# CCSP: Controlled Relaxation of Content Security Policies by Runtime Policy Composition

Stefano Calzavara, Alvise Rabitti and Michele Bugliesi
*Università Ca' Foscari Venezia*

## Abstract

Content Security Policy (CSP) is a W3C standard designed to prevent and mitigate the impact of content injection vulnerabilities on websites by means of browser-enforced security policies. Though CSP is gaining a lot of popularity in the wild, previous research questioned one of its key design choices, namely the use of static white-lists to define legitimate content inclusions. In this paper we present Compositional CSP (CCSP), an extension of CSP based on runtime policy composition. CCSP is designed to overcome the limitations arising from the use of static white-lists, while avoiding a major overhaul of CSP and the logic underlying policy writing. We perform an extensive evaluation of the design of CCSP by focusing on the general security guarantees it provides, its backward compatibility and its deployment cost. We then assess the potential impact of CCSP on the web and we implement a prototype of our proposal, which we test on major websites. In the end, we conclude that the deployment of CCSP can be done with limited efforts and would lead to significant benefits for the large majority of the websites.

## 1 Introduction

Content Security Policy (CSP) is a W3C standard introduced to prevent and mitigate the impact of content injection vulnerabilities on websites [11]. It is currently supported by all modern commercial web browsers and deployed on a number of popular websites, which justified a recently growing interest by the research community [20, 5, 13, 1, 18, 10].

A content security policy is a list of directives supplied in the HTTP headers of a web page, specifying browser-enforced restrictions on content inclusion. Roughly, the directives associate different content types to lists of sources (web origins) from which the CSP-protected web page can load contents of that specific type. For instance, the following policy:

```
script-src  https://example.com;
img-src     *;
default-src 'none'
```

specifies these restrictions: scripts can only be loaded from `https://example.com`, images can be loaded from any web origin, and contents of different type, e.g., stylesheets, cannot be included. Moreover, CSP prevents by default the execution of inline scripts and bans a few dangerous JavaScript functions, like `eval`; these restrictions can be explicitly deactivated by policy writers to simplify deployment, although they are critical for the security of CSP.

Simple as it looks, however, CSP is typically hard to deploy correctly on real websites [19, 20, 1, 18] and there are several, diverse reasons for this:

1. an effective content security policy must not relax the default restrictions which forbid inline scripts and `eval`-like functions. However, removing inline scripts from existing websites proved to require a significant effort [19, 20], hence relaxing the default restrictions of CSP is a common routine even for major websites [18, 1];

2. white-lists are hard to get right. On the one hand, if a white-list is too liberal, it can open the way to security breaches by allowing the communication with JSONP endpoints or the inclusion of libraries for symbolic execution, which would enable the injection of arbitrary malicious scripts [18]. On the other hand, if a white-list is too restrictive, it can break the intended functionality of the protected web page [1]. The right equilibrium is difficult to achieve, most notably because it is hard for policy writers to predict what needs to be included by active contents (scripts, stylesheets, etc.) loaded by their web pages;

3. many web contents have a dynamic nature, which is not easily accommodated by means of the static white-lists available in CSP. For instance, writing appropriate content security policies may be hard when using CDNs for load balancing, when including advertisement libraries based on real-time bidding, or in presence of HTTP redirects [20, 1].

The industry was relatively quick in realizing that the pervasive presence of inline scripts is a serious obstacle to the widespread adoption of CSP and more recent versions of the standard introduced controlled mechanisms based on hashes and nonces to white-list individual inline scripts [15]. This is an interesting approach to deal with the first problem we mentioned: by selectively enabling only a few known inline scripts, web developers can significantly improve the security of their websites against script injection, while avoiding a major overhaul of their code base by moving inline scripts to external files.

However, the other two problems which hinder a wider and more effective deployment of CSP are still largely unsolved, since they both stem from the inherent complexity of accurately predicting the capabilities dynamically needed by real, content-rich web applications. The recent *Strict* CSP proposal [4] based on CSP Level 3 [16] may help in dealing with these challenges in some practical cases, but unfortunately it only provides a very partial solution to them (see Section 2 for a discussion).

## 1.1 Goals and Contributions

The goal of the present paper is proposing a simple extension of CSP which naturally and elegantly solves the delicate issues discussed above. The most important design goal of our proposal is to avoid both a major overhaul of the existing CSP specification and a dramatic change to the logic behind policy writing, so as to simplify its practical adoption by web developers who are already familiar with CSP.

Our proposal builds on the pragmatic observation that static white-lists are inherently complex to write down for modern web applications and do not really fit the dynamic nature of common web interactions, so we devise Compositional CSP (CCSP), an extension of CSP based on *runtime policy composition*. In CCSP an initial, simple content security policy is incrementally relaxed via the interactions between the protected page and its content providers. More precisely, content providers can loosen up the policy of the protected page to accommodate behaviours which were not originally admitted by the latter, although the protected page reserves itself the last word on its security by specifying suitable upper bounds for policy relaxation. Notably, by introducing a dynamic dimension to CSP and by delegating part of the policy specification efforts to the content providers, it is possible to come up with white-lists which are much more precise than those which could be realistically written by the developers of the protected page alone, since they often lack an in-depth understanding of the externally included contents and their dependencies.

Concretely, we make the following contributions:

1. we provide a precise specification of CCSP and we discuss two realistic use cases which may benefit from this extension of the CSP standard. We show CCSP policies for these use cases and we discuss how the enforcement model of CCSP supports desirable security guarantees (Section 3);

2. we perform an extensive evaluation of the design of CCSP by focusing on the general security guarantees it provides, its backward compatibility and its deployment cost (Section 4);

3. we assess the potential impact of CCSP in the wild by building and analysing a dataset of CSP violations found on the Alexa Top 100k websites. Specifically, we show that violations are pervasive and mostly caused by behaviours which are hard to accommodate in CSP, which confirms the need for a more expressive mechanism like CCSP. We also establish that only a few selected players need to embrace CCSP to provide a significant benefit to the majority of the websites (Section 5);

4. we develop a proof-of-concept implementation of CCSP as a Chromium extension and we test it by manually writing CCSP policies for a few popular websites which normally trigger CSP violations. Our experiments show that CCSP is easy to deploy and fully supports the intended functionality of the tested websites (Section 6).

## 2 Motivations

## 2.1 Example

Consider a web page *w* including the following script tag in its HTML contents:

```
<script src="https://a.com/stats.js"/>
```

The `stats.js` script, in turn, is implemented as follows:

```
// load dependency.js from https://b.com
var s = document.createElement('script');
s.src = 'https://b.com/dependency.js';
document.head.appendChild(s);
...
// load banner.jpg from https://c.com
var i = document.createElement('img');
i.src = 'https://c.com/banner.jpg';
document.body.appendChild(i);
```

This script includes another script from `https://b.com` and an image from `https://c.com` and all these contents must be allowed by the content security policy of *w* to let the web page work correctly.

**CSP 1.0 and CSP Level 2.** Both CSP 1.0 [14] and CSP Level 2 [15] restrict the inclusion of external contents purely by means of a white-listing approach. This means that the content security policy of *w* must not only white-list `https://a.com` as a valid source for script inclusion to load `stats.js`, but it must also white-list all the dependencies of `stats.js` with their correct type. An appropriate content security policy for *w* would thus be:

```
script-src https://a.com https://b.com;
img-src    https://c.com
```

This approach has two significant problems. First, the definition of the policy is complex, since it requires one to carefully identify all the dependencies of the scripts included by *w*, including those recursively loaded, like `dependency.js`. Second, the policy above is brittle: if the developers of `stats.js` or `dependency.js` change the implementation of their scripts, for instance to include additional libraries from other sources, the policy of *w* must correspondingly be updated to white-list them.

In principle, these limitations pay off in terms of security, since the developers of *w* have full control on which contents can be included by scripts loaded by their page. Unfortunately, previous research showed that web developers typically write an overly liberal white-list to avoid issues with their websites, for instance by allowing the inclusion of scripts from any domain [20, 1, 18].

**CSP Level 3 (Strict CSP).** Recently, the CSP community realized that the white-list approach advocated by CSP 1.0 and CSP Level 2 is often inconvenient to use, because it may be hard to write white-lists which are neither too liberal, nor too restrictive. The latest version of the standard, called CSP Level 3 [16], thus added new mechanisms which support a different policy specification style, known as *Strict* CSP [4]. Strict CSP drives away from the complexities of white-lists and simplifies the process of recursive script inclusion by transitively propagating trust. Concretely, the example web page *w* must be changed to bind a randomly generated *nonce* to its script tag as follows:

```
<script src="https://a.com/stats.js"
        nonce="ab3f5k"/>
```

Correspondingly, its content security policy is adapted as follows:

```
script-src 'nonce-ab3f5k' 'strict-dynamic';
img-src     https://c.com
```

Under this policy, only those scripts whose tag includes the nonce `ab3f5k` are allowed to be loaded, irrespective of the web origin where they are hosted. Since the nonce value is random and unpredictable, an attacker cannot inject malicious scripts with a valid nonce on *w*. Since nonce-checking may break benign scripts which are dynamically inserted without a valid nonce, the `'strict-dynamic'` source expression is included in the policy: this ensures that any script request triggered by a non-parser-inserted script element like `stats.js` is also allowed by CSP [16].

Besides making policy specification simpler, nonces also make the resulting policies stronger against script injection attacks. Since a valid nonce is only bound to the script tag loading `https://a.com/stats.js`, other dangerous contents hosted on `https://a.com` cannot be unexpectedly loaded and abused by an attacker. Moreover, the use of `'strict-dynamic'` simplifies the definition of policies when recursive script inclusion is needed. In our example, the script `dependency.js` can be relocated from `https://b.com` to `https://d.com` without any need of updating the content security policy of *w*. Also, both `stats.js` and `dependency.js` can include new libraries without violating the policy of *w*, again thanks to the presence of `'strict-dynamic'`.

