# Semantics-Based Analysis of Content Security Policy Deployment

**Article** *in* ACM Transactions on the Web · January 2018

DOI: 10.1145/3149408

Stefano Calzavara
Università Ca' Foscari Venezia
**62** PUBLICATIONS **414** CITATIONS
SEE PROFILE

Alvise Rabitti
Università Ca' Foscari Venezia
**17** PUBLICATIONS **86** CITATIONS
SEE PROFILE

Michele Bugliesi
Università Ca' Foscari Venezia
**132** PUBLICATIONS **1,651** CITATIONS
SEE PROFILE

# Semantics-Based Analysis of Content Security Policy Deployment

STEFANO CALZAVARA, Università Ca' Foscari Venezia
ALVISE RABITTI, Università Ca' Foscari Venezia
MICHELE BUGLIESI, Università Ca' Foscari Venezia

Content Security Policy (CSP) is a recent W3C standard introduced to prevent and mitigate the impact of content injection vulnerabilities on websites. In this paper we introduce a formal semantics for the latest stable version of the standard, CSP Level 2. We then perform a systematic, large-scale analysis of the effectiveness of the current CSP deployment, using the formal semantics to substantiate our methodology and to assess the impact of the detected issues. We focus on four key aspects that affect the effectiveness of CSP: browser support, website adoption, correct configuration and constant maintenance. Our analysis shows that browser support for CSP is largely satisfactory, with the exception of few notable issues, but unfortunately there are several shortcomings relative to the other three aspects. CSP appears to have a rather limited deployment as yet and, more crucially, existing policies exhibit a number of weaknesses and misconfiguration errors. Moreover, content security policies are not regularly updated to ban insecure practices and remove unintended security violations. We argue that many of these problems can be fixed by better exploiting the monitoring facilities of CSP, while other issues deserve additional research, being more rooted into the CSP design.

CCS Concepts: • **Security and privacy** → **Browser security**; *Formal security models*;

Additional Key Words and Phrases: Content Security Policy, Formal methods, Web security

**ACM Reference Format:**
Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. 2017. Semantics-Based Analysis of Content Security Policy Deployment. *ACM Trans. Web* 1, 1 (October 2017), 36 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

The same-origin policy (SOP) is the baseline defense mechanism implemented in web browsers to provide confidentiality and integrity guarantees for contents provided by unrelated websites. Under SOP, data from `https://www.mybank.com` is shielded from read or write attempts by scripts from other web origins, like `https://www.evil.com` and `http://www.mybank.com`. Though apparently secure, it is well-known that SOP can be bypassed by *content injection* attacks. In these attacks, attacker-controlled contents are injected in benign web pages and become indistinguishable from legitimate contents, thus inheriting their privileges.

The most effective techniques to defend against content injection are *input sanitization* and *output encoding*, which prevent dangerous contents like malicious script tags from entering benign web pages [21]. Unfortunately, input sanitization is typically difficult to get right and output encoding can be accidentally overlooked, so content injections are still pervasive on the Web [20]. This motivated the development of complementary in-depth defense mechanisms aimed at mitigating the effects of a successful content injection [8, 12, 17, 18, 25]. Among these, Content Security Policy

1559-1131/2017/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

(CSP) is by far the most popular and well-established solution, being standardized by the W3C and supported by all major web browsers [25, 29].

CSP is a language for defining restrictions on the functionality of web pages, ideally to limit their capabilities to the least set of privileges they need to work correctly. Most notably, CSP significantly mitigates the dangers of a successful content injection by disallowing the execution of inline scripts and by banning a few dangerous functions used for turning strings into code, like the infamous `eval`. These default restrictions can be explicitly relaxed to simplify deployment, though this is strongly discouraged by the standard. Moreover, CSP allows the specification of constraints on content inclusion based on a white-listing mechanism, whereby different content types, like images or scripts, are bound to the sole set of origins allowed to supply those contents. This way, injected markup elements can only be abused to load contents from white-listed web origins, thus further reducing the room for attacks.

### 1.1 Research Goals and Contributions

Our main goal is to assess the state of the art in the use and effectiveness of CSP as a security mechanism for websites against content injection attacks. To better understand the standard, we first define a *formal semantics* for its latest stable version, CSP Level 2 (Section 3). The semantics provides a rigorous and concise representation of the most important elements of CSP, which allows us to substantiate our analysis methodology and to formally assess the impact of the detected practical issues.

We focus on four key aspects affecting the effectiveness of CSP:

(1) *browser support*: we design a set of experiments to test the browser implementations of the CSP specification. We run the experiments in all major web browsers, including their mobile variants. We report on the outcome of the experiments, highlighting the cases where at least one browser does not behave as expected and discussing their security import. Our investigation reveals a dangerous behaviour of Microsoft Edge and a subtle quirk in all browser implementations, which deserves a careful security analysis (Section 4);
(2) *website adoption*: we collect the content security policies sent by the Alexa Top 1M websites [2] and we analyze them to shed light on the current state of the CSP deployment, which turns out to be quite limited. We also investigate which features and use cases of CSP are popular among web developers and which ones are largely ignored, identifying a few common bad practices (Section 5);
(3) *correct configuration*: we identify five common classes of misconfigurations made by web developers when writing content security policies and we discuss their security and usability import. We show that the very large majority of the websites we surveyed deploy content security policies which do not provide robust defenses against script injection (Section 6);
(4) *constant maintenance*: we repeat the crawling of the Alexa Top 1M for 22 weeks, automatically collecting both CSP headers and violations to the policies contained therein. We identify websites committing to CSP or abdicating from it during this timespan and we analyze how existing policies changed during the 22 weeks, discussing good and bad practices in the wild. Finally, we investigate correlations between changes to policies and policy violations, concluding that content security policies change less frequently than needed (Section 7).

We present our perspective on the main findings of the paper in Section 8. Our take is that many of the problems we found can be fixed by better exploiting the monitoring facilities of CSP, while other issues deserve more research by the community and the industry, being more rooted into the CSP design. Finally, we refer to Section 9 for a discussion on related work.

### 1.2 Novel Contents for Journal Publication

The present work extends and improves a published conference paper [4]. On the theoretical side, the main addition is the definition of a formal semantics for CSP Level 2, presented in Section 3. This is useful to pursue several practical goals: assessing the security import of inaccurate browser implementations of the CSP specification, checking policies for vulnerability to script injection and comparing policy permissiveness. Specifically, the following changes and additions make our investigation more rigorous and comprehensive:

(1) the security considerations about inaccurate browser implementations of CSP presented in [4] are now backed up by theorems and formal proofs;
(2) the syntactic conditions defined in [4] to detect policies vulnerable to script injection are proved correct with respect to the formal model. Moreover, we identify new syntactic conditions on policies which prove the absence of these vulnerabilities under a few additional assumptions;
(3) the study of the evolution of the CSP deployment presented in [4] has been significantly expanded and automated by developing a policy comparison tool, which implements the permissiveness analysis defined in the formal model and allows us to check how policies evolve over time.

We also performed a number of useful revisions, additions and updates to the original study, most notably by significantly enlarging its scope (to more than 16,000 websites) and by performing a more systematic deduplication of the collected data.

## 2 BACKGROUND: CONTENT SECURITY POLICY

We provide a brief introduction to Content Security Policy (CSP). This section contains just enough information to understand the essence of the CSP specification and our main technical contributions. We refer to the official documentation for full details about the standard [29].

