading to these configurations, and (ii) simplifying the model at compile time (e.g., by disabling frames) so that the resulting SMT formula is easier to solve.

We describe these optimizations in the following, and we refer the reader to Appendix C for a discussion on the scalability of our browser model, which confirms the effectiveness of lemmas in mitigating the complexity that arises from the addition of new Web components.

A. User-defined Lemmas

The key idea underlying this optimization is to enable users to define additional lemmas that guide the solver into constructing interesting browser states that can be used as a starting point to discover new attacks. Intuitively, lemmas represent encodings of common Web Security threat models, which map to the general preconditions of specific classes of Web attacks. Consider the following example:

```
1  lemma script_state_is_reachable : \forall gb.
2  script_state_constraints gb \rightarrow
3  Reachable gb (script_state_events gb) script_state.
```

Here script_state is a Coq definition of a concrete browser state where a script is loaded in the page rendered in the top-level window. This state maps to the threat model in which an attacker controls a script running in the same page as the target Web application, as a result of, e.g., XSS or the inclusion of untrusted scripts [44]. The lemma encodes that this browser state is reachable by applying the list of events script_state_events assuming that script_state_constraints is satisfied. Once we prove that the state defined by a lemma is Reachable, the lemma can be compiled as an additional model rule to CHC logic. Since the BMC engine solves queries by iterative unrolling, it prioritizes the rules that result in the smallest amount of unrolling steps. Lemmas exploit this property by providing a one-step solution for the generation of states that would require multiple steps if the solver had to build them from scratch.

WebSpec includes the definition of five generic lemmas, which represent two variants of the aforementioned attacker controlled script, one running in a secure context and the second being served from an insecure connection, and three variants of the gadget attacker of [15] in which a same-origin, same-site or cross-site iframe containing an attacker controlled script is included into the target Web applications. When compiling the model to an SMT formula, the Using Lemmas directive is used to specify which user-defined lemmas need to be considered for the query. For our experiments, we provided every query with all the lemmas that are included in WebSpec, since we experimentally observed that the number of lemmas added to a query does not negatively impact solving time. This behavior may result from the effectiveness of μZ in discarding all non applicable initial states. For this reason, we expect only an improvement of the solver performance from the definition of a larger library of lemmas.

The results of this optimization are highlighted in Table III where we can see that the usage of lemmas always reduces the runtime to less than a day. It may however happen that the solver is not able to apply any of the user-defined lemmas, as it is the case for queries #5 and #6 (marked with –). In such cases no performance improvement can be obtained. For the queries marked with *, we automatically extracted a lemma from an attack trace discovered by WebSpec and confirmed that it can be used by successive runs of the solver. The extraction of lemmas from traces has several benefits: on the one hand, it allows us to quickly test for the absence of a known attack after applying a fix to the model; on the other
hand, it allows us to add new reachable browser configurations to our library of lemmas. Since these configurations represent the preconditions for the execution of a specific attack, the extraction of additional lemmas augments our counter example pipeline with a method to quickly discover novel attacks assuming known preconditions. We leave the extension of this library and the definition of a methodology to generate generic lemmas as future work.

B. Compile-Time Simplification

Configurable inlining of auxiliary relations. Our model relies on aReachable relation that models state transitions, a ScriptState relation that models scripts knowledge, and multiple auxiliary relations that are used within Reachable to, e.g., recursively update the DOM. The presence of multiple relations prevents us to directly use the best performing version of the BMC engine, the linear solver, because it requires the model to be encoded as linear Horn clauses, i.e., clauses containing at most one recursive term. In order to satisfy this requirement, every auxiliary relation needs to be inlined within the main Reachable relation. To this end, WebSpec automatically unrolls all the applications of recursive relations that are marked for inlining. For each relation we specify the depth of the unrolling. For instance, the declaration

```
Inline Relation is_script_in_dom_path With Depth 3.
```

says that the relation is_script_in_dom_path, which searches a script inside the DOM, must be unrolled up to recursion level 3. Depth 0 disables all recursive calls and expands the relation to the base case only. For instance, support for nested frames can be easily deactivated by specifying 0 as depth for all the relations handling the DOM tree.

