# RETROCSP: Retrofitting Universal Browser-Support for CSP

Moritz Wilhelm, Sebastian Roth, and Ben Stock
CISPA Helmholtz Center for Information Security
{moritz.wilhelm, sebastian.roth, stock}@cispa.de

***Abstract*—To protect their users from the threats caused by successful Cross-Site Scripting (XSS) attacks, site operators can rely on a Content Security Policy (CSP) as a defense-in-depth mechanism. CSP, when properly deployed, severely limits the attacker's capability to execute malicious code, thereby mitigating the effect of an XSS flaw. However, several studies have shown that *secure* CSP adoption lacks in the wild, and a recent study uncovered the lack of uniform browser support as a major roadblock for CSP. In this paper, we propose, implement, and evaluate RETROCSP, a solution which allows developers to provide consistent levels of protections to their users. Through a Service Worker, we automatically rewrite the page's code and apply retrofitting purely in JavaScript (JS) userland. Our solution has a small overhead of around 20ms and not only retrofits new CSP features but also addresses browser-specific bugs. Further, our findings led to several fixes in major browsers and the CSP specification itself.**

## I. Introduction

The Web is one of the most important application platforms of our daily lives. We all use it regularly to conduct online business, stay in touch with friends and family, or to get in contact with government agencies. One of the core security policies of the Web is the so-called Same-Origin Policy [9]. It isolates Web content from each other and allows access if the origin of documents is the same, i.e., they share a protocol/scheme, the host/hostname, and port. One of the most dangerous attacks against Web applications is Cross-Site Scripting (XSS). Here, the attacker abuses a flaw in server- or client-side code [26] to inject their own code into the vulnerable application. Hence, this code now runs in the origin of the flawed application, giving the attacker the ability to execute arbitrary actions in the name of the user.

To limit the damage that can be caused by such injections, the W3C has standardized the Content Security Policy (CSP). This is a server-sent, but client-enforced mechanism (usually delivered via an HTTP response header), which allows to limit the sources from which resources are loaded. With respect to mitigating XSS, this allows the developer of a site to limit the sources for scripts and disable inline scripts. Hence, while it does not fix the underlying XSS issue, CSP serves as a defense-in-depth mechanism that mitigates the problem. If the policy is well-chosen, it limits the attacker's ability to inject arbitrary script code or, in the best case, even ensures that attackers cannot execute any code at all.

The security which can be provided by CSP is governed by two aspects: the choice of a secure server-sent CSP and the proper implementation of the CSP mechanism within browsers. Prior work has repeatedly shown that the former step fails in practice [22, 11, 30]. Recently, Roth et al. [23] conducted an interview study with twelve developers to understand the roadblocks of CSP. Next to technical roadblocks, such as being blocked from sane CSPs through third-party code [27], a repeated theme was the lack of uniform support of the full CSP standard. These issues ranged from the well-known problem of lacking support of the `'strict-dynamic'` source expression to incorrect handling of how WebSockets are controlled. This means that even if developers are willing to take all necessary steps to work out a safe CSP, they are left without the ability to enforce it, as it might break functionality on some clients.

To ease the burden of having to have non-uniform behavior in different browsers, in this paper, we propose RETROCSP, a JavaScript-based solution that addresses inconsistent behavior in browsers. Therefore, we allow a developer to curate a CSP for full compliance, and RETROCSP automatically retrofits the intended functionality and security for browsers without full support. To that end, we first identify the set of non-uniformly supported directives and expressions and decide which can be retrofitted. We then rely on a Service Worker which automatically rewrites a page's code to retrofit support for the mechanisms. In doing so, we also conduct a thorough analysis of the standard compliance of modern browsers with respect to CSP. Our findings not only allow us to cover violations in a safe way, but we also report the issues to affected vendors. Further, we even detected underspecification of the CSP standard, which has since been rectified.

To sum up, our paper makes the following contributions:

- Through a thorough set of test cases, we uncover non-uniform support for several critical CSP features and bugs in their implementation. Further, our tests highlighted an underspecified case in the CSP standard.
- We design, implement, and evaluate RETROCSP, a purely client-side, JavaScript-based, easily-extendable framework for retrofitting CSP features to non-supporting browsers.
- Our evaluation shows that RETROCSP is not only able to retrofit lacking features but also to fix bugs in existing implementations. The overhead in our tests amounted to just 20ms, making it feasible to deploy in practice.
- To facilitate future work, reproduction of our results, and allow for immediate roll-out to real-world Web sites, we make our prototype available [1].

## II. Background & Related Work

In this section, we introduce the two core technical concepts, CSP and Service Workers, and surveys related work.

### A. Content Security Policy

CSP was originally proposed in Stamm et al. [25] in 2010. Its initial purpose was to mitigate XSS attacks. To that end, it features a set of directives that control resource inclusion. For example, the ability of a page to include images or scripts is governed by `img-src` and `script-src`, respectively. CSP also features a fallback directive, on which all of the so-called fetch directives fall back to: `default-src`. If for any type of resource there is no explicit directive, `default-src` is used instead. Otherwise, the most specific directive is applied.

By default, if a policy contains a `script-src` (or `default-src` fallback), the page cannot rely on inline scripts, since these are disallowed by default. To re-enable these, for the initial version of CSP, developers had to add the `'unsafe-inline'` keyword. However, this also undermines any protective capability of the CSP since an attacker can simply inject an inline script of their choosing. The trouble this creates in the wild was first documented by Weissbacher et al. [31], who showed that CSP had little to no adoption in the wild. To alleviate the issue, CSP level 2 introduced the concept of hashes and nonces. Here, the developer can use the CSP to specifically allow scripts by their cryptographic hash or to attach a number used only once (nonce) to their policy. Then, the developer can attach the same nonce to all of *their* scripts. Under the assumption that the nonce is random and used exactly once, an attacker cannot guess it, i.e., their injected code cannot be executed. Moreover, if nonces or hashes are present, the `'unsafe-inline'` keyword is ignored by supporting browsers. This design is to allow CSPs to be backwards compatible, i.e., a developer might secure their site with nonces, but leaves `'unsafe-inline'` in the policy to not break the page for legacy clients. Notably, while inline scripts can be allowed through nonces or hashes, event handlers *cannot*. To ease developers' burdens, CSP Level 3 specifies the `'unsafe-hashes'` keyword. Combining this with the hash of an inlined event handler, developers can enable the usage of inline events. Note that the *unsafe* part of the name refers to the fact that an attacker might reuse any script allowed through its hash for their purposes. Further, since event handlers are bound to an element, an attacker may also inject a crafted element to abuse an event handler that accesses any of the properties of the elements (e.g., `eval(this.name)`.

While the overall adoption of CSP picked up over the years [30, 11, 22], the vast majority of policies were still insecure. Across all of these works, 90-95% of observed policies were trivially insecure. As discussed by Weichselbaum et al. [30], this in part stems from the nature of the Web: its reliance on third parties [27, 21, 28]. In particular, if a site includes, e.g., advertisements, these scripts add additional content from other sources. Hence, it is not sufficient to allow just the ad's domain, but in theory, a developer would need a priori knowledge about any potential additional sources of scripting content. In practice, to avoid breakage of the all-important and revenue-generating ads, sites therefore often need to allow sources such as `https://*`. Of course, this again undermines the security of a CSP, since now the attacker can inject scripts coming from arbitrary HTTPS URLs. To address this issue, Weichselbaum et al. [30] proposed an extension of the `script-src` directive: `'strict-dynamic'`. With this, any scripts that are explicitly trusted (through a nonce or hash) can programmatically add additional scripts. These scripts are also flagged as trusted and can themselves include additional content. In addition, the presence of `'strict-dynamic'` means that supporting browsers will disregard any hosts that are present. This also serves for backward compatibility, since non-supporting browsers can rely on the list of allowed sources.

This keyword, along with several other additions such as the `'unsafe-hashes'` keyword, have since become part of the CSP Level 3 Living Standard. Given its shifting nature, the standard is not fully implemented by all browsers. Since much of the CSP development is driven by Google, Chrome (and its derivatives) support most features. However, other browsers lack behind in adoption of the features or implement them inconsistently with respect to the standard.

In general, while a CSP can protect the page from including resources from unauthorized sources, it cannot disable navigation. Hence, while exfiltrating information through the inclusion of an image pointing to an attacker's site is impossible, the attacker can simply redirect their victim's browser. That is, the attacker runs `location.href='//attacker.com/?leak=' + data`. To stop such data leakage from occurring, CSP Level 3 introduces the `navigate-to` directive. This is meant to control where a navigation can occur to (through clicking links or from JavaScript). Unless the target of a navigation is in the list, the browser will refuse to navigate away. The mechanism can be relaxed through the `'unsafe-allow-redirects'` keyword. This means that the browser will make the initial request irrespective of its target and only navigate back if the final destination is not in the allowed list. However, this implies a number of implementational challenges; e.g., any headers (such as HSTS) or cookies which were sent as part of a disallowed redirection chain must be deleted. Hence, browser support is still lacking altogether.