### 2.1 Overview

A content security policy is a list of *directives*, restricting content inclusion for web pages by means of a white-listing mechanism. Directives bind content types to lists of sources from which a CSP-protected web page is allowed to include resources of that specific type. For instance, the directive `img-src https://a.com` specifies that a web page can only load images from the host `a.com` via the HTTPS protocol. CSP is a client-server defense mechanism: content security policies are specified by web developers using HTTP(S) headers or meta elements in HTML pages, while their enforcement is performed at the browser side on a per-page basis. Content security policies can be run in two modes: the *enforcement* mode applies all the content restrictions specified by the policy, while the *report-only* mode does not restrict the website functionality, but it just tells browsers to log policy violations in the JavaScript console. In both modes, the `report-uri` directive can be used to specify a URI where browsers should send JSON-based security reports when a policy violation occurs. Policies in enforcement mode are sent in the `Content-Security-Policy` header, while report-only policies are sent in the `Content-Security-Policy-Report-Only` header.

Table 1 reports selected directive types available in CSP: if a content security policy does not include an explicit directive for a given content type, the `default-src` directive is applied to it as a fallback. Allowed sources for content inclusion are specified using *source expressions*, a sort of regular expressions used to express sets of web origins in a compact way. Content inclusion from a URL is only allowed if the URL *matches* any of the source expressions specified for the appropriate content type. The relevant details of the matching algorithm will be formalized in Section 3, for now we just assume the existence of such algorithm.

Table 1. Selected CSP Directives

| *Directive* | *Restricted Contents* |
|---|---|
| `img-src` | Images |
| `script-src` | JavaScript, XSLT |
| `style-src` | Stylesheets (CSS) |
| `connect-src` | Targets of XMLHttpRequest |
| `default-src` | Contents without explicit directives |

The informal semantics of a content security policy can be summarized as follows:

(1) inline scripts are blocked, unless the source expression `'unsafe-inline'` is included in the `script-src` directive (or in the `default-src` directive in absence of `script-src`);
(2) inline styles are blocked, unless the source expression `'unsafe-inline'` is included in the `style-src` directive (or in the `default-src` directive in absence of `style-src`);
(3) the conversion of strings into code via `eval` and similar functions is blocked, unless the source expression `'unsafe-eval'` is in the `script-src` directive (or in the `default-src` directive in absence of `script-src`);
(4) some dangerous methods of the CSS Object Model like `insertRule` are blocked, unless the source expression `'unsafe-eval'` is in the `style-src` directive (or in the `default-src` directive in absence of `style-src`);
(5) the inclusion of a content of type $t$ from a URL $u$ is allowed if and only if one of these conditions holds:
   (a) $u$ matches a source expression in $t$-`src`;
   (b) there is no $t$-`src` directive and $u$ matches a source expression in `default-src`;
   (c) there is neither $t$-`src` nor `default-src`.

If more than one content security policy is deployed on the same web page, each policy must be individually enforced following the rules above.

## 2.2 CSP Versions

The core of CSP is a fine-grained mechanism for white-listing content inclusions, defined in the CSP 1.0 specification [28] and summarized in the previous section. The latest stable version of the standard, called CSP Level 2 [29], includes a number of new features on top of the original CSP core. One of the major changes with respect to CSP 1.0 is the introduction of mechanisms to relax the above restrictions on inline scripts and stylesheets, without falling back to the dramatic absence of security guarantees provided by `'unsafe-inline'`. Specifically, CSP Level 2 allows one to white-list individual inline scripts and styles by using *nonces* or *hashes*.

The `'nonce-$value'` source expression white-lists inline scripts or styles with a `nonce` attribute equal to `$value`, while the `'shaXXX-$value'` source expression white-lists inline scripts or styles whose hash (computed using the `shaXXX` algorithm) is `$value`. Nonces should be random values which are freshly generated upon each page request. The same nonce attribute can be assigned to multiple scripts or styles, so that multiple inline elements can be white-listed using just a single nonce, which simplifies policy specification. Hashes, however, provide better security guarantees than nonces, because they additionally provide an integrity guarantee for the white-listed scripts or styles, while nonces can be reused to white-list arbitrary inline elements when they fall under the control of an attacker, for instance because they are easily predictable or not freshly generated upon each page request.

*Example 2.1.* To exemplify the most important concepts of CSP, consider the following content security policy:

```
script-src https://a.com 'nonce-a33f5b005d';
img-src https://b.com;
default-src https://*
```

This policy allows the inclusion of scripts from `https://a.com` and the inclusion of images from `https://b.com`. Inline scripts are blocked, unless their script tag is marked with a `nonce` attribute set to `a33f5b005d`. Contents which are not scripts or images, e.g., stylesheets, can be included from every host, provided that they are delivered using the HTTPS protocol.

We conclude this section by mentioning the current working draft of the CSP specification, called CSP Level 3 [32]. The main extension with respect to CSP Level 2 is the introduction of the `'strict-dynamic'` source expression, designed to simplify the process of recursive script inclusion without triggering security violations. Our study focuses on CSP Level 2, because it is the latest stable version of the standard and a candidate recommendation of the W3C, but the formal semantics presented in the next section can be straightforwardly adapted to the current draft specification of CSP Level 3.

## 3 FORMAL ANALYSIS OF CSP LEVEL 2

We introduce here a denotational semantics for a significant fragment of CSP Level 2, which we call CoreCSP. We then discuss some security applications of the semantics: reasoning on the import of inaccurate browser implementations, checking vulnerability to script injection and comparing policy permissiveness.

### 3.1 Syntax and Semantics of CoreCSP

*3.1.1 Syntax.* We let *str* range over the denumerable set of strings. The syntax of policies is shown in Table 2, where we use dots (...) to denote additional omitted elements of a syntactic category. We assume a finite number of content types and an arbitrary number of schemes.

Table 2. Syntax of CoreCSP

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

The syntax of CoreCSP is a rather direct counterpart of the syntax of CSP Level 2. A policy $p$ is either a list of directives $\vec{d}$ or the conjunction of two policies $p_1 + p_2$. Directives, in turn, bind content types $t$ to directive values $v$; their syntax also includes a default directive, applied to all the contents not restricted by other directives. Directive values are sets of source expressions $se$, whose semantics will be explained in the following.

A few points of the syntax are worth discussing:

(1) we assume the existence of a distinguished scheme `inl`, used to identify inline scripts and styles. This scheme cannot syntactically occur inside policies, but it is convenient to define their formal semantics;
(2) we only model hashes rather than nonces as a mechanism to white-list individual inline scripts and styles. The reason is that it is not possible to define the semantics of a policy using nonces just based on the syntax of the policy itself, but one would need to model also the contents of the HTML page where the policy is enforced to identify the white-listed elements, based on the value of their `nonce` attribute;
(3) we define directive values as sets of source expressions, rather than lists of source expressions. This difference is uninteresting in practice, since lists of source expressions are always parsed as sets;
(4) for simplicity, we do not model ports and paths in the syntax of source expressions. They can be easily added to the formalism, at the cost of making the technical details more tedious without adding much to the formalization insights.

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

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

The syntax of CSP Level 2 is more liberal than this and it allows the specification of policies violating the constraint above. However, in practice there is no loss of generality in focusing only on well-formed policies, since if both `unsafe-inline` and $\mathtt{hash}(str)$ occur in the same directive, only one of them is enforced by web browsers. Specifically, browsers compliant with CSP 1.0 would ignore $\mathtt{hash}(str)$, while browsers supporting CSP Level 2 would ignore `unsafe-inline`. We assume CSP policies are simplified like this before being represented in CoreCSP.