The recursion depth affects the solving time of $\mu Z$ since multiple applications of the relation need to be considered. Disabling nested frames for the queries which do not require them simplifies the compiled model and allows for faster solving. When frames are required, we set the Depth parameter so that a single level of nesting is allowed. Although our model can handle an arbitrary number of nested frames, a single level suffices to discover the minimum-size trace for all queries.

The effects of this optimization are shown in Table II. We can see that disabling framing for the queries that do not require multilevel DOM trees considerably lowers the solving time.

### Fixed size arrays.

Our model uses functional arrays [40], [47] for the definition of the HTML and DOM objects and the implementation of the window/frame tree. However, functional arrays are known for significantly increasing the complexity of queries [54], [53]. Therefore, in order to ease the resolution, our compiler provides an optimization which turns functional arrays into arrays of a fixed size chosen at compile time. Since choosing a small size could make a query unsolvable, we run in parallel several instances of the same query with different sizes and keep the first one that succeeds. Surprisingly, a size of 5 is enough for all the queries except those for Safe policy inheritance (#7 and #8 in Table II) which require a size of 7.

### VII. RELATED WORK

#### Models of the Browser.

Bohannon [17] proposed Featherweight Firefox, a model of a Web browser written in Coq for the verification of security properties concerning JavaScript execution. The model supports several JavaScript features, such as DOM manipulation, XHR requests, event listeners, and code evaluation via the eval function. However, the set of modeled Web components comprises only windows, cookies, and selected HTML tags (<script>, <div>). Bugliesi et al. [13] extended Featherweight Firefox to formalize the security guarantees of HttpOnly and Secure cookies against network and Web attackers able to exploit XSS vulnerabilities. In [19] the authors use Featherweight Firefox as a starting point to develop a pen-and-paper model of a security-enhanced browser which enforces a Web session integrity property that captures attacks like CSRF and credential theft via XSS.

In contrast to WebSpec, Bohannon’s model and later extensions were developed with machine-checked proofs in mind and have not been used to automatically detect vulnerabilities. They also lack support for most of the Web features considered in our invariants, e.g., CORS, CSP, service workers. Because of their focus on Web script security, they formalize script executions using a small-step semantics. This choice allows
for a precise modeling of JavaScript but hinders automatic verification, as it forces solvers to handle JavaScript programs.

**Models of the Web.** In their seminal work, Akhawe et al. [9] developed the first formal model of the Web ecosystem. The authors encoded in the model a set of security goals, which include fundamental properties of the Web platform that are assumed to hold, and a notion of session integrity capturing CSRF attacks. The validity of these goals has been checked with the Alloy Analyzer [34] and their violations pointed out the existence of novel and previously known attacks.

Despite being similar in spirit to our proposal, there are important differences between WebSpec and the model of Akhawe et al. First, the model cannot be used to prove security properties, since the Alloy Analyzer uses SAT-based bounded model checking, but just to disprove them, while WebSpec can be used also to produce automated or machine-checked proofs. Second, being developed in 2010, it lacks many features of the modern Web (e.g., frames, CSP and service workers) that are a fundamental part of our model. Adding these features a posteriori would not be possible without rewriting the model from scratch, since some of them (e.g., a faithful handling of frames) are core components of Web browsers. Last, contrary to WebSpec, the model of Akhawe et al. is stateless. For this reason, temporal relations between events, e.g., the correct sequencing of requests and response pairs, need to be explicitly modeled. Considering that Web Standards are typically written using a stateful imperative style, a more natural modeling follows a stateful approach, as employed in our model.

Bansal et al. [11] developed WebSpi, a generic library that defines the basic components of the Web infrastructure (browsers, HTTP servers) and enables developers to automatically verify security properties of specific Web applications / protocols using ProVerif [16]. The browser model of WebSpi is rather primitive and includes a subset of the features supported in WebSpec. This is in line with the intended usage of WebSpi, i.e., the verification of Web protocols, for which it suffices to model only the features used by the protocol under analysis. Instead, we target inconsistencies between Web features themselves, without focusing on a specific Web protocol or application, for which we need a much more comprehensive browser model. Similarly to WebSpi, WebSpec supports automated security proofs: if this does not succeed, however, we can still fall back to machine-checked proofs in Coq.