### B. Service Workers

With the increasing reliance of Web applications for applications such as Office (e.g., through Google Docs), the need for offline support rose. To accommodate this need, browsers nowadays support *Service Workers* [24]. A *Service Worker* is a script that runs on a *separate* thread in the browser. They are isolated from the main thread and run in the background [18]. However, they can react to specific functional events. Use cases of Service Workers include push notifications, background sync [8] as well as caching. Most

| **Feature** | **Chrome** | **Edge** | **Opera** | **Firefox** | **Safari** |
|---|---|---|---|---|---|
| | 88.0.4324 | 88.0.705 | 74.0.3911 | 85.0.2 | 14.0.3 |
| manifest-src | ✓ | ✓ | ✓ | ✓ | ✗ |
| plugin-types | ✓ | ✓ | ✓ | ✗ | ✓ |
| script-src-attr | ✓ | ✓ | ✓ | ✗ | ✗ |
| script-src-elem | ✓ | ✓ | ✓ | ✗ | ✗ |
| strict-dynamic | ✓ | ✓ | ✓ | ✓ | ✗ |
| unsafe-hashes | ✓ | ✓ | ✓ | ✗ | ✗ |
| style-src-attr | ✓ | ✓ | ✓ | ✗ | ✗ |
| style-src-elem | ✓ | ✓ | ✓ | ✗ | ✗ |
| navigate-to | ✗ | ✗ | ✗ | ✗ | ✗ |
| report-to | ✓ | ✓ | ✗ | ✗ | ✗ |
| prefetch-src | ✗ | ✗ | ✗ | ✗ | ✗ |
| trusted-types | ✓ | ✓ | ✓ | ✗ | ✗ |
| worker-src | ✓ | ✓ | ✓ | ✓ | ✗ |

***TABLE I:*** *Browser compatibility with selected CSP features [15, 14]*

importantly, Service Workers can work as a *client-side proxy*. By fetching resources, accessing the responses and modifying them completely, they can effectively overwrite the default browser behavior.

Service Workers have a separate life-cycle from the page itself. First of all, a Service Worker has to be fetched and *registered* in order to be deployed on a Web site. This is achieved using the `ServiceWorkerContainer.register()` method. If the registration is successful, the Service Worker is bound to a *scope*. Said scope defines which URLs a Service Worker controls. The scope of a Service Worker is directly inherited by its own location. A Service Worker can *at most* control its own origin.

Whenever a client initiates a request, the Service Worker can react to the triggered `fetch` event and handle the request accordingly. The Service Worker can hijack the request by installing a `fetch` event handler. This handler is able to overwrite the default browser behavior by passing a `Response` object [36] to the `event.respondWith()` method. The manually crafted response will be served to the client instead of automatically fetching a resource from the server.

At the time of this paper, the `ServiceWorker` Application Programming Interface (API) is supported in at least Google Chrome, Mozilla Firefox, Opera, Safari, and Microsoft Edge [7] although it is still an Editor's Draft [24]. Thus, it can be used to implement a client-side proxy that runs in parallel to the main thread in these particular browsers and allows to rewrite responses for URLs under its control.

## III. Assessing CSP Compliance

To first understand the exact nature of potential inconsistent implementations, we built a comprehensive suite of test cases based on the CSP standard in the form of HTML pages. Since the main goal of our work is to provide a framework to retrofit CSP features to browsers that lack support for those, we chose features for closer analysis that fulfilled the following three characteristics at the time of writing:

1) The feature must lack wide-spread *browser support*. It makes only sense to retrofit features that are not supported by all modern browsers (see Table I).
2) In order to only focus on the most relevant directives, the feature has to have impact on the XSS mitigation capabilities of CSP, ease the deployment of a non-trivially bypassable CSP, or must limit the attackers capability to extract sensitive data in case of a successful XSS attack.
3) Lastly, it should be possible to implement the feature using client-side JavaScript, such that can extend browser functionality without the need to install additional software on the client side.

### *A. Standard Violations*

Our analysis indicates that, albeit browser support for CSP is widespread, there are even inconsistencies in features that are implemented. During the functionality evaluation, we encountered several standard violations among all browsers. In this section, we present each of them and demonstrate what kind of actions RETROCSP performs to eliminate browser inconsistencies.

*1) Incorrect implementations of 'strict-dynamic':* According to the specification, apart from enabling trust propagation, `'strict-dynamic'` deactivates any scheme and host source expression as well as the `'self'` keyword. Notably, this should have effect on scripting content, not other resources. All chromium-based browsers, however, enforce this mechanism too harshly if `'strict-dynamic'` is present in a `default-src` directive. They deactivate host-based allowlisting for *all* resources guarded by `default-src`. In particular, this means that merging an `img-src` and `script-src` directive into one functional equivalent `default-src` directive is not possible in these browsers.

*Google Chrome*, *Microsoft Edge* and *Opera* prevent loading the image in Fig. 7 although it is explicitly trusted by the CSP. We documented and reported [4] the standard violation to the *Chromium Project*. The behavior is already fixed and was shipped with version 89.0.4351.0.

*Mozilla Firefox* is not affected by this explicit standard violation; however this due to another standard violation (Section III-A2). It does not block the image from being loaded. *Safari* does also not enforce this policy as intended due to the missing `'strict-dynamic'` support. However, a general lack of implementation is not a standard *violation* by our definition.

A similar issue exists in Chromium derivatives when the `'strict-dynamic'` keyword is used outside of `script-src-elem`, `script-src`, or `default-src`. We also reported this incorrect handling of the keyword [5] and it was fixed along with the previously discussed bug. Note that Firefox is not affected by the bug, and neither is Safari (for its general lack of `'strict-dynamic'` support).

*2) Lacking support for nonces, hashes, and 'strict-dynamic' in default-src:* Although *Firefox* is not affected by the previous two discovered standard violations, it still has its own flaws. We figured out that the *Firefox* browser does not support hashes, nonces or `'strict-dynamic'`[1] within a

[1]The console states that it is ignored, but it cannot be used without nonces and hashes anyways.

`default-src` directive. Neither CSP Level 2 nor Level 3 are fully enforced in this directive.

In *Firefox*, neither the first nor the second inline script of Fig. 8 is executed. The CSP is interpreted and enforced as `default-src 'none'`. However, `default-src` still works as the fallback mechanism for scripting resources if `script-src` is not present. This behaviour is only exhibited by Firefox, and all others implement hashes and nonces correctly when used in `default-src`. This bug was already reported four years ago, but we have since updated it and provided additional test cases for validation [2].

*3) 'unsafe-hashes' as default behavior:* As noted in Section II-A, `'unsafe-hashes'` can be used to allow inline event handlers by their hash. Notably, Firefox does not support this directive (yet). However, by default, *Firefox* enforces CSP as if `'unsafe-hashes'` was set, i.e., any script allowed through its hash is allowed to be as an event handler for any element. CSP specifically excludes hashes as a source for event handlers. This design choice is made for two reasons: first, if developers hash their inline event handlers, an attacker can use these hashed scripts as regular inline scripts, e.g., triggering some behavior automatically, which would have been triggered only on a click event. Second, if an event handlers uses the element to which it is bound (e.g., evaling `element.data`), an attacker can re-use the allowed script with an element of their own choosing. Hence, defaulting to this insecure behavior needlessly increases the attack surface of a potential XSS vulnerability in Firefox. We also reported this bug to Mozilla [3].

*4) Hash value of javascript: URLs:* Browsers also behaved inconsistently when deciding whether a `javascript:` URL navigation should be allowed. The chromium-based browsers required the CSP to contain the hash of the entire URL. On the other hand, *Firefox* only allowed code execution if the CSP contained the hash value of the URL without its scheme.

For example, given the URL `javascript:alert(42)`, *Firefox* requires the hash source expression to contain `SHA256('alert(42)')`. In contrast, all of the Chromium-based browsers require the full string, i.e., `SHA256('javascript:alert(42)')`. The first approach makes it easy to re-use trusted code for inline scripts or `javascript:` navigation as their hash value is identical. For the second one, this is not the case. However, it has an advantage from a security point of view. The hash is now context-aware. Depending on *where* the code is specified, its hash value differs.

The CSP standard makes no statement about the correct hash value computation of `javascript:` URL, and there are arguments for both approaches. However, the inconsistent behavior makes it needlessly harder for developers to build a functional CSP.