The definition of the formal semantics is based on three main entities: *locations* are uniquely identifiable sources of contents; *subjects* are HTTP(S) web pages enforcing a content security policy; and *objects* are contents available to subjects for inclusion.

*Definition 3.1 (Location).* A *location* is a pair $l = (sc, str)$, where $str$ is a string representing a hostname. We let $\mathcal{L}$ stand for the denumerable set of all locations and we let $L$ range over subsets of $\mathcal{L}$.

*Definition 3.2 (Subject).* A *subject* is a pair $s = (l, str)$, where $l = (sc, str')$ with $sc \in \{\mathtt{http}, \mathtt{https}\}$ and $str$ is a string representing a path.

*Definition 3.3 (Object).* An *object* is a pair $o = (l, str)$. We let $\mathcal{O}$ stand for the denumerable set of all objects and we let $O$ range over subsets of $\mathcal{O}$.

We use the projection functions $\pi_1(\cdot)$ and $\pi_2(\cdot)$ to extract the components of a pair, be it a location, a subject or an object. We also make the following typing assumption, which ensures that objects can only be white-listed for inclusion by using directives of the expected type. This is useful to develop a faithful model of CSP.

Assumption 2 (Typing of Objects). *We assume that objects are* typed. *Formally, this means that $\mathcal{O}$ is partitioned into the subsets $\mathcal{O}_{t_1}, \ldots, \mathcal{O}_{t_n}$, where $t_1, \ldots, t_n$ are the available content types. We also assume that, for all objects $o = ((\mathtt{inl}, str'), str)$, we have $o \in \mathcal{O}_{\mathtt{script}} \cup \mathcal{O}_{\mathtt{style}}$, i.e., the only inline objects are scripts and stylesheets.*

*3.1.2 Semantics of Source Expressions.* 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

Table 3. Denotational Semantics of Source Expressions ($se \rightsquigarrow_s L$)

$$\texttt{self} \rightsquigarrow_s \{\pi_1(s)\} \qquad sc \rightsquigarrow_s \{l \mid \pi_1(l) = sc\} \qquad * \rightsquigarrow_s \{l \mid \pi_1(l) \notin \{\texttt{data}, \texttt{blob}, \texttt{fsys}, \texttt{inl}\}\}$$

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

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

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

$$\texttt{unsafe-inline} \rightsquigarrow_s \{l \mid \pi_1(l) = \texttt{inl}\} \qquad \texttt{hash}(str) \rightsquigarrow_s \{(\texttt{inl}, str)\}$$

locations $L$. The formal definition is given in Table 3, where we let $\rhd$ denote the smallest reflexive relation on schemes such that $\texttt{http} \rhd \texttt{https}$.

A brief explanation follows. The `self` source expression only denotes the location of the subject. A scheme source expression *sc* denotes all the locations with that scheme. The $*$ source expression white-lists all the locations, with the exception of those with scheme `data`, `blob`, `fsys` or `inl`. A string source expression *str* denotes the locations (`http`, *str*) and (`https`, *str*) for HTTP subjects, but only the location (`https`, *str*) for HTTPS subjects. The semantics of $*.str$ follows the same logic on the scheme, but any location whose second component has $.str$ as suffix is white-listed. The semantics of $(sc, str)$, $(sc, *.str)$ and $(sc, *)$ is straightforward. The `unsafe-inline` source expression denotes all the locations with scheme `inl`, while the `hash`(*str*) source expression only white-lists the location (`inl`, *str*). Having defined the semantics of source expression, the semantics of directive values $v$ is defined as expected:

$$v \rightsquigarrow_s \{l \mid \exists se \in v, \exists L \subseteq \mathcal{L} : se \rightsquigarrow_s L \land l \in L\}.$$

*3.1.3 Semantics of Policies.* The semantics of policies readily follows from the semantics of directive values. It is based on a *lookup* operator which, given a list of directives $\vec{d}$ and a content type $t$, returns the directive value $v$ which determines the restrictions enforced by $\vec{d}$ when including contents of type $t$. Intuitively, $v$ is the value bound to the first $t$`-src` directive in $\vec{d}$, if any, otherwise it is the value bound to the first `default-src` directive; if there is not even a default directive in $\vec{d}$, the wildcard $\{*\}$ is returned. The formal definition of the lookup operator is given next.

*Definition 3.4 (Lookup).* Given a list of directives $\vec{d}$ and a content type $t$, we define the *syntactic lookup* operator $\vec{d}.t$ as follows:

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

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

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

The judgement $p \vdash s \leftarrowtail_t O$ 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 4. Rule

Table 4. Denotational Semantics of Policies ($p \vdash s \leftarrowtail_t O$)

$$\text{(D-VAL)}\quad \frac{\vec{d} \downarrow t = v \qquad v \rightsquigarrow_s L}{\vec{d} \vdash s \leftarrowtail_t \{o \in O_t \mid \pi_1(o) \in L\}} \qquad\qquad \text{(D-CONJ)}\quad \frac{p_1 \vdash s \leftarrowtail_t O_1 \qquad p_2 \vdash s \leftarrowtail_t O_2}{p_1 + p_2 \vdash s \leftarrowtail_t O_1 \cap O_2}$$

(D-Val) allows the inclusion of the objects of the appropriate type whose locations are white-listed by the directive value $v$ returned by the lookup operator. Rule (D-Conj) defines the semantics of policies built using the conjunction operator + by intersecting the sets of objects white-listed by the individual policies. In other words, a content inclusion is allowed if and only if it is allowed by all the individual policies.

*Example 3.5.* Consider the subject $s = ((\texttt{https}, \texttt{example.com}), \texttt{/home})$ and the following policy:

$$p = \texttt{style-src}\ \{\texttt{nice-css.com}, \texttt{hash(xyz)}\}, \texttt{default-src}\ \{*\}.$$

Let $p \vdash s \leftarrowtail_{\texttt{style}} O_{\texttt{style}}$ and $p \vdash s \leftarrowtail_{\texttt{script}} O_{\texttt{script}}$, we have:

(1) $o_1 = ((\texttt{inl}, \texttt{xyz}), \texttt{body \{color: purple\}}) \in O_{\texttt{style}}$, since $\texttt{hash(xyz)} \in p \downarrow \texttt{style}$ and $\texttt{hash(xyz)} \rightsquigarrow_s \{(\texttt{inl}, \texttt{xyz})\}$;
(2) $o_2 = ((\texttt{https}, \texttt{nice-css.com}), \texttt{/cool-style.css}) \in O_{\texttt{style}}$, since $\texttt{nice-css.com} \in p \downarrow \texttt{style}$ and $\texttt{nice-css.com} \rightsquigarrow_s \{(\texttt{https}, \texttt{nice-css.com})\}$;
(3) $o_3 = ((\texttt{http}, \texttt{nice-css.com}), \texttt{/cool-style.css}) \notin O_{\texttt{style}}$, though `nice-css.com` is an allowed source for stylesheet inclusion. The reason is that $s$ runs on HTTPS and $p$ does not provide a scheme for `nice-css.com`, hence only HTTPS contents from `nice-css.com` are white-listed;
(4) $o_4 = ((\texttt{inl}, \texttt{xyz}), \texttt{alert(1)}) \notin O_{\texttt{script}}$, since $p \downarrow \texttt{script} = \{*\}$ and $*$ does not white-list inline elements.

### 3.2 Formal Reasoning on CSP Policies

We now set the ground for the security applications of the semantics we anticipated. We do this by defining a few technical ingredients which are useful to support formal reasoning on CSP policies: a *pre-order* on source expressions characterizing their permissiveness and a *smart lookup* operator on policies defining the restrictions enforced by multiple conjuncted policies in terms of a single directive value.