## 2.2 Criticisms to Strict CSP

**Limited Scope.** The `'strict-dynamic'` source expression only targets the problem of recursive script inclusion, but it does not solve similar issues for other content types. In our example, if the script `stats.js` is updated to load its image from `https://d.com` rather than from `https://c.com`, or if the script is left unchanged but the image is relocated to `https://d.com` by means of a HTTP redirect, the content security policy of *w* must also be updated to ensure a correct rendering of the website. This means that the developers of *w* must change their content security policy to deal with something which is unpredictable and not under their control.

Generalizing `'strict-dynamic'` to deal with these cases would have a negative "snowball effect" on the effectiveness of CSP, since basically all the content restrictions put in place by the policy would need to be ignored.

**Poor Granularity.** The `'strict-dynamic'` source expression uniformly applies to all the scripts loaded by a web page, thus offering an all-or-nothing relaxation mechanism. We believe there are two sound motivations underlying this design: ease of policy specification and the fact that the main security goal of Strict CSP is protecting against XSS. The key observation justifying the design of `'strict-dynamic'` is that, if a script loaded by a web page is malicious, it can already attack the page, without any need of abusing `'strict-dynamic'` to load other malicious scripts from external domains.

However, it is worth noticing that the design of `'strict-dynamic'` does not support the *principle of least privilege*, since it gives all scripts with a valid nonce the ability of putting arbitrary relaxations on script inclusion, even though not every script requires this capability. As a matter of fact, most scripts included in a trusted website are not actually malicious, but they may have bugs or be developed by programmers who are not security experts, and there is no way for a benign script to declare that it only needs limited policy relaxation capabilities to perform its intended functionalities, thus limiting the room for unintended harmful behaviours.

**Use of Nonces.** Nonces are a convenient specification tool, but they also have severe drawbacks. First, the security of nonces is questionable: it is now recognized that the protection offered by their use can be sidestepped, most notably because nonces are still included in the DOM. This leaves room for attacks, for instance when script injection happens in the scope of a valid nonce[1] or when nonces are exfiltrated by means of *scriptless* attacks[2]; also other attacks have been found recently[3].

Moreover, the use of nonces makes the security review of a page deploying CSP much harder to carry out: one cannot just inspect the content security policy of the page to understand which security restrictions are put in place, but she must also check the code of the page to detect which scripts are bound to a valid nonce. (In fact, the use of nonces makes particularly troublesome to compare the permissiveness of two content security policies, which instead is a fundamental building block of the upcoming CSP Embedded Enforcement mechanism [17].) Finally, nonces are not easily assigned to dynamically generated script tags in legacy code: `'strict-dynamic'` is just one way to circumvent this issue, but it comes with the limitations we previously discussed.

**Discussion.** Strict CSP provides significant improvements over CSP 1.0 and CSP Level 2 in terms of both security and ease of deployment, and there is pragmatic evidence about the effectiveness of `'strict-dynamic'` at major websites [18]. Nevertheless, we discussed practical cases where `'strict-dynamic'` is not expressive enough to fix CSP violations and web developers still need to account for these behaviors by means of extensive white-listing. This complicates policy specification and maintenance, because policy changes may be dictated by elements which are not under the direct control of the policy writers, such as script dependencies and HTTP redirects. We confirm the existence of these expressiveness issues of Strict CSP in the wild in Section 5.

CCSP complements Strict CSP with more flexible tools for policy specification, while supporting the principle of least privilege and removing security-relevant information from the page body, thus simplifying policy auditing and preventing subtle security bypasses. Moreover, its design is backward compatible to ensure a seamless integration with the existing CSP deployment.

## 3 Compositional CSP (CCSP)

### 3.1 Overview

In our view, web developers should be able to keep their content security policies as simple as possible by focusing only on the direct dependencies required by their web pages, largely ignoring other dependencies needed by the contents they include. In CCSP, direct dependencies are specified by means of *fine-grained white-lists* reminiscent of CSP 1.0 and CSP Level 2, since these dependencies are relatively easy to identify for web developers and it is feasible to come up with reasonably strict and secure white-lists for them. Indirect dependencies, instead, are dealt with by *dynamically composing* the policy of the protected web page with additional content security policies which define the dependencies of the included contents. These policies are written and served by the content providers, who are the only ones who can accurately know the dependencies of the contents they serve and keep them constantly updated. Ideally, only the least set of dependencies required to work correctly should be white-listed to respect the principle of least privilege and avoid weakening protection unnecessarily. To keep under control the power on policy specification delegated to external content providers, CCSP grants web developers the ability of putting additional restrictions on the policy relaxation mechanism.

Concretely, let us move back to our example. In our proposal, the web page $w$ would send to the browser the following headers:

```
CSP-Compose
   script-src https://a.com/stats.js;

CSP-Intersect
   scope       https://a.com/stats.js;
   script-src  https://*;
   img-src     *;
   default-src 'none'
```

The `CSP-Compose` header contains the initial content security policy of the protected page: in this case, it specifies that only the script `https://a.com/stats.js` can

[1] http://blog.innerht.ml/csp-2015/
[2] http://sirdarckcat.blogspot.com/2016/12/how-to-bypass-csp-nonces-with-dom-xss.html
[3] http://sebastian-lekies.de/csp/bypasses.php

be included in the page. The `CSP-Intersect` header, instead, tracks that the script `https://a.com/stats.js` is entitled to relax the content security policy of *w* up to the specified upper bound, expressed again in terms of CSP. For instance, in this case the script can relax the content security policy of the protected page to white-list any HTTPS script and any image, but nothing more. Different scripts can be assigned different upper bounds on policy relaxation.

When delivering `stats.js`, the script provider can attach it the following header:

```
CSP-Union
   script-src https://b.com/dependency.js;
   img-src    https://c.com
```

The `CSP-Union` header includes what `stats.js` needs to operate correctly. In this case, the additional script dependency is white-listed very precisely, while there is much more liberality on images, since any image from `https://c.com` is white-listed by the policy.

A CCSP-compliant web browser would join together the original policy of the page and the policy supplied by the script provider before including `stats.js`, while enforcing the upper bounds on policy relaxation specified by the developers of the protected page. In this case, the policy supplied by the script provider is compliant with said upper bounds, hence the browser would update the content security policy of the page as follows:

```
script-src https://a.com/stats.js
           https://b.com/dependency.js;
img-src    https://c.com
```

This policy is reminiscent of the policy we would write using CSP 1.0 or CSP Level 2, but it is assembled dynamically by interacting with the content providers. This significantly simplifies the specification of the original policy for the page developers and makes it naturally robust to changes in the included contents, as long as the capabilities required by the updated contents still comply with the original upper bounds on policy relaxation specified by the page developers. This flexibility is crucial to appropriately deal with highly dynamic web contents, which can hardly be accommodated by static white-lists, and with complex chains of dependencies, which may be difficult to predict for page developers.

It is also worth noticing that, since the burden of policy specification is now split between page developers and content providers, CCSP makes it feasible in practice to white-list individual contents rather than entire domains, which makes the resulting policy stricter and more secure. In the end, the resulting policy can realistically be as strict and precise as a nonce-based policy, but all the security information is available in the HTTP headers, thus overcoming the typical limitations associated with the use of nonces. Finally, observe that the dynamically enforced policy is much tighter than the upper bound for policy relaxation specified by the protected web page. Though the page developers could deploy a standard CSP policy which is as liberal as the upper bound, that policy would be significantly more permissive than the enforced CCSP policy built at runtime.

## 3.2 Example Use Cases

We discussed how our proposal overcomes some important limitations of CSP, but we now observe that these improvements come with a cost on the content providers, which in our proposal become actively involved in the policy specification process. We believe that many major content providers would be happy to contribute to this process, because mismatches between the capabilities required by their contents and the content security policies of their customers may lead to functionality issues resulting in economic losses, like in the case of broken advertisement. To further exemplify the benefits of CCSP, however, we discuss now two concrete use cases.

As a first use case, we pick a provider of JavaScript APIs, for example Facebook. The lead developer of the Facebook APIs may stipulate that all the developers in her team are allowed to use external libraries in the scripts they write, but only if they are hosted on `https://connect.facebook.net`, because libraries which are put there are always updated, subject to a careful security scrutiny and delivered over a secure channel. The lead developer can thus ensure that the following header is attached to all the available JavaScript APIs:

```
CSP-Union
   script-src https://connect.facebook.net
```

This way, Facebook can make its customers aware of the fact that the API code only needs to access internal contents to operate correctly, which may increase its level of trust and simplify a security auditing. If a Facebook API contained a bug or was developed by an uncaring developer who did not respect the indications of the lead developer, a sufficiently strong content security policy on the pages using the API may still prevent the unintended inclusion of dangerous contents.

As a second use case, we consider an advertisement library. Web advertisement may involve delicate trust relationships: it is not uncommon for web trackers to collaborate and share information about their visitors[4]. For instance, an advertisement library developed by `a.com` may actually import an external advertisement library by `b.com`. Developers at `a.com` may want to mitigate the impact of vulnerabilities in the `b.com` library, since the

[4]`https://blog.simeonov.com/2013/04/17/anatomy-of-an-online-ad/`

end user of the advertisement library may be unaware of the inclusion of external contents and just blame the developers of `a.com` for any security issue introduced by the advertisement system. The `a.com` developers can thus attach the following headers to their library:

```
CSP-Union
   script-src https://b.com/adv.js;

CSP-Intersect
   scope        https://b.com/adv.js;
   img-src      *://b.com;
   default-src 'none'
```

This way, the developers at `a.com` declare their need of including a script by `b.com`, but they only grant it enough capabilities to relax the content security policy of the embedding page to white-list more images from its own domain (using any protocol) and nothing more. This significantly reduces the impact of bugs in the script by `b.com`, as long as the page using the advertisement library deploys a reasonably strong content security policy in the first place.