*5) Non-Punycode-encoded host source expressions:* A CSP only allows host source expressions to consist of *ASCII* characters. According to the specification, any URL containing non-ASCII characters has to be Punycode-encoded [12]. When a directive's source list contains a host source expression with an internationalized domain that is *not* Punycode-encoded, browsers behave inconsistently. To evaluate this, we rely on test case shown in Fig. 9.

*Safari* and the three chromium-based browsers warn the client using a console message that the CSP contains an invalid character and suggest to percent-encode [10] it. However, rather than ignoring only the incorrect source expression, these browsers ignore the entire directive. Hence, the inline script will open the alert box, even though the `script-src` contains no valid entries, i.e., should block any script execution. *Firefox* on the other hand, simply ignores the invalid source expression.

We also reported this browser inconsistency and the false warning message to the affected browser vendors [6]. Whereas the warning message has already been adjusted, the underlying behavior was not changed at the time of this writing.

## IV. RETROCSP

In the following, we outline the design of RETROCSP, explain its core component, i.e., the Organizer, and present how our retrofitters are implemented.

### A. Design

The following section provides an overview of our retrofitting architecture.

Depending on the retrofitted feature, we need the ability to block requests while ensuring that we do not falsely block any request. For this reason, we constructed the architecture on the *client side*, to be able to block or change requests before the browser sends them. Furthermore, the architecture has to be able to *read and modify* the Content Security Policy (CSP) sent by a Web service. While it is no problem to access CSPs that are set via Hypertext Markup Language (HTML) `meta` tags, it is impossible to access the HTTP headers of a response in the context of a document.

In order to fulfill both requirements, we decided to use a *Service Worker* [24] as a base of the retrofitting architecture because they are able to act as a client-side proxy. Which can intercept, change, and block requests and responses. Since Service Workers are supported by all modern browsers [7], we also ensure that RETROCSP works for the majority of all clients.

Fig. 1 illustrates the general structure of RETROCSP. The tool itself consists of an *Organizer* and multiple retrofitting modules, called *retrofitters*. The main Service Worker, as an organization component, acts as Man-In-The-Middle proxy, in order to read all incoming CSP headers and enable the corresponding retrofitters. Each retrofitter itself analyzes the CSP and performs its associated retrofitting actions on the response. Those retrofitting actions include:

1) the modification, addition, and deletion of directives and source-expressions of the received CSP.
2) hooking JavaScript functions and APIs in order to implement/retrofit specific CSP access control checks.
3) reusing universally supported CSP features to simulate those that are not implemented in all modern browsers.

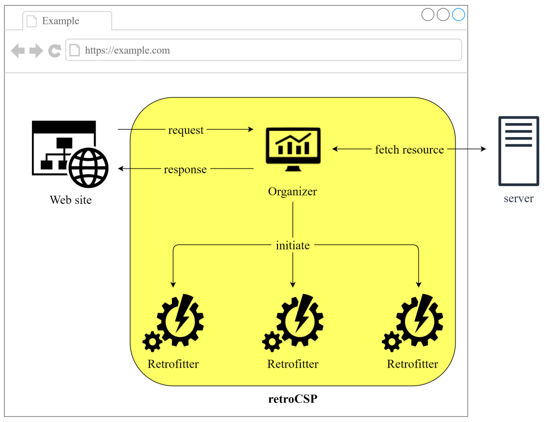

***Fig. 1:*** *General structure of* RETROCSP

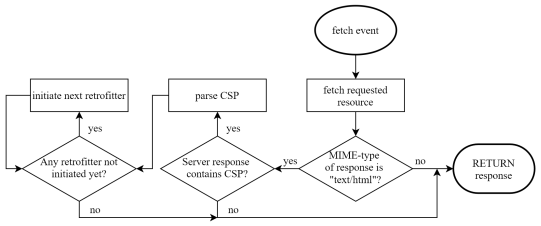
fetch event
fetch requested resource
parse CSP
initiate next retrofitter
yes
yes
Any retrofitter not initiated yet?
Server response contains CSP?
yes
MIME-type of response is "text/html"?
no
RETURN response
no
no

***Fig. 2:*** *Organizer workflow*

The retrofitting architecture is configurable, such that a Website's operator can activate or deactivate individual retrofitters. Due to that configurable and module-based architecture, RETROCSP can be easily extended by adding new retrofitting modules. Moreover, retrofitters can be removed from the framework as soon as they are no longer needed, for example, if the retrofitted feature is supported by all browsers.

## *B. Organizer*

The core logic of the Service Worker is the *organization* functionality. It is responsible for handling Hypertext Transfer Protocol (HTTP) requests initiated by the client, communicating with the server to fetch a requested resource, parsing a potentially attached CSP and scheduling the corresponding retrofitters.

The standard workflow of the Organizer can be seen in Fig. 2. Whenever a client initiates a request, the Service Worker's `fetch` event will fire. As soon as the browser issues a request, the Service Worker intercepts this request and simulates the default browser behavior by fetching the requested resource from the server. If this request succeeds, the Organizer inspects the `Content-Type` header of the response to determine the response body's media type [17]. A CSP header is only relevant if the requested resource is a HTML document. Therefore, if the response type is not an HTML document, the response is directly forwarded to the client as no retrofitting has to be done for those resources.

Only if the response's content type is `text/html`, it is scanned for a Content Security Policy. First, the organizer scans the HTML document for CSPs declared via `meta` elements and then checks whether the `Content-Security-Policy` header is set. If no CSP was specified by the HTTP response, the unmodified response is passed on.

All found CSPs are merged into *one* functionally equivalent comma-separated Content Security Policy header. In order to be equivalent, `frame-ancestors`, `sandbox` and `report-uri` directives are removed from CSPs declared by `meta` elements since they are illegal in such a context. Besides this, all `meta` elements defining a CSP are removed from the HTML document. This introduces one possible minor functional change. A CSP specified via a `meta` element does not apply to elements that are declared above this exact element within the HTML. By merging the CSPs, this case cannot be tracked anymore. Apart from this, merging meta and header CSPs does not alter the functionality nor security guarantees provided by the CSP.

The composed CSP is strictly parsed according to the standard [34] by the Organizer. Nevertheless, two modifications are done during this process. First, if a policy within the CSP chain does not specify an explicit `script-src` directive but declares its fallback, a `default-src` directive, a new `script-src` directive is appended to the corresponding policy. The source-expressions of the `default-src` are then used as expressions for the newly created directive. Furthermore, if the `default-src` contains the `'strict-dynamic'` keyword, it is removed from the `default-src` directive's source list as it is no longer responsible for scripting content. Notably, this keyword has no semantic meaning for any other directive apart from `script-src`. Moreover, this eases retrofitting CSP features targeting scripting content as it is no longer required to analyze the potentially clobbered `default-src` fallback. As a second modification, the CSP has to be changed if it does not allow all inline script content. A CSP allows any inline script content if either none of its policies specifies a `script-src` directive (or indirectly via a `default-src`) or all directives governing scripting resources contain the `'unsafe-inline'` keyword. If this is not the case, the retrofitters need a way to inject *authenticated* JavaScript code into the response. For this reason, RETROCSP generates a *cryptographically secure* nonce. This *retrofitting nonce* is inserted into all such restrictive policies. Thus, neither functionality nor security is negatively affected by these modifications.

If we can parse the CSP successfully, the Organizer successively initiates the *retrofitters*. The Organizer holds a list of all active retrofitting modules. Hence, retrofitting modules can be easily de-/activated by being removing or appending entries to this list. As JavaScript is single-threaded, this list serves as a task queue. The Service Worker takes care of the retrofitter's tasks one by one, executing one after the other. If this *retrofitting phase* has finished, the organizer component is in control of the Service Worker again.

The Organizer concludes its work by updating the `Content-Security-Policy` header with its modified value. The retrofitted HTTP response with the updated header and modified body is delivered to the client. The browser

```
class Retrofitter {
    static retrofit(csp, responseText) {
        ... // code executed on the proxy side
    }
}

Retrofitter.retrofittingScript = function (arguments) {
    ... // code executed on the client side
};
```

***Fig. 3:*** *Semantic structure of a retrofitter*

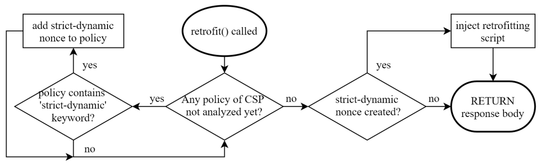
add strict-dynamic nonce to policy
retrofit() called
inject retrofitting script
yes
yes
policy contains 'strict-dynamic' keyword?
yes
Any policy of CSP not analyzed yet?
no
strict-dynamic nonce created?
no
RETURN response body
no

***Fig. 4:*** *Proxy phase of the StrictDynamicRetrofitter*

interprets and enforces the adjusted CSP and response. From a user's point of view, this complete process is not distinguishable from a normal server response. The Service Worker is a transparent client-side proxy.