*3.2.1 Policy Normalization.* The semantics of policies depends on the subject enforcing them, which complicates formal reasoning. We thus introduce a class of policies, called *normal policies*, whose semantics does not depend on the enforcing subject. We then show that 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 $h$ in Table 2 with $\overline{h}$, where:

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

Normal source expressions and normal directive values are defined accordingly.

*Definition 3.6 (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 syntactically occurring in $p$.

For example, building on Example 3.5, the normalization of the policy $p$ under the subject $s$ is the following policy:

$$\langle p \rangle_s = \texttt{style-src}\ \{(\texttt{https}, \texttt{nice-css.com}), \texttt{hash(xyz)}\}, \texttt{default-src}\ \{*\}.$$

It is easy to note that the sets of objects white-listed by $p$ and $\langle p \rangle_s$ coincide. This is in fact a general result.

Lemma 3.7 (Properties of Normalization). *The following properties hold:*

(1) *for all policies $p$ and subjects $s$, $\langle p \rangle_s$ is a normal policy;*
(2) *for all policies $p$, subjects $s$ and content types $t$, we have $p \vdash s \leftarrowtail_t O$ iff $\langle p \rangle_s \vdash s \leftarrowtail_t O$;*
(3) *for all normal policies $p$, subjects $s_1, s_2$ and content types $t$, if we have $p \vdash s_1 \leftarrowtail_t O_1$ and $p \vdash s_2 \leftarrowtail_t O_2$, then $O_1 = O_2$.*

Proof. See Appendix A.1. □

*3.2.2 Ordering Source Expressions.* We introduce a binary relation $\sqsubseteq_{src}$ on normal source expressions such that $se_1 \sqsubseteq_{src} se_2$ if and only if $se_1$ denotes no more locations than $se_2$ for all subjects. Formally, $\sqsubseteq_{src}$ is defined as the least reflexive relation on normal source expressions satisfying the rules in Table 5. It is easy to prove that $\sqsubseteq_{src}$ is also transitive, hence it defines a pre-order on normal source expressions.

Table 5. Ordering Normal Source Expressions ($se_1 \sqsubseteq_{src} se_2$)

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

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

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

We use the $\sqsubseteq_{src}$ relation to define a binary relation $\sqsubseteq$ on normal directive values, which generalizes to them the previous intuition. For all normal directive values $v_1, v_2$, let $v_1 \sqsubseteq v_2$ if and only if $\forall se_1 \in v_1 : \exists se_2 \in v_2 : se_1 \sqsubseteq_{src} se_2$. The desired properties of $\sqsubseteq$ can be formalized as follows.

Lemma 3.8 (Correctness of $\sqsubseteq$). *For all normal directive values $v_1, v_2$, the following properties hold true:*

(1) *If $v_1 \sqsubseteq v_2$, then for all subjects $s$ we have $v_1 \rightsquigarrow_s L_1$ and $v_2 \rightsquigarrow_s L_2$ with $L_1 \subseteq L_2$;*
(2) *If there exists a subject $s$ such that $v_1 \rightsquigarrow_s L_1$ and $v_2 \rightsquigarrow_s L_2$ with $L_1 \subseteq L_2$, then $v_1 \sqsubseteq v_2$.*

Proof. See Appendix A.2. □

*3.2.3 Smart Lookup.* The $\sqsubseteq$ relation is a powerful tool to reason about the security of policies, since the set of objects which can be included according to a policy ultimately depends on the locations white-listed via its directive values. To effectively use $\sqsubseteq$ on arbitrary policies, 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` or `default-src`. One has then to ensure that the appropriate directive value is taken into account.

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 two normal directive values.

*Definition 3.9 (Meet).* Given two normal directive values $v_1, v_2$, we define their *meet* as:

$$v_1 \sqcap v_2 = \{se \in v_1 \mid \exists se' \in v_2 : se \sqsubseteq_{src} se'\} \cup \{se \in v_2 \mid \exists se' \in v_1 : se \sqsubseteq_{src} se'\}.$$

The definition of the smart lookup operator is now simple. If a policy is defined as the conjunction of multiple policies, the smart lookup operator computes the meet of the directive values returned by the standard lookup operator on the individual conjuncted policies. Otherwise, it just behaves like the standard lookup operator.

*Definition 3.10 (Smart Lookup).* Given a normal policy $p$ and a content type $t$, we define $p \Downarrow t$ as follows:

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

The desired property of the smart lookup operator can be formalized as follows.

Lemma 3.11 (Correctness of Smart Lookup). *For all normal policies $p$, subjects $s$ and content types $t$, we have $p \vdash s \leftarrowtail_t \{o \in \mathcal{O}_t \mid \exists L \subseteq \mathcal{L} : p \Downarrow t \rightsquigarrow_s L \wedge \pi_1(o) \in L\}$.*

Proof. See Appendix A.3. □

We conclude this section with a mild technical assumption, which ensures that all the white-listed locations host at least one object of the expected type. In other words, we assume that policies do not contain any useless information: if a location is white-listed, something is available for inclusion therein.

Assumption 3 (Proper White-Listing). *For all normal policies $p$, subjects $s$ and content types $t$, we have that $p \Downarrow t \rightsquigarrow_s L$ implies that for all $l \in L$ there exists $o \in \mathcal{O}_t$ such that $\pi_1(o) = l$.*

### 3.3 Application 1: Reasoning on Browser Implementations

CoreCSP is a faithful model of the official CSP Level 2 specification [29]. Unfortunately, it is well-known that browsers do not always implement meticulously existing specifications and the security import of these inaccuracies may not be obvious. In our investigation, we observed that CSP is no exception, because Microsoft Edge does not follow the CSP specification when enforcing multiple policies on the same page (see Section 4.3) and all the major browsers implement an unexpected behaviour when dealing with inline scripts and styles (see Section 4.4).

When these inaccurate browser implementations are identified, e.g., by code review or testing, one can use CoreCSP to get a formal understanding of their security import: we refer to the aforementioned sections for such an analysis.

### 3.4 Application 2: Vulnerability to Script Injection

Content injection may take different forms and be exploited to mount a number of attacks, like UI redressing. In our view, however, the most dangerous form of client-side content injection on the Web is XSS, where arbitrary attacker-controlled scripts are injected in benign web pages. Our goal here is defining syntactic checks on content security policies under which a content injection can lead to arbitrary script injection, despite a correct policy enforcement. We use these checks to automate the security analysis of existing content security policies deployed in the wild (see Section 6.6).

*3.4.1 Formal Definition.* We start by defining the threat model. We represent an attacker $A \subseteq \mathcal{L}$ just as a set of locations identifying attacker-controlled contents, which we call *tainted objects*. This general model is useful to reason about *white-list safety*: indeed, the policy semantics is agnostic to the trust of web hosts, but our threat model allows one to discriminate between a policy which white-lists `good.com` and a policy which white-lists `evil.com` as legitimate sources for script inclusion.

*Definition 3.12 (Tainted Objects).* Given an attacker $A$, the set of its *tainted objects* of type $t$ is defined as $A_t = \{o \in \mathcal{O}_t \mid \pi_1(o) \in A\}$.

If a policy allows the inclusion of tainted objects, then there is a potential security issue which deserves scrutiny. This is formalized as follows.

*Definition 3.13 (Vulnerability to Injection).* A policy $p$ leaves the subject $s$ *vulnerable to injection* of contents of type $t$ by the attacker $A$, written $p \vdash s \curvearrowleft_t A$, if and only if there exists a set of objects $O$ such that $p \vdash s \leftarrowtail_t O$ and $O \cap A_t \neq \emptyset$. Vulnerability to script injection is defined by instantiating $t$ to `script`.