## 3.3 Specification

**Preliminaries.** We start by reviewing some terminology from the original CSP specification. A content security policy is a list of *directives*, defining content restrictions on *protected resources* (web pages or iframes) by means of a white-listing mechanism. White-lists are defined by binding different content types (images, scripts, etc.) to lists of *source expressions*, noted as $\vec{se}$, which are a sort of regular expressions used to succinctly express sets of URLs. The inclusion of a content of type $t$ from a protected resource $r$ is only allowed if the URL $u$ of the content *matches* any of the source expressions bound to the type $t$ in the content security policy of $r$. We abstract from the details of the matching algorithm of CSP and we just write $matches(u, \vec{se})$ if $u$ matches any of the source expressions in the list $\vec{se}$.

We let $P$ stand for the set of the content security policies and we let $p$ range over it. We let $\sqsubseteq$ stand for the binary relation between content security policies such that $p_1 \sqsubseteq p_2$ if and only if all the content inclusions allowed by $p_1$ are also allowed by $p_2$. It can be proved that $(P, \sqsubseteq)$ is a *bounded lattice* and there exist algorithmic ways to compute the *join* $\sqcup$ and the *meet* $\sqcap$ of any two content security policies. The join $\sqcup$ allows a content inclusion if and only if it is allowed by at least one of the two policies (union of the rights), while the meet $\sqcap$ allows a content inclusion if and only if it is allowed by both policies (intersection of the rights). We let $\top$ and $\bot$ stand for the top and the bottom elements of the lattice respectively. In the following, we do not discuss how the join and the meet of two policies are actually computed, but we provide an abstract specification of CCSP which uses these operations as a black box. The formal metatheory is presented in Appendix A for the sake of completeness.

**Security Headers.** The CCSP specification is based on three new security headers:

1. `CSP-Compose`: only used by the web developers of the protected resource. It includes a content security policy specifying the initial content restrictions to be applied to the protected resource;

2. `CSP-Union`: only used by the content providers. It includes a content security policy which should be joined with the content security policy of the protected resource to accommodate the intended functionality of the supplied contents;

3. `CSP-Intersect`: (optionally) used by both the web developers of the protected resource and the content providers. It includes a list of bindings between a source expression list (a *scope*) and a content security policy, expressing that contents retrieved from a URL matching a given scope are entitled to relax the policy of the protected resource only up to the specified upper bound.

The next paragraph makes these intuitions more precise.

**Enforcement Model.** Conceptually, each protected resource needs to keep track of two elements: the enforced content security policy $p$ and the upper bounds on policy relaxation $R = \{(\vec{se}_1, p_1), \ldots, (\vec{se}_n, p_n)\}$ collected via the `CSP-Intersect` headers. We call $R$ a *relaxation policy* and we refer to the pair $(p, R)$ as the *protection state* of the protected resource.

In the protection state $(p, R)$, a content inclusion is allowed if and only if it is allowed by the content security policy $p$, whose weakening is subject to the relaxation policy $R$. Initially, the protection state is set so that $p$ is the policy delivered with the `CSP-Compose` header of the protected resource and $R = \{(\vec{se}_1, p_1), \ldots, (\vec{se}_n, p_n)\}$ is the `CSP-Intersect` header originally attached to it. The protection state can be dynamically updated when the protected resource includes contents with a `CSP-Union` header. To formalise the update of the protection state, it is convenient to introduce a few auxiliary definitions. First, we define the set of the upper bounds for policy relaxation given to the URL $u$ by the relaxation policy $R$.

**Definition 1** (Specified Upper Bounds)**.** Given a URL $u$ and a relaxation policy $R$, we define the *specified upper bounds* for $u$ under $R$ as:

$$bnds(u, R) = \{p \mid \exists \vec{se} : (\vec{se}, p) \in R \wedge matches(u, \vec{se})\}.$$

Using this auxiliary definition, we can define the upper bound for policy relaxation as follows:

**Definition 2** (Upper Bound)**.** Given a URL $u$ and a relaxation policy $R$, we define the *upper bound* for $u$ under $R$ as:

$$ub(u,R) = \begin{cases} p_1 \sqcup \ldots \sqcup p_n & \text{if } bnds(u,R) = \{p_1,\ldots,p_n\} \\ & \quad \text{with } n > 0 \\ \bot & \text{if } bnds(u,R) = \emptyset \end{cases}$$

Though simple, this definition is subtle. If no upper bound for a given URL $u$ is defined in a relaxation policy $R$, then $ub(u,R) = \bot$ and no policy relaxation is possible when processing a response from $u$. However, if multiple upper bounds are specified for $u$, then their *join* is returned. This means that, if multiple content providers specify different, possibly conflicting upper bounds on policy relaxation, then all of them will be honored, which is the most liberal behaviour. This design choice is crucial to ensure the functionality of the protected resource, as we extensively discuss in Section 4.

We are now ready to explain how the protection state of a resource gets updated. Let $(p,R)$ be the current protection state and assume that a content is loaded from the URL $u$. Assume also that the corresponding HTTP response attaches the following headers: a `CSP-Union` header including the content security policy $p'$ and a `CSP-Intersect` header defining the relaxation policy $R'$. Then, the protection state $(p,R)$ is updated to:

$$\begin{aligned} &(p \sqcup (p' \sqcap ub(u,R)), \\ &\quad R \cup \{(\vec{se}_i, p_i \sqcap ub(u,R)) \mid (\vec{se}_i, p_i) \in R'\}). \end{aligned} \tag{1}$$

In words, the protection state of the protected resource is updated as follows:

1. the content security policy $p$ is relaxed to allow all the content inclusions allowed by $p'$ which are compatible with the restrictions $ub(u,R)$ enforced on $u$ by the relaxation policy $R$. This means that all the content inclusions allowed by $p'$ are also allowed to the protected resource, unless the relaxation policy $R$ specifies a tighter upper bound for $u$;

2. the relaxation policy $R$ is extended to allow all the behaviours allowed by $R'$ which are compatible with the restrictions $ub(u,R)$ enforced on $u$ by the relaxation policy $R$. This prevents trivial bypasses of the relaxation policy $R$, where $u$ specifies a more liberal upper bound than $ub(u,R)$ for other contents recursively loaded by itself.

Observe that CCSP gives web developers the possibility of granting different capabilities on policy relaxation to different content providers, but content security policies are still enforced per-resource (web page or iframe), rather than per-content. Though certainly useful in principle, enforcing different content security policies on different contents is not possible without deep browser changes, whose practical feasibility and backward compatibility are unclear.

## 3.4 Example

To exemplify the enforcement model of CCSP, we show our proposal at work on the advertisement library example of Section 2. Recall the example focuses on a library developed by `a.com` and importing an external library from `b.com`. Since the users of the `a.com` library are not necessarily aware of the inclusion of contents from `b.com`, the developers at `a.com` are careful in limiting the capabilities granted to the imported library. In particular, they deploy the following CCSP policy declaring the need of importing a script from `b.com`, which in turn should only be allowed to load images from the same domain, using any protocol:

```
CSP-Union
   script-src https://b.com/adv.js;

CSP-Intersect
   scope       https://b.com/adv.js;
   img-src     *://b.com;
   default-src 'none'
```

A user of the `a.com` library may not know exactly what the library needs to work correctly. However, since she trusts the provider of the library, she may deploy the following CCSP policy on her homepage:

```
CSP-Compose
   script-src https://a.com/lib.js;

CSP-Intersect
   scope       https://a.com/lib.js;
   script-src  https://*;
   img-src     https://*;
   default-src 'none'
```

Hence, in the initial protection state, the page is only allowed to load the `a.com` library, but the library is also granted the capability of relaxing the content security policy of the page to include more scripts and images over HTTPS. After loading the `a.com` library and processing its CCSP headers, the content security policy of the protected page is updated as follows:

```
script-src https://a.com/lib.js
           https://b.com/adv.js;
```

This allows the inclusion of the external script from `b.com`. What is more interesting, however, is how the relaxation policy of the homepage is updated after processing the response from `a.com`. Specifically, the relaxation policy will include a new entry for `b.com` of the following format:

```
scope       https://b.com/adv.js;
img-src     https://b.com;
default-src 'none'
```

This entry models the combined requirement that the imported script from `b.com` can only relax the content security policy of the protected page to load images from its own domain (as desired by `a.com`), but only using the HTTPS protocol (as desired by the protected resource and originally enforced on the importer at `a.com`).

Assume now that the script from `b.com` sends the following CCSP header:

```
CSP-Union
   img-src *
```

When processing the response from `b.com`, the page will further relax its content security policy as follows:

```
script-src https://a.com/lib.js
           https://b.com/adv.js;
img-src    https://b.com
```

Hence, even though `b.com` asked for the ability of loading arbitrary images from the web, the restrictions put in place by `a.com` actually ensured that the content security policy of the protected page was only relaxed to load images from `b.com`. At the same time, the protected page successfully enforced that these images are only loaded over HTTPS.

## 4 Design Evaluation

### 4.1 Security Analysis

**Threat Model.** CCSP is designed to assist *honest* content providers in making their end users aware of the capabilities needed by the contents they make available and simplify their robust integration with the content security policy of the embedding resource. As such, CCSP aims at mitigating the impact of *accidental* security vulnerabilities, whose threats can be prevented by the mismatch between the unintended harmful behaviours and the expected capabilities requested in the CCSP headers. If we assume that both the initial content security policy of the protected resource and the following relaxations (by honest content providers) comply with the principle of the least privilege, imported contents can only recursively load additional contents served from sources which were white-listed to implement a necessary functionality: this greatly reduces the room for dangerous behaviours.

However, CCSP is *not* designed to protect against malicious content providers. If a protected resource imports malicious contents, the current CSP standard offers little to no protection against data exfiltration and serious integrity threats [13]. Since CCSP ultimately relies on CSP to implement protection, the same limitations apply to it, though attackers who are not aware of the deployment of (C)CSP on the protected resource may see their attacks thwarted by the security policy.