## *C. Retrofitting*

RETROCSP's retrofitting phase is performed by the retrofitting modules of the Service Worker. In principle, *one* retrofitter is responsible for *one* CSP feature. This can either be an entire directive or a source expression of a specific directive. In this work, we implement *three* distinct retrofitting modules.

When retrofitting a CSP feature, it is important that this is done browser-independently. After retrofitting, the feature has to work correctly for browsers that did not support it as well as for browsers that correctly implemented it from the very beginning. Syntactically speaking, retrofitters are JavaScript classes. The execution logic of retrofitters is separated into two individual *phases*. The first phase, the *proxy phase*, completely runs on the Service Worker side.

To initiate the proxy-phase, the `retrofit` method is called. It implements the proxy-side retrofitting mechanism. The Organizer passes the CSP and the server response's body to the retrofitter. First, the CSP is analyzed. The retrofitter checks if there is anything to be retrofitted that it is responsible for. During this phase, the CSP can be modified. If the feature the current retrofitter governs does not occur in the CSP, it returns the unmodified response. If the CSP has been changed, the second phase, the *client phase*, is initiated. In order to be able to execute code in the context of the HTML document, the Service Worker injects JavaScript code into the DOM. This script is injected as an authenticated inline script, that carries the *retrofitting nonce* such that the CSP does not block its execution. The code that is executed on the client-side is specified by the retrofitter's `retrofittingScript`. This is an anonymous function implementing any client-side retrofitting mechanism, e.g., the hooking of a Document Object Model (DOM) API. To pass *proxy-side* arguments to the code executed on the *client side*, the function is enclosed by a closure providing any pre-established arguments. This additionally *isolates* the retrofitter's scope from the main scope of the Web site. The retrofitter can access and modify the document's global scope, however, its *own* variables and functions cannot be accessed or tampered with from the outside. The authenticated retrofitting script is appended to the beginning of the HTML's head section, such that it is the first code being executed in the context of the Web site. This assures that crucial API methods are hooked before they are used for the first time. The retrofitter module is in the *client* phase when the retrofitting script is executing on the client-side. This phase is no longer performed by the Service Worker but takes place asynchronously on the client-side.

In the following subsections, we describe each of the three implemented retrofitters in more detail.

*1) StrictDynamicRetrofitter:* The first retrofitter is the *StrictDynamicRetrofitter*. It is responsible for the `script-src` directive's `'strict-dynamic'` source-expression. This source-expression is one of the core CSP additions introduced by Level 3 [32]. It is supported by all modern browsers, except *Safari* (see Table I). `'strict-dynamic'` allows to *propagate trust*. This can ease the usage of third-party code in a Web site, as their trust is automatically propagated, such that a Web sites operator no longer needs to rely on host-based allowlists [30].

The *challenge* of `'strict-dynamic'` resides in the retrofitting of this *trust propagation*. *Trust*, in the context of the expression, can be achieved via nonces. If a resource can provide a valid nonce, it passes the CSP and is trusted as a result. We leverage the fact that an including *including* script can pass its own nonce to a newly *inserted* script and thereby propagate its trust. The *StrictDynamicRetrofitter* creates a cryptographically secure *strict-dynamic nonce*, similar to the retrofitting nonce, during the *proxy* phase. This nonce is injected into all policies that contain the `'strict-dynamic'` keyword. During the retrofitter's *client* phase, the nonce is utilized to simulate the trust propagation.

The establishment of the strict-dynamic nonce represents almost the entire *proxy* phase of the *StrictDynamicRetrofitter*. The workflow of the *StrictDynamicRetrofitter*'s proxy phase can be seen in Fig. 4. The retrofitter analyzes all policies within the CSP. It generates and adds the *strict-dynamic nonce* to all of those that include the `'strict-dynamic'` source expression in a `script-src` directive. In addition to that, it removes all host source expressions in the source list of such directives as `'strict-dynamic'` disables host-based allowlisting for script resources. If it did not find any policy containing the keyword, no strict-dynamic nonce is generated, and no client-side retrofitting script is injected into the HTTP response's body. As such, this retrofitter only has a *client* phase if a strict-dynamic nonce was generated.

During its *client* phase, the *StrictDynamicRetrofitter* implements the trust propagation mechanism. When trying to dynamically include `script` elements in a non-parser-inserted way, the element itself must be created first. This is the case for all available API methods that allow the addition of new elements to the DOM, such as `Node.insertBefore()`, `Node.appendChild()`, `ParentNode.append()` and `ParentNode.prepend()`. Therefore, it suffices to hook those APIs that enable to programmatically create an HTML element, namely `Document.createElement()` and `Document.createElementNS()`.

Both functions are hooked in the same way. First, the original unmodified API method is called, by the hook. Then, before returning the resulting element, the strict-dynamic nonce that was generated during the *proxy* phase is added to the element. The hooking of these functions ensures that all programmatically created elements are authenticated. This furthermore does not tamper with the security as an attacker can only abuse these modified API methods if they are able to execute code in the Web site's origin in the first place.

*2) UnsafeHashesRetrofitter:* The second retrofitter is the *UnsafeHashesRetrofitter*. It takes care of the `'unsafe-hashes'` source-expression. Similar to the previous retrofitter, this one is also part of the `script-src` directive. The `'unsafe-hashes'` source-expression is also one of the major changes introduced by CSP Level 3 [32], such that developers can now not only allow the executions of inline scripts via hashes, but can now also use hashes to allow inline JavaScript events added to DOM elements.

The *proxy* phase of the *UnsafeHashesRetrofitter* consists of two steps. First, it collects all hash values that are allowed to be used for inline event handlers via the `'unsafe-hashes'` source expression. If the CSP allows all inline script content via `'unsafe-inline'`, the client phase is skipped since there is nothing to be retrofitted.

Because some browsers, for example Firefox, have the unsafe-hashes behavior as default, the *UnsafeHashesRetrofitter* injects a `retrofittingScript` even if no `'unsafe-hashes'` source expression is present in the CSP. As a consequence, the retrofitter changes to the *client* phase if it encounters any hash source-expression during the scan of the CSP, such that also the wrong default behavior is fixed by the retrofitter. For example, the absence of `'unsafe-hashes'` dictates that hash sources are not valid for inline event handlers or `javascript:` navigation, which must be blocked as a consequence. In addition to that, the retrofitter scans all policies for a `form-action` directive, or its fallback `navigate-to`, that does not contain a `javascript:` scheme source expression. Code execution through `javascript:` URLs within `form` elements has to pass the `script-src` directive *and* `form-action` check. Thus, we have to keep track of whether such a restrictive `form-action` directive that does not allow code execution is present. The presence of such a restrictive `form-action` directive, as well as the list of allowed hashes, are provided to the retrofitting script that the *UnsafeHashesRetrofitter* injects to the DOM of the Web site.

During the retrofitter's *client* phase, it has to retrofit inline event handler and `javascript:` URL control checks. For most of its work, it uses a *Mutation Observer* [35]. This Mutation Observer enables the retrofitter to react during the HTML parsing process and, most importantly, before crucial `load` events trigger. This allows the *UnsafeHashesRetrofitter* to block inline event handlers before they execute. At the same time, it can retrofit the functionality of inline event handlers or `javascript:` navigation that the browser would block due to missing support of the `'unsafe-hashes'` keyword source expression. Whenever a new element is modified or added to the DOM, the Mutation Observer is triggered and can check whether the element has to be retrofitted before it is actually added to the DOM.

```
function defineGlobalFunction(code) {
  let functionDefinition = document.createElement('script');
  let functionName = `globalFunction${functionID++}`;
  functionDefinition.innerText = `function
↪  ${functionName}(event){${code}}`;
  functionDefinition.nonce = retrofittingNonce;
  document.head.prepend(functionDefinition);
  return window[functionName];
}
```

***Fig. 5:*** *Transforming text content into executable code*

This comes with *two* major challenges. First, as `'unsafe-hashes'` is completely based on hashes, the retrofitter has to be able to compute the CSP hash value of JavaScript code. Secondly, this has to be done synchronously, such that an asynchronous event does not trigger before the element is fully retrofitted. Since JavaScript in the browser only provides an asynchronous cryptography API [29] due to performance reasons, we had to switch to a synchronous third-party JS library. We decided to use Jeff Mott's `CryptoJS` [20], as it provides all necessary hashing algorithms, namely `SHA256`, `SHA384`, and `SHA512`.