This threat model is very general, but in this work we find particularly useful to focus on a particular class of attackers modelling the standard *web attacker* from the literature, which is normally used when reasoning about content injection [1]. The web attacker operates a set of malicious websites and can respond to HTTP(S) requests sent to them with arbitrary content. We assume the attacker set up HTTPS on his web servers. Also, the attacker can attempt to exploit code injection vulnerabilities by means of inline scripts and data URIs, which provide a means to include inline elements as if they were external resources. Notice that the attacker's ability to inject inline scripts is limited by the use of hashes in content security policies.

*Definition 3.14 (Web Attacker).* Let $H, I \neq \emptyset$ be sets of strings representing hosts and identifiers of inline scripts respectively. We define the *web attacker* $W[H, I]$ as:

$$\begin{array}{rcl} W[H,I] & = & \{(\texttt{http}, \mathit{str}), (\texttt{https}, \mathit{str}) \mid \mathit{str} \in H\} \cup \{(\texttt{inl}, \mathit{str}) \mid \mathit{str} \in I\} \\ & \cup & \{l \in \mathcal{L} \mid \pi_1(l) = \texttt{data}\}. \end{array}$$

We let $W$ stand for the *canonic* web attacker $W[\{\texttt{attacker.com}\}, \{\texttt{att}\}]$, where `att` represents the identifier of a malicious inline script.

*3.4.2 Syntactic Checks.* Having defined the threat model, we can introduce the following notion of *liberal* source expression. Liberal source expressions constitute a poor mechanism to restrict script inclusion, since some of the locations they white-list are controlled by the canonic web attacker.

*Definition 3.15 (Liberality).* A source expression is *liberal* if and only if it is the wildcard $*$, the `unsafe-inline` source expression, or any of the schemes `http`, `https` or `data`. A directive value $v$ is *liberal* iff it contains at least one liberal source expression.

The explanation of the definition is quite intuitive: the wildcard $*$ and the HTTP(S) scheme include `attacker.com` as a valid source for content inclusion, while `unsafe-inline` and `data` enable the injection of inline scripts.

The first result of this section provides a syntactic criterion to check whether a policy leaves a website vulnerable to script injection attacks (XSS).

Theorem 3.16 (Vulnerability to XSS). *For all policies $p$ and subjects $s$, if $\langle p \rangle_s \Downarrow \texttt{script} = v$ for some liberal directive value $v$, then $p \vdash s \curvearrowleft_{\texttt{script}} W$.*

Proof. Assume $\langle p \rangle_s \Downarrow \texttt{script} = v$ for some liberal $v$ and let $L$ be the set of locations such that $v \rightsquigarrow_s L$. Since $v$ is liberal, there must exist a liberal source expression $se \in v$. Since $\langle p \rangle_s$ is a normal policy, $\langle p \rangle_s \vdash s \leftarrowtail_{\texttt{script}} O$ with $O = \{o \in \mathcal{O}_{\texttt{script}} \mid \pi_1(o) \in L\}$ by Lemma 3.11. We show that $O \cap W_{\texttt{script}} \neq \emptyset$, which proves $p \vdash s \curvearrowleft_{\texttt{script}} W$, since we have $p \vdash s \leftarrowtail_{\texttt{script}} O$ by Lemma 3.7. To show $O \cap W_{\texttt{script}} \neq \emptyset$, it is enough to prove that $L \cap W \neq \emptyset$ by Assumption 3. Let $L'$ be the set of locations such that $se \rightsquigarrow_s L'$. Since $L' \subseteq L$ by definition of $v \rightsquigarrow_s L$, we can prove $L \cap W \neq \emptyset$ by showing $L' \cap W \neq \emptyset$. This can be shown by a case analysis on $se$:

- $se = \texttt{http}$: we have $L' = \{l \mid \pi_1(l) = \texttt{http}\}$, hence $L' \cap W = \{(\texttt{http}, \texttt{attacker.com})\}$;
- $se = \texttt{https}$: analogous to the previous case;
- $se = *$: we have $\{l \mid \pi_1(l) = \texttt{http}\} \subseteq L'$, hence $(\texttt{http}, \texttt{attacker.com}) \in L' \cap W$;
- $se = \texttt{data}$: we have $L' = \{l \mid \pi_1(l) = \texttt{data}\}$, hence $L' \cap W = L'$;
- $se = \texttt{unsafe-inline}$: we have $L' = \{l \mid \pi_1(l) = \texttt{inl}\}$, hence $L' \cap W = \{(\texttt{inl}, \texttt{att})\}$.

□

Given a web attacker, it is also possible to check whether a policy provides protection against script injection attempts by that attacker. We do this by identifying the set of the sole source expressions which may open a breach for script injection.

*Definition 3.17 (Weakness).* Given a web attacker $W[H, I]$, a source expression $se$ is *weak* against $W[H, I]$ iff any of the following conditions holds true:

(1) there exists $str \in H$ such that $(\texttt{http}, str) \sqsubseteq_{src} se$ or $(\texttt{https}, str) \sqsubseteq_{src} se$;
(2) there exists $str \in I$ such that $\texttt{hash}(str) \sqsubseteq_{src} se$;
(3) we have $se \sqsubseteq_{src} \texttt{data}$.

A directive value $v$ is *weak* against $W[H, I]$ iff it contains at least one source expression which is weak against $W[H, I]$.

We can prove that the presence of weak source expressions is a necessary condition for a successful script injection by the considered web attacker.

Theorem 3.18 (Protection against XSS). *For all policies $p$ and subjects $s$, if $p \vdash s \curvearrowleft_{\texttt{script}} W[H, I]$ for the web attacker $W[H, I]$, then $\langle p \rangle_s \Downarrow \texttt{script} = v$ for some directive value $v$ which is weak against $W[H, I]$.*

Proof. Assume $p \vdash s \curvearrowleft_{\texttt{script}} W[H, I]$ for the web attacker $W[H, I]$, then there exists $O$ such that $p \vdash s \leftarrowtail_{\texttt{script}} O$ and $O \cap W[H, I]_{\texttt{script}} \neq \emptyset$, i.e., there exists $o \in O \cap W[H, I]_{\texttt{script}}$. We then observe that $p \vdash s \leftarrowtail_{\texttt{script}} O$ implies $\langle p \rangle_s \vdash s \leftarrowtail_{\texttt{script}} O$ by Lemma 3.7. Let $\langle p \rangle_s \Downarrow \texttt{script} = v$ and let $L$ be the set of locations such that $v \rightsquigarrow_s L$. Since $\langle p \rangle_s$ is a normal policy, we have $O = \{o' \in \mathcal{O}_{\texttt{script}} \mid \pi_1(o') \in L\}$ by Lemma 3.11, hence $\pi_1(o) \in L$. Since we also know that $o \in W[H, I]_{\texttt{script}}$, we must have $\pi_1(o) \in W[H, I]$. We then perform a case distinction on the structure of $\pi_1(o)$ and we show for each case that $v$ must include a weak source expression:

- $\pi_1(o) = (\texttt{http}, str)$ with $str \in H$: since $(\texttt{http}, str) \in L$, we have $\{(\texttt{http}, str)\} \sqsubseteq v$ by Lemma 3.8. This means that there exists $se \in v$ such that $(\texttt{http}, str) \sqsubseteq_{src} se$, hence $v$ is weak against $W[H, I]$;
- $\pi_1(o) = (\texttt{https}, str)$ with $str \in H$: analogous to the previous case;
- $\pi_1(o) = (\texttt{inl}, str)$ with $str \in I$: since $(\texttt{inl}, str) \in L$, we have $\{(\texttt{inl}, str)\} \sqsubseteq v$ by Lemma 3.8. This means that there exists $se \in v$ such that $(\texttt{inl}, str) \sqsubseteq_{src} se$, hence $v$ is weak against $W[H, I]$;
- $\pi_1(o) = (\texttt{data}, str)$: since $(\texttt{data}, str) \in L$, we must have $\{(\texttt{data}, str)\} \sqsubseteq v$ by Lemma 3.8. This means that there exists $se \in v$ such that $(\texttt{data}, str) \sqsubseteq_{src} se$. An inspection of the rules defining the $\sqsubseteq_{src}$ relation (in Table 5) shows that $se = (\texttt{data}, he)$ for some $he$ or $se = \texttt{data}$, hence $se \sqsubseteq_{src} \texttt{data}$ and $v$ is weak against $W[H, I]$.