**Policy Upper Bounds.** In CCSP, the initial protection state $(p, R)$ is entirely controlled by the developers of the protected resource. If $R = \emptyset$, no policy relaxation is possible and the security guarantees offered to the protected resource are simply those provided by the initial content security policy $p$. Observe that no policy relaxation is allowed even if a content provider at $u$ sends its own relaxation policy $R' \neq \emptyset$, since all the relaxation bounds in $R'$ will be set to $ub(u, R) = \bot$ when updating the protection state, thanks to the use of the meet operator in Equation 1. Otherwise, let $R = \{(\vec{se}_1, p_1), \ldots, (\vec{se}_n, p_n)\}$ with $n > 0$ be the initial relaxation policy. In this case, the most liberal content security policy eventually enforced on the protected resource can be $p \sqcup p_1 \sqcup \ldots \sqcup p_n$, again because the initial upper bounds on policy relaxation can never be weakened when the protection state is updated, due to the use of the meet operator in Equation 1. Remarkably, this bound implies that the developers of the protected resource still have control over the most liberal content security policy enforced on it and may reliably use CCSP to rule out undesired behaviours, e.g., loading images over HTTP, just by writing an appropriate initial policy and upper bounds on policy relaxation.

Notice that the upper bounds defined by the initial relaxation policy may be way more permissive than the actual policy enforced on the protected resource, since the policy relaxation process happens dynamically and depends on the responses of the different content providers. In particular, if all the content providers are honest and prudent, they should comply with the principle of the least privilege, hence the enforced policy will realistically be much tighter than the original upper bounds.

### 4.2 Compatibility Analysis

**Legacy Browsers.** Legacy browsers lacking support for CCSP will not recognise the new security headers defined by our proposal, hence these headers will just be ignored by them. If we stipulate that CCSP-compliant browsers should only enforce standard content security policies in absence of CCSP policies, which is a reasonable requirement being CCSP an extension of CSP, developers of protected resources can provide support for legacy browsers just by sending both a CCSP policy (enforced by CCSP-compliant browsers) and a standard content security policy (enforced by legacy browsers).

Clearly, the latter policy would need to white-list all the legitimate content inclusions, though, as we said, these are often hard to predict correctly. Luckily, there is a simple way to build a working content security policy from a CCSP policy, which is including all the upper bounds specified by the relaxation policy directly in the content security policy. This can be done automatically by a server-side proxy and it will produce a policy which is typically more liberal than necessary, yet permissive enough to make the protected resource work correctly (and not necessarily weaker than the policy the average web developer would realistically write using CSP).

**Legacy Content Providers.** Legacy content providers will not attach any `CSP-Union` header to the contents they serve, although developers of resources protected by CCSP may expect them to supply this information to relax their policies. There are two alternative ways to deal with the absence of a `CSP-Union` header, both of which are plausible and worth discussing:

1. perform no policy relaxation: this conservative behaviour can break the functionality of protected resources, but it ensures that, whenever policy relaxation happens, both the developers of the protected resource and the content providers agreed on the need of performing such a sensitive operation;

2. relax the policy to the upper bound specified for the content provider: this alternative choice privileges a correct rendering of contents served by legacy content providers, at the cost of enforcing a policy which may be more permissive than necessary. Notice, however, that this still takes into account the (worst case) expectations of the developers of the protected resource.

Though both choices are sensible, we slightly prefer the first option as the default in CCSP, most notably because it is consistent with a similar design choice taken in the latest draft of CSP Embedded Enforcement [17], where the lack of an expected header on an embedded content triggers a security violation on the embedding resource. We do not exclude, however, that it could be useful to extend CCSP to give developers a way to express which of these two choices should be privileged.

**Compatibility Issues from Security Enforcement.** In CCSP, the content security policy of the protected resource can never be restricted by an interaction with a content provider, but it can only be made more liberal, hence it can never happen that a content provider forbids a content inclusion which is needed and allowed by the protected resource. It is also important to remark that different content providers can specify different, possibly conflicting upper bounds for policy relaxation with the same scope, but conflicts cannot lead to compatibility issues in practice, because all bounds are joined together (see Definition 2) and taken into account upon policy relaxation. This choice privileges the correct rendering of contents over security, but the opposite choice of taking the meet of the upper bounds would make the integration of contents from different providers too difficult to be practical, because these providers are not necessarily aware of the presence of each other in the same protected resource and may disagree on the relaxation needs. Moreover, taking the meet of the upper bounds would open the room to "denial of service" scenarios, where two competitor content providers could maliciously put unduly restrictions on each other.

If the content security policy of the protected resource is not liberal enough to let a content be rendered correctly, there are only two possibilities:

1. the original content security policy of the protected resource was not permissive enough in the first place and was never appropriately relaxed;

2. the content was loaded by a provider enforcing overly tight restrictions on policy relaxation for contents recursively loaded by another provider.

The first possibility may already occur in CSP and it is inherent to the nature of any whitelist-based protection mechanism. The second possibility, instead, is specific to CCSP, but it is not really a compatibility issue, because providers are not forced to put restrictions on policy relaxation and they are assumed to behave rationally, i.e., they do not deliberately put restrictions to break contents which are recursively loaded from other providers as part of their intended functionality.

## 4.3 Deployment Considerations

**Deployment on Websites.** Two actors must comply with CCSP to benefit of its protection: developers of protected resources and content providers. Assuming a reasonably large deployment of CCSP by content providers, developers who are willing to deploy a standard content security policy on their websites would have a much simpler life if they decided to run CCSP instead, because the policies written in the `CSP-Compose` headers are a subset of the policies which would need to be written using the standard CSP; moreover, the direct dependencies of the protected resource are much simpler to identify than the indirect ones. Writing accurate `CSP-Intersect` headers for controlled policy relaxation might be more complex for the average web developer, but quite liberal policies would be easy to write and still appropriate for content providers with a high level of trust.

Content providers, instead, would need to detect the (direct) dependencies of the contents they serve and write appropriate `CSP-Union` headers. We think this a much simpler task for them rather than for the end users of their contents, because they have a better understanding of their implementation. We also believe that pushing part of the policy specification effort on the content providers is beneficial to a wide deployment of CCSP, because in practice few selected providers supply a lot of contents to the large majority of the websites. Configuring correctly the `CSP-Union` headers of these providers would thus provide benefits to a significant fraction of the web, which is a much more sensible practice than hoping that most web developers are able to identify correctly the dependencies of the contents they include. We substantiate these claims with the experiments in Section 5.

**Deployment in Browsers.** CCSP does not advocate any major change to the CSP specification and uses it as a black box, because the content restrictions applied to a CCSP-protected page follow exactly the semantics of a standard content security policy. The only difference with respect to CSP is that the protection state of the protected resource is not static, but can change dynamically, so that different content security policies are applied on the same protected resource at different points in time. This means that CCSP should be rather easy to deploy on existing browsers, because the implementation of CSP available therein could be directly reused.

**Incremental Deployment.** Given that CCSP is an extension of CSP, it naturally supports the coexistence of CCSP-compliant and legacy content providers in the same policy. Developers of protected resources can write `CSP-Intersect` headers for CCSP-compliant providers and trust that they provide appropriate `CSP-Union` headers for their contents; at the same time, however, developers can also include the dependencies of legacy content providers directly in the `CSP-Compose` header. This allows an incremental deployment of CCSP on the web, which is particularly important because not all content providers may be willing to deploy CCSP.

### 4.4 Criticisms to CCSP

CCSP is more expressive than Strict CSP, because it extends the possibility of relaxing the white-listed content inclusions beyond what is allowed by the use of `'strict-dynamic'`. In this section, we argued for the security, the backward compatibility and the ease of deployment of CCSP. Still, there are a few potential criticisms to CCSP that we would like to discuss.

**Practical Adoption.** A first criticism to CCSP is fundamental to its design: achieving the benefits of CCSP requires adoption by third-party content providers. One may argue that it is difficult enough to get first parties to adopt CSP, let alone convince third parties to write CCSP policies. Two observations are in order here.

First, as anticipated, content providers typically have an economic interest on the correct integration between the contents they supply and the CSP policies of the embedding pages, such as in the case of advertisements, hence content providers often do not need further convincing arguments to deploy CCSP. Moreover, one may argue that the challenges faced by the first-party adoption of CSP may actually depend on the lack of third-party support for policy deployment, which proved to be difficult for web developers [9, 20, 1, 18]. If content providers could provide the correct policies for the content they supply, then also the first parties might be more willing to adopt CCSP, because they will encounter significantly less challenges upon deployment. Major content providers supporting CSP, such as Google, could play an important role in pushing the adoption of CCSP.

**Increased Complexity.** We acknowledge that CCSP is more complex than CSP and its enforcement model is subtle, because it aims at reconciling security, flexibility and backward compatibility. Complexity may be a concern for the practical adoption of CCSP, though one may argue that the simplicity of CSP bears limits of expressiveness which may actually complicate its deployment when `'strict-dynamic'` is not enough, e.g., in the presence of complex script dependencies or HTTP redirects.

That said, we designed CCSP as an extension of CSP exactly to ensure that web developers who do not need the additional expressive power of CCSP can ignore its increased complexity. On the contrary, web developers who need more flexibility in policy writing can find in CCSP useful tools to accommodate their needs.

**Complex Debugging.** A peculiarity of CCSP is that the enforced security policy changes dynamically, which can make policy debugging more complex than for CSP. This is a legitimate concern: even if content providers write appropriate `CSP-Union` headers for their resources, policy violations may arise due to some additional restrictions enforced by the `CSP-Intersect` headers sent by the protected resource.

We propose to make these conflicts apparent by extending the monitoring facilities of CCSP so that all the policy relaxations performed by a protected resource are reported to web developers. However, we acknowledge that designing a robust reporting system for CCSP is a complex and delicate problem, which we plan to investigate further as future work.