When retrofitting an element's inline event handler or `javascript:` URL attribute, we first check if the underlying code is allowed by one of the hashes collected during the *proxy* phase. If *none* of the three hashing algorithms supported by CSP yields a valid (algorithm, hash) pair, the code is *untrusted*. Untrusted code execution must be prevented, e.g., by using the `Event.preventDefault()` method that prevents an inline event handler from executing and does not interfere with event handlers that are added programmatically. However, if any (algorithm, hash) pair is valid, the code is *trusted* and can be executed. For this to work, the attribute's value has to be converted from a string into executable code. Transforming text content into code can be dangerous, because this can lead to a trivial code injection vulnerabilities if the text contains any user-controllable input. Unless the `'unsafe-eval'` keyword is set, a CSP prevents the usage of `eval()` and similar functions as long as a script content controlling directive is specified. As we did not want to depend on this security eroding keyword, we had to implement a way to *securely* transform text into executable code.

We use the exact same functionality as the *StrictDynamicRetrofitter*. Fig. 5 shows the gadget we implemented within the retrofitting script. It transforms text content into code by creating a new *authenticated* inline script element using RETROCSP's *retrofitting nonce*. The script defines the code as a function in the document's global scope such that it can be accessed from within the retrofitter's scope. This newly created `script` element is prepended to the HTML head such that it is executed immediately.

The function in Fig. 5 simply returns a handle to the newly defined global function that can be used to retrofit the event handler or `javascript:` URL attribute. This can be done by specifying a corresponding event handler programmatically. The code-generating gadget itself does not provide an additional security risk as the retrofitting script is isolated in its closure. Thus, it is not callable from the outside.

In addition to the Mutation Observer, the *UnsafeHashes-Retrofitter* also hooks the `window.open()` method, because it can be used to execute code via a `javascript:` URL. Hence, the exact same hash value computations and CSP checks are performed here. Only if the code provided by the URL is trusted through a valid (algorithm, hash) pair, the `window.open()` call is executed. Otherwise, `window.open` behaves as if it was is provided with an invalid URL and code execution is prevented.

*3) NavigateToRetrofitter:* The `navigate-to` directive is a completely new directive added in CSP Level 3. As such, it introduces a completely new way to *control* navigation events, in order to prevent unintended data leakage by restricting what navigations the document may *initiate*. At the time of this writing, the directive is not supported by any modern browser [14, 15]. As a consequence, we created the *NavigateToRetrofitter* as a third retrofitting module of RETROCSP.

The *NavigateToRetrofitter* begins its *proxy* phase by collecting the set of sources allowed by a `navigate-to` directive. In order to control navigation events issued by HTML forms, if a `navigate-to` directive is present within a policy, but no `form-action` directive is set, the module already starts to retrofit on the proxy side. Therefore, a new `form-action` directive is added to the policy that inherits *all* sources of its fallback. Since the `form-action` directive is fully supported by all modern browsers [15, 14], this already correctly implements the form-based navigation checks by transferring this task to another directive.

The retrofitter also prepares the HTML document to be retrofitted during the *client* phase by modifying `meta` elements that initiate navigation, so-called *meta refreshes*.

This kind of navigation redirects the client to a specified target after a certain period of time. To prevent prohibited navigations, the *NavigateToRetrofitter* has to modify the element's attributes. Because a navigation initiated by a meta refresh can not be canceled if it is scheduled, the `content` attribute's name is replaced with `refresh-target`. This hinders the navigation as the browser no longer understands the attribute. The retrofitter, however, can process this attribute of the meta tag during the *client* phase.

Because `javascript:` URLs can be used for code execution wherever standard navigation are initiated, the *client* phase of the *NavigateToRetrofitter* is almost equivalent to the one of the *UnsafeHashesRetrofitter*. Notably, we did not consider `javascript:` navigation as navigation that is governed by the `navigate-to` directive for several reasons. First of all, those URLs are merely a way to *execute inline code* whereas `navigate-to` controls navigations. The `script-src` directive is already responsible for this, and code execution itself does not count as a navigation event. The CSP Level 3 specification [32] also does not state that the `navigate-to` directive would be responsible for this kind of code execution. As such, code execution could only be allowed by adding a `javascript:` scheme source expression to the `navigate-to` directive, this would further not really improve the security in any way. It would only be possible to allow all, or no `javascript:` navigation via the `navigate-to` directive. In addition to the reasons presented so far, other directives like the `frame-src` directive are not triggered if a resource they are governing is specified using a `javascript:` URL.

| **Browser** | strict-dynamic | unsafe-hashes | navigate-to |
|---|---|---|---|
| Chrome | ✓ | ✓ | ✗ |
| + RETROCSP | ✓ | ✓ | ✓ |
| Edge | ✓ | ✓ | ✗ |
| + RETROCSP | ✓ | ✓ | ✓ |
| Opera | ✓ | ✓ | ✗ |
| + RETROCSP | ✓ | ✓ | ✓ |
| Firefox | (✓) | (✗) | ✗ |
| + RETROCSP | ✓ | ✓ | ✓ |
| Safari | ✗ | ✗ | ✗ |
| + RETROCSP | ✓ | ✓ | ✓ |

***TABLE II:*** *Functionality overview of* RETROCSP *per browser*

As before, the retrofitter hooks the `window.open()` method and registers a Mutation Observer that reacts to all modified or injected elements. If these elements fall under the scope of the `navigate-to` directive, the access control checks specified by the CSP standard are performed. We *strictly* implemented the algorithm defined by the standard [33] to check whether a URL is a valid navigation target. Navigation initiated by the document is only allowed if the target matches a valid source declared by the CSP.

## V. Evaluation

Here, we evaluate RETROCSP with respect to retrofitted functionality and its performance.

### A. Functionality

Here, we discuss the individual retrofitted functionality in more detail (see Table II for an overview).

*1) strict-dynamic:* All *chromium-based* (*Google Chrome*, *Microsoft Edge* and *Opera*) browsers completely support the `'strict-dynamic'` source expression out of the box. They successfully passed all of the test cases. RETROCSP does not need to tamper with the browsers' functionality. Even when we deploy RETROCSP, our tests indicate that `'strict-dynamic'` still works as intended and we did not detect any other defects.

*Mozilla Firefox* does not *fully* implement `'strict-dynamic'`. It behaves correctly if the keyword is part of a `script-src` directive's source list. However, the source expression is not supported when present in a `default-src` directive. In this case, it is ignored. The retrofitting architecture does not interfere with the browser's correct `script-src` behavior but fixes the missing support for the `default-src` directive. Thus, with RETROCSP, *Mozilla Firefox* adheres to the standard and passes all examined test cases.

As stated by *MDN Web Docs* [15] and *Can I Use* [14], *Safari* is not compatible with `'strict-dynamic'`. We were able to verify this finding at the time of our experiments. Note that Webkit has since adopted a patch to support strict-dynamic, but the current version of Safari at the time of this writing (15.1) does not yet support it. Safari simply ignores the source expression and enforces the CSP as if the keyword was not specified. As such, scripting resources only allowed by their host are not blocked but loaded since host-based allowlists are still valid. Moreover, scripts cannot indirectly propagate their trust to non-parser-inserted scripts. RETROCSP completely re-implements support for `'strict-dynamic'`. Our tests show that using RETROCSP, `'strict-dynamic'` is correctly enforced in the Safari browser.

*2) unsafe-hashes:* Just as for the previous source expression, all chromium-based browsers implement the `'unsafe-hashes'` keyword correctly. With and without RETROCSP, these browsers passed all test cases.

According to *MDN Web Docs* [15] and *Can I Use* [14], the *Firefox* browser does not support `'unsafe-hashes'`. As a result of the evaluation, we conclude that this is only partially true. In fact, it is the other way around. *Firefox* enforces `'unsafe-hashes'` even when it is not indicated by the CSP. For this reason, RETROCSP has to invoke the *UnsafeHashesRetrofitter* even when said source expression cannot be found in the CSP. The retrofitting architecture fixes this behavior accordingly and when deployed in *Firefox*, the browser now correctly implements `'unsafe-hashes'` as specified by the CSP standard [32].

Again, Safari does not support this keyword source expression at all. Instead, it is ignored and the browser suggests setting the `'unsafe-inline'` keyword.

RETROCSP enables the basic usage of `'unsafe-hashes'` but fails to pass one test case. The `window.location` object can be used to execute code by navigating the client to a `javascript:` URL. This code execution cannot be governed by RETROCSP. Due to security reasons, it is not possible to modify and hook the `window.location` object as it is non-configurable. Thus, even when RETROCSP is deployed on a Web site, *Safari* is still overblocking for this kind of code execution and would require the `'unsafe-inline'` source expression. For *Firefox*, it is the other way around. RETROCSP cannot block such code execution if the CSP contains a valid (algorithm, hash) pair but no `'unsafe-hashes'` source expression. The code execution performed by *Firefox* cannot be undone.