□

### 3.5 Application 3: Policy Permissiveness Analysis

We now formalize a notion of *policy permissiveness* and identify syntactic checks to prove or disprove that one policy is no more permissive than another one. We implemented these checks in a policy comparison tool developed in PHP, which we make publicly available online[1]. This tool allows one to analyze the evolution over time of existing content security policies deployed in the wild and to understand the import of the observed policy changes (see Section 7.3).

*3.5.1 Formal Definition.* Given the denotational style of the formal semantics, it is very natural and intuitive to compare the permissiveness of two policies by comparing the sets of their white-listed objects.

*Definition 3.19 (Permissiveness).* Given two policies $p_1, p_2$, we say that $p_1$ is *no more permissive than* $p_2$ for the subject $s$ and the content type $t$ (written $p_1 \leq_{s,t} p_2$) if and only if $p_1 \vdash s \hookleftarrow_t O_1$ and $p_2 \vdash s \hookleftarrow_t O_2$ imply $O_1 \subseteq O_2$. When universally quantifying over all subjects and/or content types, we omit $s$ and/or $t$ from the notation.

Observe that the definition is consistent with the threat model, since $p_1 \vdash s \curvearrowleft_t A$ and $p_1 \leq_{s,t} p_2$ imply $p_2 \vdash s \curvearrowleft_t A$ for all the attackers $A$. Despite its simplicity, however, policy permissiveness is non-trivial to check syntactically.

*Example 3.20.* Consider the following two policies:

$$\begin{array}{lcl} p_1 & = & \texttt{script-src}\ \{\texttt{a.com}\}, \texttt{style-src}\ \{\texttt{b.com}\}, \texttt{default-src}\ \{\texttt{https}\} \\ p_2 & = & \texttt{script-src}\ \{\texttt{a.com}, \texttt{c.com}\}, \texttt{default-src}\ \{*\} \end{array}$$

We have $p_1 \leq p_2$. However, the syntactic structure of the two policies is different, since $p_2$ has less directives than $p_1$. Also, the directive values occurring in $p_2$ are more permissive than those in $p_1$. This may be due to the addition of new source expressions ($\{\texttt{a.com}\}$ vs. $\{\texttt{a.com}, \texttt{c.com}\}$) or the relaxation of existing ones ($\{\texttt{https}\}$ vs. $\{*\}$).

*3.5.2 Syntactic Checks.* We propose syntactic checks to prove or disprove $p_1 \leq_{s,t} p_2$, based on the $\sqsubseteq$ relation and the smart lookup operator.

THEOREM 3.21 (CHECKING PERMISSIVENESS). *For all policies $p_1, p_2$, we have $p_1 \leq_{s,t} p_2$ if and only if $\langle p_1 \rangle_s \Downarrow t \sqsubseteq \langle p_2 \rangle_s \Downarrow t$.*

PROOF. We show the two directions separately:

[1] http://www.dais.unive.it/~csp/csp-comparison-tool/

($\Rightarrow$) Let $p_1 \leq_{s,t} p_2$, we show $\langle p_1 \rangle_s \Downarrow t \sqsubseteq \langle p_2 \rangle_s \Downarrow t$. By definition of $p_1 \leq_s p_2$, we have $p_1 \vdash s \leftarrowtail_t O_1$ and $p_2 \vdash s \leftarrowtail_t O_2$ with $O_1 \subseteq O_2$. By Lemma 3.7, $p_1 \vdash s \leftarrowtail_t O_1$ and $p_2 \vdash s \leftarrowtail_t O_2$ imply $\langle p_1 \rangle_s \vdash s \leftarrowtail_t O_1$ and $\langle p_2 \rangle_s \vdash s \leftarrowtail_t O_2$. By Lemma 3.11 we have $O_1 = \{o \in \mathcal{O}_t \mid \exists L_1 \subseteq \mathcal{L} : \langle p_1 \rangle_s \Downarrow t \leadsto_s L_1 \wedge \pi_1(o) \in L_1\}$ and $O_2 = \{o \in \mathcal{O}_t \mid \exists L_2 \subseteq \mathcal{L} : \langle p_2 \rangle_s \Downarrow t \leadsto_s L_2 \wedge \pi_1(o) \in L_2\}$. Since $O_1 \subseteq O_2$, we must have $L_1 \subseteq L_2$ by Assumption 3. By Lemma 3.8, we get $\langle p_1 \rangle_s \Downarrow t \sqsubseteq \langle p_2 \rangle_s \Downarrow t$.

($\Leftarrow$) Let $\langle p_1 \rangle_s \Downarrow t \sqsubseteq \langle p_2 \rangle_s \Downarrow t$, we show $p_1 \vdash s \leftarrowtail_t O_1'$ and $p_2 \vdash s \leftarrowtail_t O_2'$ for some $O_1', O_2'$ such that $O_1' \subseteq O_2'$. By Lemma 3.8, $\langle p_1 \rangle_s \Downarrow t \sqsubseteq \langle p_2 \rangle_s \Downarrow t$ implies $\langle p_1 \rangle_s \Downarrow t \leadsto_s L_1$ and $\langle p_2 \rangle_s \Downarrow t \leadsto_s L_2$ with $L_1 \subseteq L_2$. By Lemma 3.11 we have $\langle p_1 \rangle_s \vdash s \leftarrowtail_t O_1$ and $\langle p_2 \rangle_s \vdash s \leftarrowtail_t O_2$ with $O_1 = \{o \in \mathcal{O}_t \mid \exists L_1 \subseteq \mathcal{L} : \langle p_1 \rangle_s \Downarrow t \leadsto_s L_1 \wedge \pi_1(o) \in L_1\}$ and $O_2 = \{o \in \mathcal{O}_t \mid \exists L_2 \subseteq \mathcal{L} : \langle p_2 \rangle_s \Downarrow t \leadsto_s L_2 \wedge \pi_1(o) \in L_2\}$. By Lemma 3.7, $\langle p_1 \rangle_s \vdash s \leftarrowtail_t O_1$ and $\langle p_2 \rangle_s \vdash s \leftarrowtail_t O_2$ imply $p_1 \vdash s \leftarrowtail_t O_1$ and $p_2 \vdash s \leftarrowtail_t O_2$. Since $L_1 \subseteq L_2$, we must have $O_1 \subseteq O_2$.