## 5 Impact of CCSP

To evaluate the benefits offered by CCSP, we built and extensively analyzed a dataset of CSP violations collected in the wild, finding a number of cases which are difficult to accommodate in CSP (and, indeed, were not correctly supported by policy writers). Our investigation confirms the need for a more expressive mechanism like CCSP. We then quantitatively assess that only few content providers need to deploy CCSP to fix most of the policy violations on the websites we visited, which substantiates the practical importance of our proposal.

### 5.1 Methodology

We developed a simple Chromium extension which intercepts the CSP headers of incoming HTTP(S) responses and changes them to report the detected CSP violations to a web server run by us (we do this by leveraging the `report-uri` directive available in CSP). We then used Selenium to guide Chromium into accessing the homepages of the 1,352 websites from the Alexa Top 100k running CSP[5]. This way, we were able to collect a dataset of CSP violations from existing websites. Notice that this is only a subset of all the CSP violations which may be triggered on these websites, since our crawler does not exercise any website functionality besides page loading.

We then performed a breakdown of the collected CSP violations. In particular, we focused on two categories of violations which are difficult to fix robustly in CSP, but are simple to address with CCSP: (*i*) violations triggered by the recursive inclusion of contents by any of the scripts loaded on the website, and (*ii*) violations triggered by HTTP redirects towards URLs which are not white-listed in the content security policy of the website. Both these scenarios are common, but challenging for CSP, since they involve elements which are not under the direct control of the developers of the websites.

To detect the violations in the first category, we relied on the structure of the collected violation reports, which includes both the URI of the website (named `document-uri`) and the URI of the element triggering the violation (named `source-file`); if there is a mismatch between the two, we put the violation into the first category. As to the second category of violations, we kept track of the detected HTTP redirects using our extension, storing the content of their `Location` header, and we performed a cross-check between this information and the dataset of violations: if there is a violation due to the inclusion of a content located at a URL found in a `Location` header, we put the violation in the second category. Violations can belong to both categories.

[5]We only focus on websites running CSP in enforcement mode. There are way more websites running CSP in report-only mode, but we excluded them from our analysis, because their policies are not necessarily accurate and intended to be eventually enforced [1].

### 5.2 Results

Overall, we found 959 CSP violations in 154 websites. We assigned 231 violations from 51 websites to the first category and 199 violations from 73 websites to the second category; we found only 7 violations belonging to both categories.

Table 1 provides the breakdown of the 231 violations due to script dependencies with respect to the violated CSP directive. One can readily observe that scripts often need to recursively include other scripts as expected, but they also typically load a bunch of other contents of different nature, most notably fonts, frames and images. The use of `'strict-dynamic'` can fix the 96 violations related to the `script-src` directive, which however represent only the 41.6% of the total number of violations in this category. To properly fix the other 135 cases in CSP, one would need to identify the missing dependencies of the included scripts and adapt the content security policy of the website accordingly, but this is not always easy for web developers, as testified by the fact that these violations occurred on popular websites.

| *Violated Directive* | *Violations* | *Sites* |
|---|---|---|
| `script-src` | 96 | 30 |
| `font-src` | 72 | 3 |
| `frame-src` | 32 | 25 |
| `img-src` | 17 | 5 |
| `connect-src` | 12 | 6 |
| `style-src` | 2 | 2 |

Table 1: Violations triggered by script dependencies

Table 2 reports the top 10 script providers by number of violations produced by the scripts they serve, as well as the number of websites where these violations are triggered. An interesting observation here is that, by writing appropriate CCSP headers for these 10 providers, one could fix 88 violations, which amount to the 38.1% of all the violations due to script dependencies we observed in the wild. Remarkably, this would fix violations in 37 websites, which amount to the 72.5% of all the websites which presented a violation in the first category. This suggests that the use of CCSP by the top script providers could provide a benefit to the majority of the websites.

As to the 199 violations due to HTTP redirects, we noticed that they were caused by redirectors from 46 different domains. Table 3 shows the top 10 redirectors by

| *Script Provider* | *Violations* | *Sites* |
|---|---|---|
| www.googletagmanager.com | 26 | 9 |
| apis.google.com | 13 | 13 |
| pagead2.googlesyndication.com | 11 | 2 |
| api.dmp.jimdo-server.com | 8 | 4 |
| assets.jimstatic.com | 7 | 4 |
| vogorana.ru | 7 | 2 |
| www.googleadservices.com | 7 | 6 |
| www.googletagservices.com | 4 | 2 |
| s.adroll.com | 3 | 3 |
| js-agent.newrelic.com | 2 | 2 |

Table 2: Top script providers by number of violations

number of violations, as well as the number of websites where these violations are triggered. It is worth noticing that, by writing appropriate CCSP headers for these 10 redirectors, one could already prevent 136 violations, which amount to the 68.3% of all the violations due to redirects. This would fix violations in 61 websites, which amount to the 83.6% of all the websites which presented a violation in the second category. This confirms again that a limited deployment of CCSP at major services could have a significant impact on the entire Web.

| *Redirector* | *Violations* | *Sites* |
|---|---|---|
| www.google.com | 47 | 38 |
| www.ingbank.pl | 20 | 2 |
| www.google-analytics.com | 14 | 13 |
| d.adroll.com | 12 | 3 |
| ads.stickyadstv.com | 9 | 1 |
| mc.yandex.ru | 9 | 1 |
| www.clearslide.com | 8 | 2 |
| cnfm.ad.dotandad.com | 6 | 1 |
| stats.g.doubleclick.net | 6 | 6 |
| ssl.google-analytics.com | 5 | 5 |

Table 3: Top redirectors by number of violations

## 6 Implementation and Testing

### 6.1 Prototype Implementation

We developed a proof-of-concept implementation of CCSP as a Chromium extension. The extension essentially works as a proxy based on the webRequest API[6]. It inspects all the incoming HTTP(S) responses looking for CCSP headers: if they are present, the extension parses them following the syntax described in the present paper and then strips away standard CSP headers (if any) to avoid conflicts. The extension internally keeps track of the protection state of all the open pages, closely following the CCSP enforcement model described in Section 3.3. Outgoing requests are then inspected to check whether they are allowed by the content security policy enforced in the current protection state of the page sending them: if this is not the case, the request is blocked.

Our prototype does not support any source expression which does not deal with outgoing requests, like `'unsafe-inline'`, since they are not trivial to handle via a browser extension (assuming it is even possible). The goal of the prototype is just providing a way to get hands-on experience with CCSP on existing websites and testify that it is possible to write accurate CCSP policies for them. On the long run, we would like to implement a more mature prototype of CCSP directly in Chromium: this should be relatively easy to do, because CCSP can use the existing CSP implementation as a black box.

### 6.2 Testing in the Wild

In our experiments, we fixed CSP violations found on two popular websites by using CCSP. We started by stipulating that their `CSP-Compose` headers should contain exactly the original content security policy and we then wrote appropriate `CSP-Union` and `CSP-Intersect` headers to fix the observed CSP violations. We finally injected these CCSP headers in the appropriate HTTP(S) responses via a local proxy.

**Twitter.** On Jan 13th 2017 we found that the content security policy of `twitter.com` was broken by the inclusion of a script from `https://cdn5.userzoom.com`, loaded by a script from `https://abs.twimg.com`.

Since `twimg.com` is controlled by Twitter, we decided to assume a high level of trust for all its sub-domains and we wrote the following `CSP-Intersect` header for the homepage of `twitter.com`:

```
CSP-Intersect:
   scope       *.twimg.com;
   script-src  https://*;
   default-src 'none';
```

This gives contents hosted on `twimg.com` the ability of relaxing the content security policy of Twitter to load arbitrary scripts over HTTPS. This is a very liberal behaviour, but it may be a realistic possibility if the team working at `abs.twimg.com` develops products independently from their final users at Twitter.

We then injected the following `CSP-Union` header in the script provided by `abs.twimg.com`:

[6]`https://developer.chrome.com/extensions/webRequest`

```
CSP-Union:
   script-src https://cdn5.userzoom.com;
```

In this specific case, we cannot white-list the exact script, QzI2OVQxNDQg.js, as its name is taken from the DOM and cannot be known by the server. However, the domain https://cdn5.userzoom.com is hard-coded in the script at abs.twimg.com, so we can reliably use that information for white-listing.

These two CCSP headers fixed the policy violation we found and allowed the script from abs.twimg.com to change its imported scripts without any intervention from the Twitter developers, as long as it correctly updates its CSP-Union header.

**Orange.** On Jan 23rd 2017 we detected three CSP violations at www.orange.sk, a national website of the popular telecommunication provider Orange.

The first violation was due to a script imported from static.hotjar.com, which was trying to create an iframe including contents from vars.hotjar.com. We fixed it by writing the following CSP-Intersect header for the homepage of www.orange.sk:

```
CSP-Intersect:
   scope       static.hotjar.com;
   frame-src   *.hotjar.com;
   default-src 'none';
```

We then attached the following CSP-Union header to the script from vars.hotjar.com:

```
CSP-Union:
   frame-src https://vars.hotjar.com/rcj-b2
      c1bce0a548059f409c021a46ea2224.html
```

Notice that this time we were able to white-list exactly the required contents, since the whole URL is readily available in the script code.

The other two violations were triggered by two images imported from www.google.com for tracking purposes, which were redirected to a national Google website not included in the content security policy. The web developers at www.orange.sk probably noticed these violations and tried to fix them by adding www.google.sk to the img-src directive, but since we were visiting the website from Italy, we got redirected to www.google.it and this domain was not included in the content security policy of www.orange.sk.

We then fixed these issues by adding the following information to the headers sent by www.orange.sk:

```
CSP-Intersect:
   scope       www.google.com;
   img-src     *;
   default-src 'none';
```

and by including the following headers to the redirect sent from www.google.com:

```
CSP-Union:
   img-src www.google.it
```

Notice that the correct top-level domain is known to the server, because it is also issuing the redirect request.