*3) navigate-to:* The `navigate-to` directive could be verified to be unsupported by any modern browser. All browsers allowed all initiated navigations, regardless of the CSP.

The retrofitting architecture universally implements the `navigate-to` directive for all of these browsers. It enables developers to use this new directive without adverse effects, although one test case could not be passed successfully due to the `window.location` object. As it is non-configurable, the directive's navigation checks could not be integrated into this object. Hence, we cannot guarantee unlimited browser support for this directive.

*a) Summary:* All in all, we were able to verify the information shown in Table I by manually testing each targeted browser's default behavior. However, the precise extent of CSP support fluctuated for the *Mozilla Firefox* browser in particular. Table II provides an overview of the status of each targeted browser and the retrofitting architecture. When deployed on a Web site, RETROCSP ensures that browsers correctly enforce the retrofitted CSP features. Moreover, it does not tamper with the functionality of browsers that already correctly implement these source expressions and the directive. Nonetheless, navigations initiated by using `window.location` cannot be verified nor blocked by RETROCSP.

## B. Performance

To measure the performance of RETROCSP, we measured both the loading time overhead and the runtime overhead of it. In this section, we present an overview of the results. Due to space restrictions, we do not discuss the exact details of our performance analysis here. We instead refer the reader to Appendix A in the Appendix.

To evaluate the overhead in loading time and the added checks of RETROCSP, we first create five different CSPs aimed at triggering different retrofitters (shown in Fig. 6). This includes a CSP to trigger the three retrofitters individually (CSP 1–3), an empty CSP as a baseline (CSP 4), and a combination which triggers all retrofitters at the same time. We then measured the overhead in two dimensions: delay in load time of the page itself for each of the CSPs and specifically the execution overhead for each retrofitter.

To account for potential outliers and ensure meaningful data collection, we ran each test 1,000 times. Since the loading time is increased through the synchronous hash computation and comparisons, we expect certain delays. Overall, the loading time overhead is at most 23ms. This occurs for CSP 5, which triggers all retrofitters at the same time. In addition to the overall overhead, we also consider the per-retrofitter overhead. To that end, we micro-benchmark the execution overhead incurred by each retrofitter separately. For a baseline, we first run a vanilla version of RETROCSP, which has all retrofitters disabled. For this, we measure a load time of 20-21ms; any load time beyond that is then the overhead of the retrofitters. When activating *all* retrofitters, we find an overhead of 21-23ms in the execution of the retrofitters. However, by individually testing the overhead of each retrofitter, the delay primarily originates from the `'unsafe-hashes'` retrofitter. This scales linearly with the number of hashes to be compared, since each script needs to be hashed in a synchronous fashion. Considering that Web sites in practice do not make extensive use of hashes [22], we do not expect that site operators add a significant number of these, implying that the overall overhead remains low.

```
# CSP 1
Content-Security-Policy: script-src 'self' 'nonce-NjUxNTEzNTYzMjgzMzI0NA=='  'strict-dynamic'; worker-src 'self'
# CSP 2
Content-Security-Policy: script-src 'self' 'nonce-NjUxNTEzNTYzMjgzMzI0NA=='
↪  'sha256-RCMdviIzxShMQNOs7R/Pq3EjJM6L0MxdBYyGSEbHucQ=' 'sha256-AOnA0C6HptNjWEo+4mP1/LTfE1YA+7+gilj9qlv+fMQ='
↪  'sha256-o1p1KBgob2gsq9r4DxsE82JZ02L71JMECff4xWMwZWY=' 'sha256-BVKPTh0klRSsJxt1gZGDRddHVlcb7xX0ZWk4UJi7uLA='
↪  'sha256-62p6MzdLMN+pbb/Do7BxJiBN/vUcxNAsTuns/guIqew=' 'unsafe-hashes'
# CSP 3
Content-Security-Policy: navigate-to 'self'
# CSP 4
Content-Security-Policy:
# CSP 5
Content-Security-Policy: script-src 'self' 'nonce-NjUxNTEzNTYzMjgzMzI0NA==' 'strict-dynamic'
↪  'sha256-RCMdviIzxShMQNOs7R/Pq3EjJM6L0MxdBYyGSEbHucQ=' 'sha256-AOnA0C6HptNjWEo+4mP1/LTfE1YA+7+gilj9qlv+fMQ='
↪  'sha256-o1p1KBgob2gsq9r4DxsE82JZ02L71JMECff4xWMwZWY=' 'sha256-BVKPTh0klRSsJxt1gZGDRddHVlcb7xX0ZWk4UJi7uLA='
↪  'sha256-62p6MzdLMN+pbb/Do7BxJiBN/vUcxNAsTuns/guIqew=' 'unsafe-hashes'; navigate-to 'self'; worker-src 'self'
```

***Fig. 6:*** *Content-Security-Policy Headers Used in Evaluation*

At first glance, the overhead of RETROCSP looks extremely high; we seemingly add 23ms overhead to a 20ms load time, implying more than 100% overhead. This, however, is in fact not correct. Rather, the overhead is exaggerated by our use of localhost as a Web server. In practice, the execution overhead does not depend on the network connection, but the computation only occurs in the Service Worker, i.e., it is unaffected by the time it takes to load a page. In contrast, an average Web page takes 10.3s to load [13]. In light of this, an overhead of 23ms actually implies that the overhead in practical deployment is merely 0.2% and hence negligible.

## VI. Discussion

RETROCSP achieves to retrofit security on the Web. We have shown that the chosen CSP features could be successfully re-implemented; yet RETROCSP has its limitations.

First, our evaluation and implementation only focus on desktop browsers. We leave an adoption of RETROCSP to future work. In addition, RETROCSP requires Service Workers to function. This requires that the browser support this technology. More importantly, though, Service Workers can only operate on secure origins, i.e., require a valid TLS connection. However, in the age of automated tools such as *Let's Encrypt* [19], this appears feasible.

In order to deploy RETROCSP successfully, developers have to host the underlying Service Worker themselves. This is due to the fact that Service Workers have to be of the same origin as the scope they are controlling. Furthermore, developers have to serve the Service Worker at the root directory of the Web site or alternatively attach a `Service-Worker-Allowed` header to the corresponding HTTP response. Since developers *want* to use the retrofitting architecture when trying to deploy it, this should not be an obstacle. The Service Worker itself is a light-weight JavaScript file, consisting of 781 source lines of code that sum up to only 46 KB of data.

As RETROCSP is code that is executed, it has to be trusted by the CSP itself. Thus, developers have to explicitly trust the Service Worker by adjusting the CSP. However, CSP provides a directive for this purpose, the `worker-src` directive. Adding a `worker-src 'self'` or declaring the explicit URL of the Service Worker suffices to update the CSP to trust the architecture. The directive itself is supported by all browsers apart from *Safari* [16, 14]. Manual testing revealed that *Safari* trusts all Service Workers automatically. As such, no extra configuration has to be done to support *Safari* users.

During the implementation of the *UnsafeHashesRetrofitter* and *NavigateToRetrofitter*, we recognized that the `window.location` object cannot be retrofitted with access control checks due to its non-configurability. Here, security considerations hindered us from integrating further security mechanisms into the `window.location` object by hooking it. As a result, it was not possible to govern `javascript:` URL navigation performed by modifying the object's `href` attribute. Moreover, it was also not feasible to integrate the navigation checks defined by the `navigate-to` directive.

The `navigate-to` directive also provides a unique source expression only valid for this directive. The `'unsafe-allow-redirects'` keyword source expression allows navigations initiated by the document to be trusted if the navigation resolved to a trusted endpoint after an arbitrary amount of redirects. With this configuration, `navigate-to` implicitly trusts all redirections. However, since our solution is JavaScript-based and intercepts redirections before they occur, it is infeasible for RETROCSP to implement the directive.

Nevertheless, RETROCSP is able to address some of the most pressing issues in lacking browser support for CSP. With RETROCSP, developers are able to deploy policies that do not require hacks to allow for compatibility, which sacrifices security. For immediate use by site operators and for reproducibility, we release our prototype implementation of RETROCSP publicly [1].

## VII. Conclusion

Developers aiming to deploy a CSP are often hindered by the lack of universal support of all CSP directives. Moreover, even if browsers seemingly implement the latest specification, their implementations have inconsistencies. In this paper, we designed, implemented, and evaluated RETROCSP, a retrofitting utility to bring improved CSP support to browsers. It comes with a low overhead and not only retrofits functionality but also fixes implementational bugs in major browsers. Beyond this, our work and thorough set of tests allowed us to find and report multiple bugs both in implementations as well as the CSP specification.