The most comprehensive and maintained model of the Web to date is the Web Infrastructure Model (WIM), a pen-and-paper model which has been used to assess the security of Web Payment APIs [26], Web protocols, e.g., OAuth 2.0 [29], OpenID Connect [30], and the Financial-Grade APIs [28]. The browser model of WebSpec supports most of the features of WIM browsers, except for (i) HSTS, since in our model we abstract away the network, (ii) HTTP basic authentication, because it is an application-specific server-side mechanism, (iii) the Web Payment APIs, since a sensitive usage of this API would require a detailed modeling of the server-side behavior of payment providers and merchants servers. On the other hand, WebSpec supports several client-side mechanisms and security policies, like domain relaxation, CSP and CORS, that are not part of WIM. Additionally, being a pen-and-paper model, WIM can neither be used to automatically discover security vulnerabilities, nor to develop automated or computer-assisted proofs, which are central features of our framework.

**Other works.** Quark [35] is a WebKit-based Web browser, whose kernel has been implemented and formally verified in Coq. The kernel is responsible for handling input/output and offers services to the other possibly compromised components of the browser, which deal with various operations such as the rendering of Web pages, handling of cookies and tab management. The separation of duties between the kernel and browser’s components, together with a set of security policies implemented in the kernel, enables Quark developers to formally prove the enforcement of security properties, such as tab non-interference, and integrity of cookies and responses. This is orthogonal to our work, which instead aims at devising a formal browser model to validate Web invariants.

Automated testing is a popular methodology employed in software development processes for bug detection. An application of this methodology in the context of Web security is BrowserAudit [33], a framework composed of over 400 automated tests, which can be used to verify the correctness of the implementation of Web security mechanisms in existing browsers. However, BrowserAudit cannot be used to spot bugs at the specification level, which is the goal of our work.

QuickChick [45], 49 is a framework for property-based testing written in Coq. It combines formal methods and testing to formally verify that the code of a test generated from a given property is indeed checking its correctness. QuickChick is orthogonal to our work since it focuses on test case generation: WebSpec relies on testing solely to prove the validity of the counterexamples to our invariants produced by the Z3 solver.

**Summary.** To conclude, WebSpec is the first framework that mechanizes formal proofs and counterexample-finding for Web invariants. In addition, our browser model is the most comprehensive one when it comes to browser-side security mechanisms, as we further detail in Appendix E.

**VIII. Conclusion**

In this paper we presented WebSpec, the first formal framework for the security analysis of Web platform components that supports the automated detection of logical flaws and allows for the development of machine-checked security proofs. We showcased the effectiveness of WebSpec by discovering novel attacks and inconsistencies affecting current Web standards, and automatically validated our findings against major browsers. Additionally, we discussed how WebSpec can be used to carry out machine-checked security proofs for vulnerability fixes. As a future work, besides expanding the model to cover more Web platform components, we plan to define additional Web invariants by reviewing newly proposed mechanisms and engaging with the community, including developers and editors of Web standards.
ACKNOWLEDGMENTS

This work has been partially supported by the European Research Council (ERC) under the European Union’s Horizon 2020 research (grant agreement 771527-BROWSER); by the Austrian Research Promotion Agency (FFG) through COMET K1 SBA, and by the Vienna Business Agency through the project Vienna Cybersecurity and Privacy Research Cluster (ViSP).

REFERENCES

APPENDIX

APPENDIX A

WEB INVARIANTS

A. Interactions with the SOP

With the script-src CSP directive, developers can specify which scripts can be included in a page and thus access the DOM. This corresponds to the following property.

Invariant I.4. The DOM of a page protected by CSP can be read/modiﬁed only by the scripts allowed by the policy.