**Other Websites.** We discussed two practical examples of CCSP deployment, but one may wonder how difficult it is to write CCSP headers for other websites. To get a rough estimate about the challenges of the CCSP deployment more in general, we inspected our dataset of CSP violations and we collected for the top 10 script providers (by number of violations) the following information: the number of scripts they serve, the number of CSP violations triggered by these scripts, and the type of these violations. The results are in Table 4.

We think that the perspective offered by the table is pretty encouraging, because it suggests that even popular script providers only serve a small number of scripts to their customers, which means that the number of CCSP headers to write for them is typically limited. Moreover, scripts often load a very limited number of resources and only few of them need to load contents of variegate type. These two factors combined suggest that writing policies for scripts should be relatively easy on average, because these policies would have limited size and complexity.

## 7 Related Work

Several studies analysed the extent and the effectiveness of the CSP deployment in the wild and highlighted that web developers have troubles at configuring CSP correctly [9, 20, 1, 18]. Indeed, there have been a number of complementary proposals, with different level of complexity, on how to automatically generate content security policies for existing websites [2, 3, 7, 8]. The effectiveness of these proposals is still unclear, since automatically generating content security policies which are at the same time accurate and secure turned out to be extremely challenging, requiring a combination of static analysis, runtime monitoring and code rewriting. However, even a perfect policy generation algorithm can still lead to functionality problems upon content inclusion, due to unanticipated changes in the behaviour of included contents due to, e.g., the use of HTTP redirects or the relocation of script dependencies. CCSP was designed to support these behaviours under the assumption that most content providers are not actually malicious. It is also worth mentioning that CCSP is naturally effective at simplifying the policy specification process for web developers, assuming that content providers are willing

| *Script Provider* | *Scripts* | *Violations* | *Types of Violations* |
|---|---|---|---|
| www.googletagmanager.com | 9 | 1 | script |
| apis.google.com | 13 | 1 | frame |
| pagead2.googlesyndication.com | 3 | 5 | script, img |
| api.dmp.jimdo-server.com | 4 | 4 | connect, img |
| assets.jimstatic.com | 2 | 2 | script, img |
| vogorana.ru | 3 | 6 | script, frame, connect |
| www.googleadservices.com | 6 | 2 | frame |
| www.googletagservices.com | 3 | 3 | script |
| s.adroll.com | 3 | 2 | script |
| js-agent.newrelic.com | 1 | 2 | script |

Table 4: Types of violations for popular script providers

to dedicate some efforts to foster the integration between their contents and the content security policies of the embedding resources.

The idea of dynamically changing the enforced CSP policy advocated in CCSP is also present in the design of COWL, a confinement system for JavaScript code [12]. COWL assigns information flow labels to contexts (e.g., pages and iframes) and restricts their communication based on runtime label checks. Labels are allowed to change dynamically using meet and join operators, and implemented on top of CSP, which makes runtime policy composition part of COWL. However, COWL targets more ambitious security goals than (C)CSP by enforcing non-interference on labeled contexts and, as such, it is less flexible and harder to retrofit on existing websites. For these reasons, we believe that COWL and (C)CSP are complementary: one system may be better than the other one, depending on the desired security properties.

CSP Embedded Enforcement is a draft specification by the W3C which allows a protected resource to embed an iframe only if the latter accepts to enforce upon itself an embedder-specified set of restrictions expressed in terms of CSP [17]. The embedder advertises the restrictions using a new `Embedding-CSP` header including a content security policy, while the embedded content must attach a `Content-Security-Policy` header including a policy with at least the same restrictions to declare its compliance. It is worth noticing that CSP Embedded Enforcement is a first step towards making the CSP enforcement depend upon an interaction between the protected resource and the content providers, though the problems it addresses are orthogonal to CCSP. Similarly to CSP, CSP Embedded Enforcement asks web developers to get a thorough understanding of the contents they include to write a content security policy for them.

Other papers on CSP studied additional shortcomings of the standard, touching on a number of different issues: ineffectiveness against data exfiltration [13], difficult integration with browser extensions [5], unexpected bad interactions with the Same Origin Policy [10] and sub-optimal protection against code injection [6].

## 8 Conclusion

We proposed CCSP, an extension of CSP based on runtime policy composition. By shifting part of the policy specification process to content providers and by adding a dynamic dimension to CSP, CCSP reconciles the protection offered by fine-grained white-listing with a reasonable policy specification effort for web developers and a robust support for the highly dynamic nature of common web interactions. We analysed CCSP from different perspectives: security, backward compatibility and deployment cost. Moreover, we assessed its potential impact on the current web and we implemented a working prototype, which we tested on major websites. Our experiments show that popular content providers can deploy CCSP with limited efforts, leading to significant benefits for the large majority of the web.

As future work, we plan to implement CCSP directly in Chromium and carry out a large-scale analysis of its effectiveness, including a performance evaluation. We would also like to investigate automated ways to generate CCSP policies for both websites and content providers: since CCSP splits policy specification concerns between these two parties, we hope there is room for simplifying the automated policy generation process and making it more effective than for CSP. Finally, we would like to investigate the problem of supporting robust debugging facilities for CCSP in web browsers.

## Acknowledgements

We thank Daniel Hausknecht, Artur Janc, Sebastian Lekies, Andrei Sabelfeld, Michele Spagnuolo and Lukas Weichselbaum for the lively discussions about the current state of CSP. We also thank the anonymous reviewers for their useful comments and suggestions, and our shepherd Adam Doupé for his assistance in the realization of the final version of the paper. The paper acknowledges support by the MIUR project ADAPT.

## References

[1] CALZAVARA, S., RABITTI, A., AND BUGLIESI, M. Content Security Problems? Evaluating the effectiveness of Content Security Policy in the wild. In *CCS* (2016), pp. 1365–1375.

[2] DOUPÉ, A., CUI, W., JAKUBOWSKI, M. H., PEINADO, M., KRUEGEL, C., AND VIGNA, G. dedacota: toward preventing server-side XSS via automatic code and data separation. In *CCS* (2013), pp. 1205–1216.

[3] FAZZINI, M., SAXENA, P., AND ORSO, A. Autocsp: Automatically retrofitting CSP to web applications. In *ICSE* (2015), pp. 336–346.

[4] GOOGLE. Strict CSP, 2016. `https://csp.withgoogle.com/docs/strict-csp.html`.

[5] HAUSKNECHT, D., MAGAZINIUS, J., AND SABELFELD, A. May I? - Content Security Policy endorsement for browser extensions. In *DIMVA* (2015), pp. 261–281.

[6] JOHNS, M. Script-templates for the Content Security Policy. *J. Inf. Sec. Appl. 19*, 3 (2014), 209–223.

[7] KERSCHBAUMER, C., STAMM, S., AND BRUNTHALER, S. Injecting CSP for fun and security. In *ICISSP* (2016), pp. 15–25.

[8] PAN, X., CAO, Y., LIU, S., ZHOU, Y., CHEN, Y., AND ZHOU, T. Cspautogen: Black-box enforcement of content security policy upon real-world websites. In *CCS* (2016), pp. 653–665.

[9] PATIL, K., AND FREDERIK, B. A measurement study of the Content Security Policy on real-world applications. *I. J. Network Security 18*, 2 (2016), 383–392.

[10] SOME, D. F., BIELOVA, N., AND REZK, T. On the Content Security Policy violations due to the Same-Origin Policy. In *WWW* (2017), pp. 877–886.

[11] STAMM, S., STERNE, B., AND MARKHAM, G. Reining in the web with Content Security Policy. In *WWW* (2010), pp. 921–930.

[12] STEFAN, D., YANG, E. Z., MARCHENKO, P., RUSSO, A., HERMAN, D., KARP, B., AND MAZIÈRES, D. Protecting users by confining javascript with COWL. In *OSDI* (2014), pp. 131–146.

[13] VAN ACKER, S., HAUSKNECHT, D., AND SABELFELD, A. Data exfiltration in the face of CSP. In *ASIA CCS* (2016), pp. 853–864.

[14] W3C. Content Security Policy 1.0, 2012. `https://www.w3.org/TR/2012/CR-CSP-20121115/`.

[15] W3C. Content Security Policy Level 2, 2015. `https://www.w3.org/TR/CSP2/`.

[16] W3C. Content Security Policy Level 3, 2016. `https://w3c.github.io/webappsec-csp/`.

[17] W3C. CSP Embedded Enforcement, 2016. `https://www.w3.org/TR/csp-embedded-enforcement/`.

[18] WEICHSELBAUM, L., SPAGNUOLO, M., LEKIES, S., AND JANC, A. CSP is dead, long live CSP! On the insecurity of whitelists and the future of Content Security Policy. In *CCS* (2016), pp. 1376–1387.

[19] WEINBERGER, J., BARTH, A., AND SONG, D. Towards client-side HTML security policies. In *HotSec* (2011).

[20] WEISSBACHER, M., LAUINGER, T., AND ROBERTSON, W. K. Why is CSP failing? Trends and challenges in CSP adoption. In *RAID* (2014), pp. 212–233.

## A Theory

Our theory of joins and meets is based on a core fragment of CSP called CoreCSP. This fragment captures the essential ingredients of the standard and defines their (denotational) semantics, while removing uninspiring low-level details.

### A.1 CoreCSP

We presuppose a denumerable set of strings, ranged over by *str*. The syntax of *policies* is shown in Table 5, where we use dots (...) to denote additional omitted elements of a syntactic category (we assume the existence of an arbitrary but finite number of these elements).