□

*Example 3.22.* Pick again the policies $p_1$ and $p_2$ from Example 3.20 and consider the subject $s = ((\texttt{https}, \texttt{example.com}), \texttt{/home})$. We have:

$$\begin{array}{lcl} \langle p_1 \rangle_s & = & \texttt{script-src}\ \{(\texttt{https}, \texttt{a.com})\}, \texttt{style-src}\ \{(\texttt{https}, \texttt{b.com})\}, \texttt{default-src}\ \{\texttt{https}\} \\ \langle p_2 \rangle_s & = & \texttt{script-src}\ \{(\texttt{https}, \texttt{a.com}), (\texttt{https}, \texttt{c.com})\}, \texttt{default-src}\ \{*\} \end{array}$$

Thus, for the different content types `script` and `style`, we get:

$$\begin{array}{lcl} \langle p_1 \rangle_s \Downarrow \texttt{script} & \sqsubseteq & \langle p_2 \rangle_s \Downarrow \texttt{script} \\ \langle p_1 \rangle_s \Downarrow \texttt{style} & \sqsubseteq & \langle p_2 \rangle_s \Downarrow \texttt{style} \end{array}$$

We then conclude $p_1 \leq_s p_2$.

## 4 TESTING BROWSER SUPPORT FOR CSP

We devised a number of experiments to test to which extent the implementation of CSP in major web browsers is compliant with the CSP Level 2 specification [29], at least as it comes to the fragment formalized in CoreCSP. Our goal was finding both subtle corner cases of the CSP specification which deserve clarification and plain deviations with respect to expected browser behaviours. When an unexpected behaviour emerged from our experiments, we used CoreCSP to assess its security import.

### 4.1 Methodology

We manually created a small set of HTML pages sending content security policies in enforcement mode, designing them so that the browser behaviour upon policy enforcement is made explicit by visual clues. We make these pages available online, along with a brief explanation of each of them[2]. We do not claim that our investigation tested all the corner cases of the specification, but we are confident about the effectiveness of our test suite in providing a good coverage of the most relevant aspects of CSP which are commonly used, as formalized by the CoreCSP semantics. We leave as future work the automated generation of more comprehensive test cases using the formal semantics.

We visited the web pages with different browsers: Mozilla Firefox 46, Chromium 50, Opera 36, Safari 9.1 and Microsoft Edge 25.10586.0.0, as well as their mobile variants. Notice that Safari and Microsoft Edge do not support CSP Level 2, but only CSP 1.0. Features specific to CSP Level 2 have not been tested on those browsers.

[2] http://www.dais.unive.it/~csp/investigating-browser-support-for-csp/

### 4.2 Passed Tests

All the browsers successfully passed the following tests:

(1) *Enforcing multiple directives*: The syntax of CSP allows the inclusion of multiple directives for the same content type (e.g., `script-src`) in the same header. The expected behaviour in this case is that only the first directive is enforced, while the other ones are ignored;
(2) *Default scheme assignment*: The syntax of source expressions includes host source expressions of the form `a.com`. In these cases lacking an explicit scheme, the CSP specification mandates a default scheme assignment based on the scheme of the page deploying the policy: `a.com` must be interpreted as `https://a.com` in HTTPS pages, and as both `http://a.com` and `https://a.com` in HTTP pages;
(3) *Wildcard*: In CSP Level 2, the `*` source expression is a wildcard matching any URL whose scheme is not `blob`, `data` or `filesystem`. These schemes are considered dangerous, since the content of URLs with these schemes is often derived from a response body and may be under the control of an attacker. Notice that in CSP 1.0 the wildcard simply matches any URL;
(4) *Ambiguities on inline scripts*: The `script-src` directive may include both `'unsafe-inline'` and nonces or hashes white-listing individual inline scripts. In this case, the CSP specification mandates that only inline scripts white-listed using nonces or hashes are allowed to run. Recall that nonces and hashes are not available in CSP 1.0.

### 4.3 Enforcing Multiple Policies

Multiple content security policies can be specified for the same web page in different headers. The CSP specification recommends that, if multiple policies are present on the same page, all of them must be individually enforced. Our experiments assessed that all browsers behave according to the specification, but for Microsoft Edge, which concatenates policies included in different headers and only enforces the first encountered directive for each content type. For instance, if the first header includes the directive `script-src a.com b.com` and the second header includes the directive `script-src a.com c.com`, the protected page can load scripts from both `a.com` and `b.com` in Microsoft Edge, though only `a.com` should be a valid source for script inclusion based on the CSP specification. Though the presence of multiple headers with different directives for the same content type may sound strange at first, this situation may happen in presence of security gateways and web application firewalls run by large organizations [29]. In these cases, the behaviour of Microsoft Edge is more permissive than the CSP specification and may leave room for attacks.

We can formally prove that the implementation of CSP provided by Microsoft Edge is potentially dangerous, i.e., it can only make policies more permissive than intended. To encode the behaviour of Microsoft Edge in our semantics, we define a *linearisation* operator (denoted by $|\cdot|$). This operator removes the pluses from the syntax of policies, thus squeezing multiple conjuncted policies into a single list of directives. The operational behaviour of Microsoft Edge can be encoded in our formalism by assuming that all policies are linearised by the browser before being enforced.

*Definition 4.1 (Linearisation).* Given a policy $p$, we define its *linearisation* $|p|$ as:

$$|p| = \begin{cases} \vec{d} & \text{if } p = \vec{d} \\ |p_1|, |p_2| & \text{if } p = p_1 + p_2 \end{cases}$$

Theorem 4.2 (Dangerous Implementation of Microsoft Edge). *For all policies $p$ and subjects $s$, we have $p \leq_s |p|$.*

Proof. The proof uses the following observations:

(a) for all normal directive values $v_1, \ldots, v_n$, we have $v_1 \sqcap \ldots \sqcap v_n \sqsubseteq v_i$ for all $v_i$. The proof is by induction on $n$, using the reflexivity of $\sqsubseteq$ for the base case and appealing to Lemma A.4 for the inductive case;

(b) for all policies $p$ and subjects $s$, we have $\langle |p| \rangle_s = |\langle p \rangle_s|$. The proof is by induction on the structure of $p$, using the definitions of the two operators;

(c) for all lists of directives $\vec{d}_1, \ldots, \vec{d}_n$ and content types $t$, we have $(\vec{d}_1, \ldots, \vec{d}_n) \downarrow t = \vec{d}_i \downarrow t$ for some $\vec{d}_i$. The proof is by case analysis, using the definition of $(\vec{d}_1, \ldots, \vec{d}_n) \downarrow t$.

We show that, for all policies $p$, subjects $s$ and content types $t$, we have $\langle p \rangle_s \Downarrow t \sqsubseteq \langle |p| \rangle_s \Downarrow t$, which proves the statement by Theorem 3.21. If $p = \vec{d}$ for some $\vec{d}$, we have $p = |p|$ and the result is trivial. Otherwise, let $p = \vec{d}_1 + \ldots + \vec{d}_n$ for some $\vec{d}_1, \ldots, \vec{d}_n$ with $n > 1$. We then have $\langle p \rangle_s = \vec{d}'_1 + \ldots + \vec{d}'_n$ for some $\vec{d}'_1, \ldots, \vec{d}'_n$; notice that $n$ does not change, since the normalization step does not affect the number of directives. By definition of smart lookup and observation (a), we have:

$$\forall i \in \{1, \ldots, n\} : \langle p \rangle_s \Downarrow t \sqsubseteq \vec{d}'_i \Downarrow t. \tag{1}$$

We then use observation (b) to show $\langle |p| \rangle_s = |\langle p \rangle_s| = |\vec{d}'_1 + \ldots + \vec{d}'_n| = \vec{d}'_1, \ldots, \vec{d}'_n$ by definition of linearisation. By definition of smart lookup and observation (c), we have:

$$\exists j \in \{1, \ldots, n\} : \langle |p| \rangle_s \Downarrow t = \vec{d}'_j \Downarrow t. \tag{2}$$

By combining equations (1) and (2), we establish $\langle p \rangle_s \Downarrow t \sqsubseteq \langle |p| \rangle_s \Downarrow t$ as desired. □

## 4.4 Blocking Inline Elements

A central design choice of CSP is that inline scripts are disabled unless otherwise specified, for instance by using `'unsafe-inline'` [25]. However, we observed in all the tested browsers a weird, unexpected difference in the treatment of inline scripts between the following two policies:

(1) `img-src www.example.com;`

(2) `img-src www.example.com; default-src *`.