We encode the invariant in our model as follows:

We encode the invariant in our model as follows:

1 Definition CSPInvariant (gb: Global) (evs: list Event) (st: State): Prop :=
2 ∀ pt sc ctx pt_u src origin tctx tt _evs,
3 Reachable gb evs st →
4 (forall script sc is present in the page *)
5 is_script_in_dom_path gb (st_window st) pt sc ctx →
6 (∀ The DOM of the top-level window was modiﬁed by sc *)
7 evs = (EvScriptUpdateHTML pt (DOMPath [] pt_u) tctx := _evs)
8 (∀ The toplevel window is protected by CSP *)
9 rp_hd_csp (dc_headers (wd_document (st_window st))) = Some
10 { script-src-sc := Some src; csp_trusted_types := tt ||} →
11 (∀ The script-sc is allowed by the CSP *)
12 origin of url (wd_location (st_window st)) = Some origin →
13 script-src-match src origin (script-src sc).

Where the script-src-match predicate holds when the src source expression matches the URL script-src sc in a page loaded from origin origin.

Attack. By running the query, our toolchain produces a counterexample that corresponds to the CSP violation discovered by the authors of [49]. The complete trace is shown in Figure 7. We see in the invariant I.3.1 where the iframe loads a script from origin 2 which alters the DOM of the parent page, violating the Web invariant [49]. Notice that this interaction is allowed by the SOP since both pages share the same origin. This violation can be prevented by disabling domain relaxation. The Coq proof of the correctness of this solution is available online [7].

B. Integrity of server-provided policies

Given the privileged position of service workers, we must ensure that the integrity of security policies attached to server responses is preserved, and such policies are correctly enforced by the browser. This corresponds to the following Web invariant.

Invariant I.5. If a response from the server contains a security policy, then the browser enforces that speciﬁc policy.

We encode the invariant in our model as follows:

APPENDIX B

COMPILER

WebSpec includes a compiler that aims to ﬁnd inhabitants of inductive types, a problem which is known to be undecidable for
CIC, the logic of Coq [27]. To this end, the compiler translates terms in a fragment of CIC into CHC logic, i.e., first-order logic with fixed-points expressed in terms of Constrained Horn Clauses, hence discharging the undecidability of the problem to CHC solvers [32], [31]. In the following, we give an overview of how our compiler performs this translation.

A. Considered CIC Fragment

Contrary to related work [23], [22], our compiler does not perform a shallow embedding into untyped first-order logic, but instead performs a type-preserving translation into CHC logic, i.e., typed first-order logic with fixed-point. If, on the one hand, this allows us to leverage all the power of CHC solvers, this comes, on the other hand, at the price of restrictions on the fragment of the logic of Coq we consider.

The considered fragment of the logic of Coq we consider is CIC without dependent types (1), and where inductive type annotations and constructor arguments are restricted to ground variables (2). We also require inductive type parameters to be instantiated when the compiler is called. Before discussing these limitations, note that the resulting logic is still extremely expressive as it contains System $F_{\omega}$, the higher-order polymorphic lambda calculus. This also means that the inhabitation problem is still undecidable on this fragment [27].

The reason for the restriction on inductive type annotations and constructor arguments (2) is twofold. The first reason is that CHC solvers do not perform type equation resolution, and therefore introducing symbolic type variables is forbidden. It is possible to circumvent this issue by performing a shallow embedding of types, however this would likely come at a significant cost in resolution time. The second reason is similar to the first, but for functions. However in this case, we expect this restriction to be relaxed in the future thanks to recent progress in function synthesis [12], [46].

The restriction on dependent types (1) could also be circumvented by shallow embedding, but again at a high cost in resolution time. Instead, upcoming development of WebSpec aims to relax this restriction so that dependent types are allowed in inductive types. This relaxation will cover a significant number of practical cases, like the famous example where the type of an array includes a program expression giving the size of that array.

B. Compilation Pipeline

In order to translate the support fragment of CIC to CHC logic, our compiler performs the following steps:

**Term-Type-Kind Hierarchy** From a syntactic point of view, types cannot be distinguished from terms in CIC. Because CHC does not permit such intricacy, we have to built a strict term-type-kind stratified hierarchy [13], where kinds are defined as $k := \text{Prop} \mid \text{Set} \mid k \rightarrow k$. This stratification is done by recursive exploration, starting from the inductive type on which the compiler is called, and following CIC typing rules to deduce to which stratum each syntactic term belongs. We rely on Coq type-checking to ensure that connections between terms, types and kinds are sound. As a side effect, this stratification makes a clear distinction between types and proposition or between terms and proofs, which will ease subsequent steps.