This a rather direct counterpart of the syntax of CSP. The most notable points to mention are the following:

| | | | |
|---|---|---|---|
| Content types | $t$ | $::=$ | $\texttt{script} \mid \texttt{style} \mid \ldots$ |
| Schemes | $sc$ | $::=$ | $\texttt{http} \mid \texttt{https} \mid \texttt{data}$ |
| | | $\mid$ | $\texttt{blob} \mid \texttt{filesys} \mid \texttt{il}$ |
| | | $\mid$ | $\ldots$ |
| Policies | $p$ | $::=$ | $\vec{d} \mid p + p$ |
| Directives | $d$ | $::=$ | $t\texttt{-src}\ v$ |
| | | $\mid$ | $\texttt{default-src}\ v$ |
| Directive values | $v$ | $::=$ | $\{se_1, \ldots, se_n\}$ |
| Source expressions | $se$ | $::=$ | $h \mid \texttt{unsafe-inline}$ |
| | | $\mid$ | $\texttt{inline}(str)$ |
| Hosts ($sc \neq \texttt{il}$) | $h$ | $::=$ | $\texttt{self} \mid sc \mid he \mid (sc, he)$ |
| Host expressions | $he$ | $::=$ | $* \mid *.str \mid str$ |

Table 5: Syntax of CoreCSP

1. we assume the existence of a distinguished scheme `il`, used to identify inline scripts and stylesheets. This scheme cannot occur inside policies, but it is convenient to define their formal semantics;

2. we do not discriminate between hashes and nonces in source expressions, since this is unimportant at our level of abstraction. Rather, we uniformly represent them using the source expression $\texttt{inline}(str)$, where *str* is a string which uniquely identifies the white-listed inline script or stylesheet;

3. we define directive values as sets of source expressions, rather than lists of source expressions. This difference is uninteresting in practice, since source expression lists are always parsed as sets;

4. for simplicity, we do not model ports and paths in the syntax of source expressions.

To simplify the formalization, we only consider *well-formed* policies, according to the following definition.

**Assumption 1** (Well-formed Policies)**.** We assume that policies are *well-formed*, i.e., for each directive value $v$ occurring therein, we have that $\texttt{unsafe-inline} \in v$ implies $\texttt{inline}(str) \notin v$.

The syntax of CSP is more liberal, because it allows one to write policies violating the constraint above. However, there is no loss of generality in focusing only on well-formed policies, since if both `unsafe-inline` and $\texttt{inline}(str)$ occur in the same directive, only one of them is enforced: browsers supporting CSP 1.0 would ignore $\texttt{inline}(str)$, while browsers implementing more recent versions of CSP would ignore `unsafe-inline`.

The definition of the semantics of CoreCSP is based on three main entities: *locations* are uniquely identified sources of contents; *subjects* are HTTP(S) web pages enforcing a CSP policy; and *objects* are contents available for inclusion by subjects.

**Definition 3** (Locations)**.** A *location* is a pair $l = (sc, str)$. We let $\mathscr{L}$ stand for a denumerable set of locations and we let $L$ range over subsets of $\mathscr{L}$.

**Definition 4** (Subjects)**.** A *subject* is a pair $s = (l, str)$ where $l = (sc, str')$ with $sc \in \{\texttt{http}, \texttt{https}\}$.

**Definition 5** (Objects)**.** An *object* is a pair $o = (l, str)$. We let $\mathscr{O}$ stand for a denumerable set of objects and we let $O$ range over subsets of $\mathscr{O}$.

We use the projection functions $\pi_1(\cdot)$ and $\pi_2(\cdot)$ to extract the components of a pair (location, subject or object). We also make the following typing assumption.

**Assumption 2** (Typing of Objects)**.** We assume that objects are *typed*. Formally, this means that $\mathscr{O}$ is partitioned into the subsets $\mathscr{O}_{t_1}, \ldots, \mathscr{O}_{t_n}$, where $t_1, \ldots, t_n$ are the available content types. We also assume that, for all objects $o = ((\texttt{il}, str'), str)$, we have $o \in \mathscr{O}_{\texttt{script}} \cup \mathscr{O}_{\texttt{style}}$.

The judgement $se \rightsquigarrow_s L$ defines the semantics of source expressions. It reads as: the source expression *se* allows the subject $s$ to include contents from the locations $L$. The formal definition is given in Table 6, where we let $\rhd$ be the smallest reflexive relation on schemes such that $\texttt{http} \rhd \texttt{https}$. The judgement generalizes to values by having: $v \rightsquigarrow_s \{l \mid \exists se \in v, \exists L \subseteq \mathscr{L} : se \rightsquigarrow_s L \wedge l \in L\}$.

$$\texttt{self} \rightsquigarrow_s \{\pi_1(s)\} \qquad sc \rightsquigarrow_s \{l \mid \pi_1(l) = sc\}$$

$$* \rightsquigarrow_s \{l \mid \pi_1(l) \notin \{\texttt{data}, \texttt{blob}, \texttt{filesys}, \texttt{il}\}\}$$

$$str \rightsquigarrow_s \{l \mid \pi_1(\pi_1(s)) \rhd \pi_1(l) \wedge \pi_2(l) = str\}$$

$$*.str \rightsquigarrow_s \{l \mid \pi_1(\pi_1(s)) \rhd \pi_1(l) \wedge \exists str' : \pi_2(l) = str'.str\}$$

$$(sc, str) \rightsquigarrow_s \{(sc, str)\}$$

$$(sc, *.str) \rightsquigarrow_s \{l \mid \pi_1(l) = sc \wedge \exists str' : \pi_2(l) = str'.str\}$$

$$(sc, *) \rightsquigarrow_s \{l \mid \pi_1(l) = sc\}$$

$$\texttt{unsafe-inline} \rightsquigarrow_s \{l \mid \pi_1(l) = \texttt{il}\}$$

$$\texttt{inline}(str) \rightsquigarrow_s \{(\texttt{il}, str)\}$$

Table 6: Semantics of Source Expressions ($se \rightsquigarrow_s L$)

We then define operators to get the value bound to a directive in a policy. Given a list of directives $\vec{d}$ and a content type $t$, we define $\vec{d} \downarrow t$ as the value bound to the first $t$`-src` directive, if any; otherwise, the value bound to the first `default-src` directive, if any; and in absence of both, we let it be the wildcard $\{*\}$.

**Definition 6** (Lookup)**.** Given a list of directives $\vec{d}$ and a content type $t$, we define $\vec{d}.t$ as follows:

$$\vec{d}.t = \begin{cases} v & \text{if } \vec{d} = \vec{d_1}, t\texttt{-src}\ v, \vec{d_2} \wedge \\ & \quad \forall d \in \{\vec{d_1}\}, \forall v' : d \neq t\texttt{-src}\ v' \\ \bot & \text{otherwise} \end{cases}$$

We then define the *lookup* operator $\vec{d} \downarrow t$ as follows:

$$\vec{d} \downarrow t = \begin{cases} \vec{d}.t & \text{if } \vec{d}.t \neq \bot \\ v & \text{if } \vec{d}.t = \bot \wedge \vec{d} = \vec{d_1}, \texttt{default-src}\ v, \vec{d_2} \wedge \\ & \quad \forall d \in \{\vec{d_1}\}, \forall v' : d \neq \texttt{default-src}\ v' \\ \{*\} & \text{otherwise} \end{cases}$$

The judgement $p \vdash s \hookleftarrow_t O$ defines the semantics of policies. It reads as: the policy $p$ allows the subject $s$ to include as contents of type $t$ the objects $O$. The formal definition is given in Table 7.

(D-VAL)
$$\frac{\vec{d} \downarrow t = v \qquad v \rightsquigarrow_s L}{\vec{d} \vdash s \hookleftarrow_t \{o \in \mathscr{O}_t \mid \pi_1(o) \in L\}}$$

(D-CONJ)
$$\frac{p_1 \vdash s \hookleftarrow_t O_1 \qquad p_2 \vdash s \hookleftarrow_t O_2}{p_1 + p_2 \vdash s \hookleftarrow_t O_1 \cap O_2}$$

Table 7: Semantics of Policies ($p \vdash s \hookleftarrow_t O$)

The semantics of a CSP policy depends on the subject restricted by the policy. This makes reasoning on CSP policies quite complicated, hence we introduce a class of policies, called *normal policies*, whose semantics does not depend on a specific subject. The restriction to normal policies does not bring any loss of generality in practice, since any policy can be translated into an equivalent normal policy by using a subject-directed compilation.

The syntax of normal policies is obtained by replacing the occurrences of $h$ in Table 5 with $\overline{h}$, where:

$$\overline{h} ::= sc \mid * \mid (sc, he).$$

We define normal source expressions and normal directive values accordingly.

**Definition 7** (Normalization)**.** Given a source expression $se$ and a subject $s$, we define the *normalization* of $se$ under $s$, written $\langle se \rangle_s$, as follows:

$$\langle se \rangle_s = \begin{cases} \{\pi_1(s)\} & \text{if } se = \texttt{self} \\ \{(sc, str) \mid \pi_1(\pi_1(s)) \rhd sc\} & \text{if } se = str \\ \{(sc, *.str) \mid \pi_1(\pi_1(s)) \rhd sc\} & \text{if } se = *.str \\ \{se\} & \text{otherwise} \end{cases}$$

The normalization of a directive value $v$ under $s$ is defined as $\langle v \rangle_s = \bigcup_{se \in v} \langle se \rangle_s$. The normalization of a policy $p$ under $s$, written $\langle p \rangle_s$, is obtained by normalizing under $s$ each directive value occurring in $p$.

**Lemma 1** (Properties of Normalization)**.** *The following properties hold true:*

1. *for all policies $p$ and subjects $s$, $\langle p \rangle_s$ is normal;*
2. *for all policies $p$, subjects $s$ and content types $t$, we have $p \vdash s \hookleftarrow_t O$ if and only if $\langle p \rangle_s \vdash s \hookleftarrow_t O$;*
3. *for all normal policies $p$, subjects $s_1, s_2$ and content types $t$, we have that $p \vdash s_1 \hookleftarrow_t O_1$ and $p \vdash s_2 \hookleftarrow_t O_2$ imply $O_1 = O_2$.*

## A.2 Technical Preliminaries

We now introduce the technical ingredients needed to implement our proposal. We start by defining a binary relation $\sqsubseteq_{src}$ on normal source expressions. Intuitively, we have $se_1 \sqsubseteq_{src} se_2$ if and only if $se_1$ denotes no more locations than $se_2$ (for all subjects).