## REFERENCES

[1] Anonymous. retroCSP implementation. https://anonymous.4open.science/r/retroCSP.

[2] Anonymous. Anonymized for Submisison, October 2016. (accessed on 31/03/2022).

[3] Anonymous. Anonymized for Submisison, December 2020. (accessed on 31/03/2022).

[4] Anonymous. Anonymized for Submisison, January 2021. (accessed on 31/03/2022).

[5] Anonymous. Anonymized for Submisison, January 2021. (accessed on 31/03/2022).

[6] Anonymous. Anonymized for Submisison, January 2021. (accessed on 31/03/2022).

[7] Jake Archibald. Is ServiceWorker ready? https://jakearchibald.github.io/isserviceworkerready/. (accessed on 31/03/2022).

[8] Jake Archibald. Introducing Background Sync. https://developers.google.com/web/updates/2015/12/background-sync, September 2017. (accessed on 31/03/2022).

[9] A. Barth. The Web Origin Concept. RFC 6454 (Proposed Standard), December 2011. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc6454.txt.

[10] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Internet Standard), January 2005. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc3986.txt. Updated by RFCs 6874, 7320.

[11] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. Content security problems?: Evaluating the effectiveness of content security policy in the wild. In *CCS*, 2016.

[12] A. Costello. Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA). RFC 3492 (Proposed Standard), March 2003. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc3492.txt. Updated by RFC 5891.

[13] Brian Dean. We Analyzed 5.2 Million Webpages. Here's What We Learned About PageSpeed. https://backlinko.com/page-speed-stats, October 2019. (accessed on 31/03/2022).

[14] Alexis Deveria. Can I use... - Content Security Policy. https://caniuse.com/?search=Content%20Security%20Policy, 2021.

[15] Mozilla Foundation. Content Security Policy - Browser compatibility. https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP, 2021.

[16] Mozilla Foundation. MDN Web Docs - CSP: worker-src. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/worker-src, February 2021. (accessed on 31/03/2022).

[17] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. RFC 2045 (Draft Standard), November 1996. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc2045.txt. Updated by RFCs 2184, 2231, 5335, 6532.

[18] Matt Gaunt. Service Workers: an Introduction. https://developers.google.com/web/fundamentals/primers/service-workers, September 2021. (accessed on 31/03/2022).

[19] Internet Security Research Group (ISRG). Let's Encrypt. https://letsencrypt.org/. (accessed on 31/03/2022).

[20] Jeff Mott. CryptoJS - JavaScript implementations of standard and secure cryptographic algorithms. https://code.google.com/archive/p/crypto-js/. (accessed 31/03/2022).

[21] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 736–747, 2012.

[22] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. Complex security policy? a longitudinal analysis of deployed content security policies. In *NDSS*, 2020.

[23] Sebastian Roth, Lea Gröber, Michael Backes, Katharina Krombholz, and Ben Stock. 12 Angry Developers - A Qualitative Study on Developers' Struggles with CSP. In *ACM CCS*, 11 2021.

[24] Alex Russel, Jungkee Song, Jake Archibald, and Marjin Kruisselbrink. Service Workers Nightly. Editor's draft, W3C, August 2021. URL https://w3c.github.io/ServiceWorker/.

[25] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *International Conference on World Wide Web (WWW)*, 2010.

[26] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild. In *Network and Distributed Systems Symposium (NDSS)*, 2019.

[27] Marius Steffens, Marius Musch, Martin Johns, and Ben Stock. Who's hosting the block party? studying third-party blockage of csp and sri. In *Network and Distributed Systems Security (NDSS) Symposium 2021*, 2021.

[28] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. How the web tangled itself: Uncovering the history of client-side web (in) security. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 971–987, 2017.

[29] Mark Watson. Web Cryptography API. W3C recommendation, W3C, January 2017. URL https://www.w3.org/TR/2017/REC-WebCryptoAPI-20170126/.

[30] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. CSP is dead, long live CSP! On the insecurity of whitelists and the future of content security policy. In *CCS*, 2016.

[31] Michael Weissbacher, Tobias Lauinger, and William Robertson. Why is CSP failing? Trends and challenges

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta http-equiv="Content-Security-Policy"
    ↪  content="default-src
    ↪  https://upload.wikimedia.org/wikipedia/en/9/95/
    ↪  Test_image.jpg 'strict-dynamic'">
</head>
<body>
<img src="https://upload.wikimedia.org/wikipedia/en/9/95/
↪  Test_image.jpg">
</body>
</html>
```

***Fig. 7:*** *Standard Compliance Test Case #1*

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta http-equiv="Content-Security-Policy"
    ↪  content="default-src 'nonce-ABCDEF'
    ↪  'sha256-sOq4p3/IUmdg10+FT4za4DO2/MDyaP9Aw+TRyl1Y09A='"></script>
</head>
<body>
<script nonce="ABCDEF">alert(1337)</script>
<script>alert(42)</script>
</body>
</html>
```

***Fig. 8:*** *Standard Compliance Test Case #2*

in CSP adoption. In *RAID*, 2014.

[32] Mike West. Content Security Policy Level 3. W3C working draft, W3C, October 2018. URL https://www.w3.org/TR/2018/WD-CSP3-20181015/.

[33] Mike West. navigate-to Pre-Navigation Check. W3C working draft, W3C, October 2018. URL https://www.w3.org/TR/CSP3/#navigate-to-pre-navigate.

[34] Mike West. Parse a serialized CSP list. W3C working draft, W3C, 2021. URL https://www.w3.org/TR/CSP3/#parse-serialized-policy-list.

[35] WHATWG. Mutation Obsevers. https://dom.spec.whatwg.org/#mutation-observers, March 2022. (accessed on 31/03/2022).

[36] WHATWG. Response() in the Fetch Standard. https://fetch.spec.whatwg.org/#dom-response, March 2022. (accessed on 31/03/2022).

## APPENDIX

*1) Loading time overhead:* To measure the load overhead of the entire architecture, we computed the total loading time of the page shown in Fig. 10 in the Appendix. This represents a merged and simplified version of the test cases used for our functionality evaluation. In addition to that, we deactivated caching for all evaluated browsers to not distort the measurements. To ensure that all client-side retrofitting has finished, we wait until the `onload` is fired.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="Content-Security-Policy"
    ↪  content="script-src https://kündigungsschutz.com/">
</head>
<body>
<script>alert(42)</script>
</body>
</html>
```

***Fig. 9:*** *Standard Compliance Test Case #3*

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Evaluation</title>
</head>
<body>
<!-- 'strict-dynamic' features -->
<script nonce="NjUxNTEzNTYzMjgzMzI0NA==">
    console.log("[SD] nonced inline script (uses
↪   document.write)");
    document.write("<scr" + "ipt>console.log('[SD] parser
↪   inserted script')</scr" + "ipt>");
</script>

<script nonce="NjUxNTEzNTYzMjgzMzI0NA==">
    console.log("[SD] nonced inline script (uses
↪   document.body.appendChild)");
    let scriptElement1 = document.createElement("script");
    scriptElement1.text = "console.log('[SD] non-parser
↪   inserted inline script')";
    document.body.appendChild(scriptElement1);
</script>

<!-- 'unsafe-hashes' features -->
<img src="" onerror="console.log('[UH] should be blocked
↪   (inline event handler)')">
<img src="" onerror="console.log('[UH] inline event
↪   handler')">

<img id="img" src="">
<script nonce="NjUxNTEzNTYzMjgzMzI0NA==">
    document.getElementById("img").setAttribute("onerror",
↪   "console.log('[UH] attribute change')")
</script>

<iframe src="javascript:console.log('[UH] iframe JS
↪   URL')"></iframe>

<a href="javascript:console.log('[UH] hashed JS URL with
↪   protocol')">with protocol</a>
<a href="javascript:console.log('[UH] hashed JS URL without
↪   protocol')" onclick="console.log('[UH]
↪   onclick')">without
    protocol</a>
<a href="javascript:console.log('[UH] should be blocked
↪   (a-tag href)')" onclick="console.log('[UH]
↪   onclick')">not
    hashed</a>

<!-- navigate-to features -->
<a
↪   href="http://localhost:8000/navigate-to/">localhost/navigate-to</a>
<a href="http://not-localhost:8000/">not-localhost</a>

<form
↪   action="http://localhost:8000/navigate-to/#from-action"
↪   method="post">
    <input type="text" name="fieldName" value="fieldValue">
    <input type="submit" value="submit">
</form>

<form action="http://not-localhost:8000/#form-action"
↪   method="post">
    <input type="text" name="fieldName" value="fieldValue">
    <input type="submit" value="submit">
</form>
</body>
</html>
```