**Partial Application** In CIC, any term can be partially applied. This includes functions of course, but also inductive types, constructors, or type definitions. Such flexibility is not allowed in CHC, and therefore all partial applications have to be removed. This is done by systematically performing $\eta$-expansion on every term that could be applied.

**Lambda Abstraction** We also have to removed lambda abstractions, both those which are present in the original CIC terms and those which were introduced by $\eta$-expansion. To this end, we perform $\beta$-reduction wherever possible and remove remaining lambda-abstractions by lambda-lifting [36], [41].

**Polymorphism and Higher-Order** Thanks to the previous steps, all functions are now defined at top-level and totally applied. Therefore we can now remove the use of polymorphism and higher-order simply by specialization: For every application of a function (resp. an inductive type) to a type or a function argument, we generate a specialized version of the function (resp. the inductive type) where the type or function parameter is replaced by the argument.

**Constructor Constraints** Constructors of inductive types in CIC can contain terms with arbitrary constraints, while CHC only supports simple algebraic datatypes. Therefore, we split every non-simple inductive type into a simple inductive type of kind Set and an inductive type of kind Prop which encapsulates these constraints.

Once these steps are done, the rest of the compilation is straightforward. Simple inductive types of kind Set are mapped to CHC algebraic datatypes, inductive types of kind Prop are mapped to relations, while CIC terms, types and proposition are mapped to CHC terms, sorts, and formulas.
**TABLE III**

<table>
<thead>
<tr>
<th>Features</th>
<th>Solving Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>71m</td>
</tr>
<tr>
<td>cookie</td>
<td>13x</td>
</tr>
<tr>
<td>redir</td>
<td>3h 34m</td>
</tr>
<tr>
<td>cors</td>
<td>58s</td>
</tr>
<tr>
<td>lsc</td>
<td>3h 25m</td>
</tr>
<tr>
<td>ref</td>
<td>1m 7s</td>
</tr>
<tr>
<td>pmsg</td>
<td>2h 14m</td>
</tr>
<tr>
<td>lst</td>
<td>48s</td>
</tr>
<tr>
<td></td>
<td>6h 54m</td>
</tr>
<tr>
<td></td>
<td>2m 11s</td>
</tr>
<tr>
<td></td>
<td>9h 32m</td>
</tr>
<tr>
<td></td>
<td>1m 52s</td>
</tr>
<tr>
<td></td>
<td>13h 20m</td>
</tr>
<tr>
<td></td>
<td>2m 7s</td>
</tr>
<tr>
<td></td>
<td>15h 46m</td>
</tr>
<tr>
<td></td>
<td>3m</td>
</tr>
</tbody>
</table>

base: Core Browser Functionality, CSP and Service Workers; cookie: Cookies; redir: HTTP Redirections; cors: CORS Protocol and Headers; lsc: Local Schemes; ref: Referer and Referrer Policy; pmsg: Post Message; lst: Local Storage

### Appendix C

**Scalability**

In this section, we report on the result of the experimental evaluation of the scalability of our browser model. In particular, we measure how the addition of individual Web components affects the performance of WebSpec in finding counterexample pipelines, and show how lemmas (Section VI-A) are the most effective tool for improving the solving time. We focus, as our main case study, on the Integrity of server-provided policies Web invariant (Appendix B), since the counterexample found by WebSpec requires the browser model to only support service workers, the cache API, and the CSP header, outside of the core browser features (e.g., requests, responses, DOM, etc).

Starting from the features listed in Table IV, we identify 10 modules, each representing a Web platform feature and refactor our model to be configurable w.r.t. the included components. With this modification our model is composed of a (core) core set of browser functionality on top of which we are able to automatically include or exclude (cookies) the Cookie and Set-Cookie headers, the cookie jar, and the document.cookie JavaScript API; (redir) HTTP redirections and response codes; (cors) the CORS protocol and its request and response headers; (csp) the Content-Security-Policy headers and rules, including the script-src and trusted-types directives; (sw) service Workers and the JavaScript Cache API; (lsc) support for local scheme URLs, the URL.createObjectURL API, and the inheritance rules for CSP; (ref) the Referer request header and the referer policy mechanism; (pmsg) the Web messaging API (window.postMessage); (lst) the local storage API (window.localStorage).