**Definition 8** ($\sqsubseteq_{src}$ Relation)**.** We let $\sqsubseteq_{src}$ be the least reflexive relation on normal source expressions defined by the following rules:

$$\frac{sc \notin \{\texttt{data}, \texttt{blob}, \texttt{filesys}, \texttt{il}\}}{sc \sqsubseteq_{src} *}$$

$$\frac{sc \notin \{\texttt{data}, \texttt{blob}, \texttt{filesys}, \texttt{il}\}}{(sc, he) \sqsubseteq_{src} *} \qquad sc \sqsubseteq_{src} (sc, *)$$

$$(sc, he) \sqsubseteq_{src} sc \qquad (sc, str) \sqsubseteq_{src} (sc, *)$$

$$(sc, *.str) \sqsubseteq_{src} (sc, *) \qquad (sc, str'.str) \sqsubseteq_{src} (sc, *.str)$$

$$\texttt{inline}(str) \sqsubseteq_{src} \texttt{unsafe-inline}$$

To compare policy permissiveness, however, there are a couple of issues left to be addressed:

1. a policy $p$ may enforce multiple restrictions on the same content type $t$, specifically when $p = p_1 + p_2$ for some $p_1, p_2$. In this case, multiple directive values must be taken into account when reasoning about the inclusion of contents of type $t$;

2. a policy $p$ may enforce restrictions on the inclusion of contents of type $t$ by using directives of two different formats, namely $t$`-src` $v$ or `default-src` $v'$. One has then to ensure that the appropriate directive value is chosen when reasoning about the inclusion of contents of type $t$.

We address these issues by defining a *smart* lookup operator $p \Downarrow t$ which, given a policy $p$ and a content type $t$, returns a directive value which captures all the restrictions put in place by $p$ on $t$. This operator is based on the following definition of *meet* of directive values.

**Definition 9** (Meet of Values)**.** Given two normal directive values $v_1, v_2$, we define their *meet* $v_1 \bowtie v_2$ as follows:

$$\begin{aligned} v_1 \bowtie v_2 &= \{se \in v_1 \mid \exists se' \in v_2 : se \sqsubseteq_{src} se'\} \\ &\cup \{se \in v_2 \mid \exists se' \in v_1 : se \sqsubseteq_{src} se'\}. \end{aligned}$$

**Lemma 2** (Correctness of Meet)**.** *For all normal directive values $v_1, v_2$ and subjects $s$, we have $v_1 \rightsquigarrow_s L_1$ and $v_2 \rightsquigarrow_s L_2$ if and only if $v_1 \bowtie v_2 \rightsquigarrow_s L_1 \cap L_2$.*

**Definition 10** (Smart Lookup)**.** Given a normal policy $p$ and a content type $t$, we define $p \Downarrow t$ as follows:

$$p \Downarrow t = \begin{cases} \vec{d} \downarrow t & \text{if } p = \vec{d} \\ (p_1 \Downarrow t) \bowtie (p_2 \Downarrow t) & \text{if } p = p_1 + p_2 \end{cases}$$

**Lemma 3** (Correctness of Smart Lookup)**.** *For all normal policies $p$, subjects $s$ and content types $t$, we have:*

$$p \vdash s \hookleftarrow_t \{o \in \mathscr{O}_t \mid \exists L \subseteq \mathscr{L} : p \Downarrow t \rightsquigarrow_s L \wedge \pi_1(o) \in L\}.$$

## A.3 Join and Meet

The *join* of two policies allows a subject to include some content if and only if at least one of the two policies does.

**Definition 11** (Join of Policies)**.** Given two policies $p_1, p_2$ and a subject $s$, we define the *join* $p_1 \sqcup_s p_2$ as the least policy s.t. $(p_1 \sqcup_s p_2).t = (\langle p_1 \rangle_s \Downarrow t) \cup (\langle p_2 \rangle_s \Downarrow t)$.

**Theorem 1** (Correctness of Join)**.** *$p_1 \vdash s \hookleftarrow_t O_1$ and $p_2 \vdash s \hookleftarrow_t O_2$ iff $p_1 \sqcup_s p_2 \vdash s \hookleftarrow_t O_1 \cup O_2$.*

*Proof.* Let $p_1 \vdash s \hookleftarrow_t O_1$, $p_2 \vdash s \hookleftarrow_t O_2$ and $p_1 \sqcup_s p_2 \vdash s \hookleftarrow_t O$, we show that $O = O_1 \cup O_2$ by proving $O \subseteq O_1 \cup O_2$ and $O \supseteq O_1 \cup O_2$:

($\subseteq$) Let $o \in O$, then there exists $v$ such that $(p_1 \sqcup_s p_2).t = v$ and $v \rightsquigarrow_s L$ for some $L$ such that $\pi_1(o) \in L$. By definition, this means that there exists $se \in v$ such that $se \rightsquigarrow_s L'$ for some $L'$ such that $\pi_1(o) \in L'$. By definition of join, we have $v = (\langle p_1 \rangle_s \Downarrow t) \cup (\langle p_2 \rangle_s \Downarrow t)$. Hence, we have either $se \in \langle p_1 \rangle_s \Downarrow t$ or $se \in \langle p_2 \rangle_s \Downarrow t$. Assume that $se \in \langle p_1 \rangle_s \Downarrow t$, then $o \in O_1$ by using Lemma 3 and the observation that normalization does not change the semantics of policies. The case $se \in \langle p_2 \rangle_s \Downarrow t$ is symmetric;

($\supseteq$) Let $o \in O_1 \cup O_2$, then either $o \in O_1$ or $o \in O_2$. Assume that $o \in O_1$, the other case is symmetric. By using Lemma 3 and the observation that normalization does not change the semantics of policies, there exists $v$ such that $\langle p_1 \rangle_s \Downarrow t = v$ and $v \rightsquigarrow_s L$ for some $L$ such that $\pi_1(o) \in L$. By definition of join, we have $(p_1 \sqcup_s p_2).t = v'$ for some $v'$ such that $v' \rightsquigarrow_s L'$ with $L \subseteq L'$. This implies $o \in O$.

□

The *meet* of two policies allows a subject to include some content if and only if both policies do. Defining the meet is more complicated in CSP, because not all browsers correctly handle the conjunction of two policies [1]. The key idea of the definition is to reuse the meet operator $\bowtie$ defined for directive values, since we proved that $v_1 \rightsquigarrow_s L_1$ and $v_2 \rightsquigarrow_s L_2$ if and only if $v_1 \bowtie v_2 \rightsquigarrow_v L_1 \cap L_2$ (see Lemma 2).

**Definition 12** (Meet of Policies)**.** Given two policies $p_1, p_2$ and a subject $s$, we define the *meet* $p_1 \sqcap_s p_2$ as the least policy s.t. $(p_1 \sqcap_s p_2).t = (\langle p_1 \rangle_s \Downarrow t) \bowtie (\langle p_2 \rangle_s \Downarrow t)$.

**Theorem 2** (Correctness of Meet)**.** *$p_1 \vdash s \hookleftarrow_t O_1$ and $p_2 \vdash s \hookleftarrow_t O_2$ iff $p_1 \sqcap_s p_2 \vdash s \hookleftarrow_t O_1 \cap O_2$.*

*Proof.* Let $p_1 \vdash s \hookleftarrow_t O_1$, $p_2 \vdash s \hookleftarrow_t O_2$ and $p_1 \sqcap_s p_2 \vdash s \hookleftarrow_t O$, we show that $O = O_1 \cap O_2$ by proving $O \subseteq O_1 \cap O_2$ and $O \supseteq O_1 \cap O_2$:

($\subseteq$) Let $o \in O$, then there exists $v$ such that $(p_1 \sqcap_s p_2).t = v$ and $v \rightsquigarrow_s L$ for some $L$ such that $\pi_1(o) \in L$. By definition of meet, we have $v = (\langle p_1 \rangle_s \Downarrow t) \bowtie (\langle p_2 \rangle_s \Downarrow t)$. Let $\langle p_1 \rangle_s \rightsquigarrow_s L_1$ and $\langle p_1 \rangle_s \rightsquigarrow_s L_2$, then $\pi_1(o) \in L_1 \cap L_2$ by Lemma 2. This implies $\pi_1(o) \in L_1$ and $\pi_1(o) \in L_2$ by definition of intersection. Hence, we have $o \in O_1$ and $o \in O_2$ by using Lemma 3 and the observation that normalization does not change the semantics of policies;

($\supseteq$) Let $o \in O_1 \cap O_2$, then $o \in O_1$ and $o \in O_2$. By using Lemma 3 and the observation that normalization does not change the semantics of policies, there exist $v_1, v_2$ such that $\langle p_1 \rangle_s \Downarrow t = v_1$, $\langle p_2 \rangle_s \Downarrow t = v_2$, $v_1 \rightsquigarrow_s L_1$ for some $L_1$ such that $\pi_1(o) \in L_1$, and $v_2 \rightsquigarrow_s L_2$ for some $L_2$ such that $\pi_1(o) \in L_2$. This implies that $\pi_1(o) \in L_1 \cap L_2$. By definition of meet, we have $(p_1 \sqcap_s p_2).t = v_1 \bowtie v_2$ and we know that $v_1 \bowtie v_2 \rightsquigarrow_s L_1 \cap L_2$ by Lemma 2, hence $o \in O$.

□

Observe that both the definitions of join and meet are parametric with respect to a subject. In the case of normal policies, however, this subject can be dropped. Since all policies can be transformed into equivalent normal policies (Lemma 1), in the body of the paper we just write $\sqcup$ and $\sqcap$ for simplicity.