***Fig. 10:*** *HTML document used for performance evaluation*

We begin the evaluation by analyzing the impact of the CSP structure on the performance of RETROCSP. For this purpose,

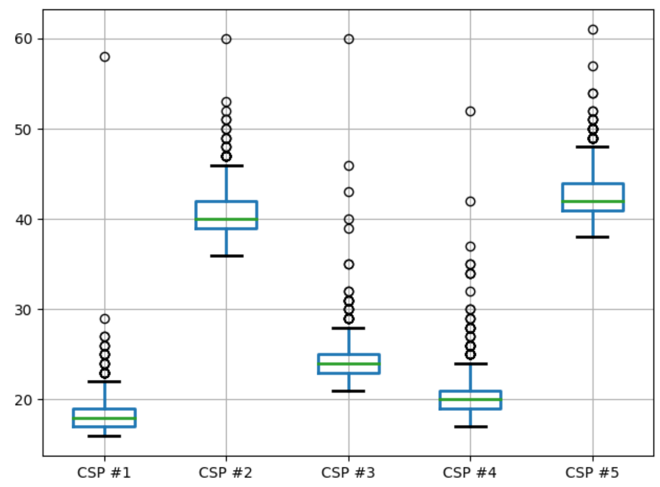

***Fig. 11:*** *Google Chrome: Loading time of evaluation Web site per CSP, showing time in milliseconds*

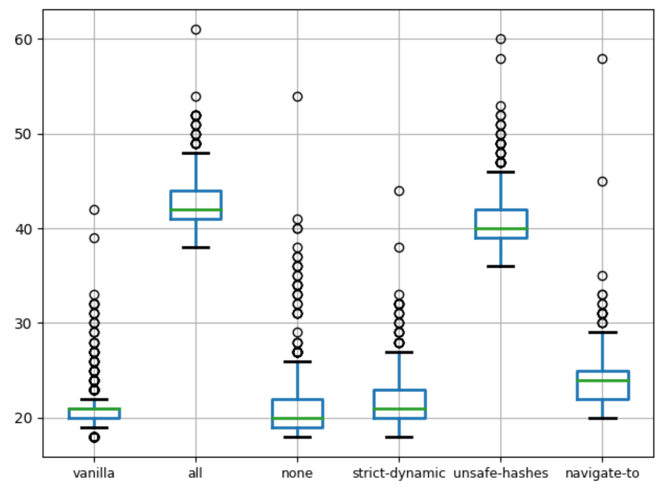

***Fig. 12:*** *Google Chrome: Loading time of evaluation Web site per retrofitter, showing time in milliseconds*

we created five different CSPs, each of which triggering each, none or all retrofitters (see Fig. 6). To collect meaningful data about the loading time, we repeated the experiment 1,000 times in the browser.

Fig. 11 shows the results of the repeated loading time measurements of the evaluation Web site for each of the five constructed CSPs. The Web site's loading time was the shortest when configured with a CSP only triggering the *UnsafeHashesRetrofitter*. In this case, the average loading time ranged between *17* and *19ms*. This is quite surprising if comparing this with the results of the *empty* CSP. On average, these measurements were about *2ms* higher. However, since an empty CSP does not restrict any code execution, more code is executed before the page has finished loading, hence delaying the load time.

Considering the second CSP, we can see that it has a noticeable effect on the Web site's loading time. Although this CSP is restricting code execution in comparison to the empty CSP, it makes the Web site take about *20ms* longer to load. Notably, the second CSP contains many hash source expressions, which are used in combination with the `'unsafe-hashes'` keyword source expression. This causes higher work load for the *UnsafeHashesRetrofitter* because each inline event handler has to be compared with each valid hash in the CSP, causing synchronous hash computations.

In contrast to that, the third CSP, targeting the `navigate-to` directive, does not cause a meaningful increase in the loading time of the Web site. The loading time increased by *6ms* in comparison to the first CSP.

The last CSP, which is a merged version of the first three, seems to do exactly what is expected. The loading time of the Web site is defined by the combined loading time effects of the three individual CSPs. Most of it is dominated by the second CSP, triggering the `'unsafe-hashes'` feature. However, it only increased by about *2ms* in total. This originates from the restriction on execution of inline scripts due to CSP 3.

We conclude that the structure of the CSP has an impact on the overall loading time of the Web site. Depending on the CSP's structure, different retrofitters are triggered by design. Moreover, the individual amount of work for each retrofitter can be increased by the CSP. The retrofitting architecture introduces a loading time overhead by retrofitting previously unsupported CSP features. However, overall, the loading time overhead was limited to about *22-23ms*

*2) RETROCSP overhead:* Apart from this, we also analyzed the general overhead of the individual retrofitters. To assure that all measurements take place under the same conditions, we only used CSP #5 which triggers all retrofitters. We compared the results of the unmodified browser with five RETROCSP configurations by de-/activating individual retrofitters:

1) no retrofitter activated
2) only *StrictDynamicRetrofitter* activated
3) only *UnsafeHashesRetrofitter* activated
4) only *NavigateToRetrofitter* activated
5) all retrofitters activated

All measurements were taken in the same environment, allowing us to calculate the full overhead of each retrofitter. Furthermore, we could deduce the pure overhead introduced by the proxy functionality of the organizer. The overview of the results is shown Fig. 12.

When visiting the test Web site without RETROCSP deployed at all, we measured a loading time of *20-21ms*. If we compare this value with the loading time of the Web site when RETROCSP is fully activated, we can compute a difference of about *21-23ms* on average. This difference yields the overhead of the entire retrofitting architecture, both the *organizer* as well as all three *retrofitters*.

Although this value might suggest a runtime overhead of 100%, several facts must be considered. First of all, the evaluation Web page is very small. Its size is only *1.98KB*. The Web site does not include any external resources, no images, nor does it frame any other domains. Furthermore, the server hosting the Web site was running on the same machine and served the Web page via `localhost`. As such, it is almost impossible to receive a server response in a faster way. This is also visible in the difference of the Web site's loading time when configured with the first and fourth CSP of Fig. 11. Even small code execution differences have a visible impact on the total loading time. The average loading time of Web pages is about *10.3s* [13]. Conclusively, a loading time increase of *21-23ms* on average yields a total runtime overhead of about 0.2%. By design, the work performed by the retrofitters is constant and completely independent of the network. Hence, we argue that an overhead equivalent to serving *1.98KB* of data via `localhost` can be considered *negligible*.

To determine the organizer's effect on the architecture's runtime, we compare the overall loading time effect of RETROCSP with the measurement results when *no* retrofitter was activated. As we can see, the runtime of the *organizer* seems to be minimal as the results seem to be equivalent to those of the vanilla browser. The organizer does not seem to have a perceptible effect on the runtime of the retrofitting architecture. We deduce that the proxy functionality of RETROCSP does not affect the user experience at all. Hence, the *proxy* phase of each retrofitter only has a small effect on the architecture's runtime as well.

The *StrictDynamicRetrofitter* did not increase the loading time meaningfully. It only raised the average loading time by about *1ms*. This is not very surprising, as the *StrictDynamicRetrofitter* only hooks two methods during its *client* phase. As such, the first retrofitter seems to account for 4-5% of RETROCSP's overhead.

On the other hand, the *UnsafeHashesRetrofitter* might dominate the runtime of the entire architecture. In comparison to the vanilla browser, it raised the overall loading time of the Web site by around *19-20ms*. This is equal to 85-90% of the architecture's loading time overhead. The second retrofitter is responsible for categorizing code based on its hash value. The synchronous hash computations might be the cause for the dominating effect on the architecture's runtime.

The last retrofitter, the *NavigateToRetrofitter*, only increased the average loading time by a small factor. We measured an average raise of *2-5ms* which is way less than the effect of the previous retrofitter. Thus, the *NavigateToRetrofitter* represents about 10% of the entire architecture's runtime. This is twice as much as the *StrictDynamicRetrofitter*. While this retrofitter also hooks a DOM method, it also installs a Mutation Observer, which accounts for the remaining overhead.

As discussed, the retrofitting architecture's loading time overhead is dominated by the *client* phase of each retrofitter. In particular, the *UnsafeHashesRetrofitter* has the most observable impact on RETROCSP's runtime.

The second performance measurements could verify the results of analyzing the impact of the CSP structure. The runtime seems to be directly defined by the triggered retrofitting module. The overhead of each retrofitter was equivalent to the overhead provided by the corresponding CSP.

All in all, the retrofitting architecture at most raised the loading time of the Web site by about *22ms*. As mentioned before, we consider this to be a negligible overhead that cannot outweigh the improved CSP support.