Table VI reports the time required by WebSpec to find the counterexample for our case study on 8 incrementally more complex versions of the browser model. For each version, we include one additional features and run the query twice, measuring the running time with or without the inclusion of lemmas. The table clearly confirm the intuition that the solving time increases with the addition of new features, as the base model requires less than one hour, compared to the 16 hours of the complete model. This increase however is not predictable, as the addition of a feature may benefit the solver by introducing constraints which limit the search space, as for example in line #4, where the addition of CORS improves the solving time by one third. In all versions of the model, lemmas offer a substantial improvement of the counterexample finding pipeline, never exceeding the 3 minutes of solving time. In our case study, the lemma which is first applied by the solver is script_state (Section VI-A), which comprises the first 5 events of the 9 present in the complete trace. This shows that limiting the number of steps the solver needs to consider brings a more noticeable improvement than the simplification obtained by the removal of Web components.

### Appendix D

**Verifier Example**

In Figure 9 we provide an overview of how the attack trace shown in Figure 5 is translated into a WPT test, executable on real browsers. Each event (on the left) is assigned a SAV tuple. Notice that we omit the Setup element of the tuples since none of the events in the trace requires it.

Each WPT test generated by the verifier has at least two components: the test launcher and the server. The test launcher is the main WPT test file and consists of an HTML page containing a script that starts the actual test. The server component of a test is a Python script that handles requests (for all origins) and their corresponding responses. For simplicity, we omit the test launcher from the DOM column of Figure 9.

The first event in the trace is an EvRequest to the URL 01/A, which is assigned an explicit action 1 and an explicit verification 2. These are translated, respectively, to a window.open call, performed by the test launcher, and a server-side assertion to ensure that the request was performed. The window.open call creates a new browser window, represented in our example by the first element in the DOM column. The second event (EvResponse) is mapped to the explicit action 3 of responding to the request, and an implicit verification. The verification of this action is implicit, since the response is known before runtime and generated by the test server. The third event in the trace (EvDOMUpdate) is mapped to an implicit action and an implicit verification, since updating the DOM after an HTTP response is internal browser behavior that cannot be explicitly tested. An incorrect DOM update, however, would cause all subsequent assertions to fail. Since the HTTP response from action 3 returns an HTML page containing an iframe (iframe1) with source 01/B, the browser is expected to load the subresource from this URL via a GET request. This request corresponds to event 4 (EvRequest) whose action is thus implicit. However, the verification of this action is necessary and explicitly done through an assertion at the server-side. The next two events (5 EvResponse and 6 EvDOMUpdate) are similar to events 2 and 3, differing only in the content of the HTTP response, this time containing an HTML page that includes a script with source 02/C.

The browser is then expected to fetch the script from 02/C (events 7, 8, and 9). At this stage, the script is loaded
in iframe and ready to execute. Script events are always mapped to explicit actions and explicit verifications. In this example, event 10 (EvScriptUpdateHTML) is translated to JavaScript code that tries to modify (via innerHTML) a target element in the parent page, and performs a WPT assertion on the content of the target element. Since the innerHTML sink in the parent page requires a valid Trusted Type, the script first creates a loose policy and then updates the target element with a TrustedHTML created with such policy. Note that the JavaScript code that is executed for event 10 is received as the response body of event 8. Since WebSpec does not model JavaScript code but only the API calls that scripts can perform (Section II-E), the verifier generates the response body (i.e., the JavaScript code) and then the target element with a TrustedHTML created with such policy. Note that the JavaScript code that is executed for event 10 is received as the response body of event 8. Since WebSpec does not model JavaScript code but only the API calls that scripts can perform (Section II-E), the verifier generates the response body (i.e., the code) for each script depending on the actions and verifications that the script needs to perform in the subsequent events.

If the test is successful, i.e., all assertions pass, the script in iframe is able to modify an element in the parent page that is protected by Trusted Types, thus confirming the presence of the attack on real browsers.

APPENDIX E
COMPLETENESS

Table IV provides an overview of the features supported by the models discussed in Section VII. In particular, here we focus exclusively on Web components implemented in Web browsers, since this is ultimately the goal of our work, but models like WIM [26], WebSpi [11] and Alloy [9] additionally implement a variety of other features that are needed to model other parts of the Web ecosystem.

One of the main differences among the different proposals lies in the way JavaScript is modeled. WIM and Featherweight Firefox model the small-step semantics of (a subset of) JavaScript: this is of fundamental importance, e.g., in WIM, since the model has been used to verify the security of Web protocols and it is necessary to define the precise semantics of scripts used by the parties involved in the protocol run. In WebSpec, similarly to the Alloy model of Akhawe et al. [9], we are only interested to the API calls that a script can perform. For this reason, we do not specify the exact behavior of a script, rather we assume that a script can call the supported APIs in any arbitrary way, using any data in its knowledge as parameters to these calls.

From the perspective of features support, WebSpec and WIM are the two most complete models available so far. As mentioned in Section VII, WebSpec does not support HSTS, HTTP basic authentication and the Web Payment API, since we abstract away from the network and from the specific implementation of Web servers. On the other hand, we support a variety of features that are missing in WIM browsers, such as CORS, cookie attributes like Domain, Path and SameSite, the __Host- prefix, CSP, the Cache API, interception of request of service workers, the document.cookie API, which play a prominent role for many of the attacks reported in this paper. As shown in the table, there are also some minor differences concerning modeled URL components, type of supported HTTP redirects, status codes and headers, and functionality of service workers.

Concerning the other models, WebSpec essentially supports all the features implemented by them. Currently we only support the Origin and Access-Control-Allow-Origin HTTP headers for CORS, while the Alloy model supports all of them, although the headers supported in WebSpec are sufficient to implement the fundamental CORS functionalities, the remaining ones allow a more careful treatment of CORS and we plan to implement them as future work.

For space reasons, in Table IV we let ACA stand for Access-Control-Allow, AC for Access-Control and ACR for Access-Control-Request.
# TABLE IV
## COMPARISON OF SUPPORTED WEB COMPONENTS IN EXISTING MODELS

<table>
<thead>
<tr>
<th>Web Components</th>
<th>WebSpec</th>
<th>WIM</th>
<th>W3C</th>
<th>Alley</th>
<th>FF</th>
<th>MIE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scheme</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTTP(S)</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pseudo-protocols</td>
<td>data, blob:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Host</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Port</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Path</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fragment</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JS API URL.createObjectURL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Request methods</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GET</td>
<td>✔️</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POST</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Others</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUT, DELETE, OPTIONS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUT, DELETE, OPTIONS, TRACE, CONNECT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUT, DELETE, OPTIONS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Response codes</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Redirection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>302, 307</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>303, 307</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>302</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>303, 307, 307</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>302</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200, 401</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>HTTP</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>HTTP headers</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cookie</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set-Cookie</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attributes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Domain</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Path</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Cookies</strong></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secure</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PttOnly</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SameSite</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__Secure *, Prefix</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__Host- Prefix</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JS API document.cookie</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SOP for cookies</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Windows</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple tabs</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Framing support</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cross-window communication (postmessage API)</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JS API window.location</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JS API window.history</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JS API window.close</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DOM</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JS API for DOM manipulation</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JS API document.domain</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SOP for DOM access</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>XMLHttpRequest / Fetch API</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SOP for XHR / fetch requests</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sending requests via Javascript</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reading responses via Javascript</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Forbidden response headers (Set-Cookie)</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CORS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Request types</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple requests</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-simple requests (w. preflight)</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>HTTP headers</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Origin</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Access-Control-Allow-Origin</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Others</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CSP directives</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>script-src</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>trusteds-types</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>require-trusted-types-for</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CSP Inheritance</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trusted types</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create trusted types (policy.createHTMl)</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secure context restriction</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Service Workers</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interception of requests (event.respondWith)</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accessing the cache API</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Messaging with other windows</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opening new windows (windowOpen)</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Storage APIs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache API</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache.put</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache.match</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secure context restriction</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local storage</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Web Payment APIs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>