Our experiments revealed that the first policy allows the execution of inline scripts, while the second one does not, despite the fact that the default sources for script inclusion must be set to the wildcard * in both cases and * does not white-list inline scripts. This mismatch is potentially confusing for web developers and not compliant with the CSP specification. More generally, we observed that any policy which lacks both a `script-src` directive and a `default-src` directive unexpectedly allows the execution of inline scripts.

Fortunately, despite our initial concerns, the security import of this unexpected behaviour is minor, since neither of the two policies puts any restriction on the set of URLs white-listed for script inclusion. This means that an attacker does not really need to inject an inline script to attack a website deploying any of the two policies above, which are equally vulnerable: indeed, under both policies, arbitrary script injection could be performed by first hosting a malicious script on an attacker-controlled website and then injecting a script tag loading the script in the target web page.

We can formally prove this claim. The idea is again to define an operator which transforms policies so that the incorrect behaviour implemented by web browsers is hard-coded in the syntax of the policy as follows.

*Definition 4.3 (Default Extension).* Given a policy $p$, we define its *default extension* $p^\sharp$ as:

$$p^\sharp = \begin{cases} \vec{d}, \texttt{default-src}\ \{*, \texttt{unsafe-inline}\} & \text{if } p = \vec{d} \\ p_1^\sharp + p_2^\sharp & \text{if } p = p_1 + p_2 \end{cases}$$

The following theorem formalizes that the adoption of the default extension does not make more policies vulnerable to script injection, since it only forces the attacker to choose a different attack vector to circumvent already vulnerable policies.

Theorem 4.4 (Assessing Default Extension). *For all policies $p$, subjects $s$ and web attackers $W[H,I]$, we have $p \vdash s \curvearrowleft_{\mathtt{script}} W[H,I]$ iff $p^{\sharp} \vdash s \curvearrowleft_{\mathtt{script}} W[H,I]$.*

Proof. Let $p \vdash s \leftarrowtail_{\mathtt{script}} O$ and $p^{\sharp} \vdash s \leftarrowtail_{\mathtt{script}} O'$, we show that $O \cap W[H,I]_{\mathtt{script}} \neq \emptyset$ if and only if $O' \cap W[H,I]_{\mathtt{script}} \neq \emptyset$. The proof is by induction on the structure of $p$:

(1) If $p = \vec{d}$ for some $\vec{d}$, we distinguish two cases. If $p$ already contains a `script-src` directive or a `default-src` directive, we have $O = O'$ and the conclusion is immediate. Otherwise, we have $p \downarrow \mathtt{script} = \{*\}$ and $p^{\sharp} \downarrow \mathtt{script} = \{*, \mathtt{unsafe\text{-}inline}\}$. We then observe that $p \downarrow \mathtt{script} = p \Downarrow \mathtt{script} = \langle p \rangle_s \Downarrow \mathtt{script}$ and $p^{\sharp} \downarrow \mathtt{script} = p^{\sharp} \Downarrow \mathtt{script} = \langle p^{\sharp} \rangle_s \Downarrow \mathtt{script}$, since the normalization step does not introduce new directives. Let $\{*\} \rightsquigarrow_s L$ and $\{*, \mathtt{unsafe\text{-}inline}\} \rightsquigarrow_s L'$, we have $O = \{o \in \mathcal{O}_{\mathtt{script}} \mid \pi_1(o) \in L\}$ and $O' = \{o \in \mathcal{O}_{\mathtt{script}} \mid \pi_1(o) \in L'\}$ by Lemma 3.7. Pick any $str \in H$, we have $(\mathtt{http}, str) \in L$, hence there must exist an object $o = ((\mathtt{http}, str), \mathtt{attack}) \in \mathcal{O}_{\mathtt{script}}$ by Assumption 3. Similarly, pick any $str' \in N$, we have $(\mathtt{inl}, str') \in L'$, hence there must exist an object $o' = ((\mathtt{inl}, str'), \mathtt{attack'}) \in \mathcal{O}_{\mathtt{script}}$ by Assumption 3. Since $o \in O \cap W[H,I]_{\mathtt{script}}$ and $o' \in O' \cap W[H,I]_{\mathtt{script}}$, we get $O \cap W[H,I]_{\mathtt{script}} \neq \emptyset$ and $O' \cap W[H,I]_{\mathtt{script}} \neq \emptyset$;

(2) If $p = p_1 + p_2$ for some $p_1, p_2$, we have $p^{\sharp} = p_1^{\sharp} + p_2^{\sharp}$. Let $p_1 \vdash s \leftarrowtail_{\mathtt{script}} O_1$ and $p_2 \vdash s \leftarrowtail_{\mathtt{script}} O_2$. Also, let $p_1^{\sharp} \vdash s \leftarrowtail_{\mathtt{script}} O_1'$ and $p_2^{\sharp} \vdash s \leftarrowtail_{\mathtt{script}} O_2'$. By induction hypothesis, we have $O_1 \cap W[H,I]_{\mathtt{script}} \neq \emptyset$ iff $O_1' \cap W[H,I]_{\mathtt{script}} \neq \emptyset$ and $O_2 \cap W[H,I]_{\mathtt{script}} \neq \emptyset$ iff $O_2' \cap W[H,I]_{\mathtt{script}} \neq \emptyset$. We now observe that $O = O_1 \cap O_2$ and $O' = O_1' \cap O_2'$ by definition, hence $O \cap W[H,I]_{\mathtt{script}} \neq \emptyset$ iff $O' \cap W[H,I]_{\mathtt{script}} \neq \emptyset$ by the induction hypothesis.

□

### 4.5 Scheme Relaxation

The `'self'` source expression identifies the origin of the web page deploying a content security policy. Since web origins are defined as triples including a scheme, a hostname and a port number [3], a directive like `img-src 'self'` enforced at `http://a.com` should only allow the inclusion of images from `a.com` over HTTP. We noticed that only Microsoft Edge and Safari strictly follow the CSP specification when interpreting `'self'`. Mozilla Firefox and Chromium are instead more liberal, since the previous directive actually allows the inclusion of images from `a.com` over both HTTP and HTTPS. We observed that Mozilla Firefox and Chromium implement this scheme relaxation also in other cases, i.e., any origin with an HTTP scheme in a directive also allows the inclusion of contents served over HTTPS from the same domain.

Though it is not mentioned in the CSP specification, the scheme relaxation mechanism implemented in Mozilla Firefox and Chromium looks perfectly sensible, since it is secure and more convenient for writing policies. Indeed, we noticed that this more liberal behaviour is recommended in the current draft of CSP Level 3 [32].

## 5 ANALYSIS OF CSP DEPLOYMENT

To evaluate the deployment of CSP and investigate the trends in its adoption, we performed weekly crawls of the homepages of the Alexa Top 1M [2] websites from March 2016 to August 2016, collecting their content security policies.

## 5.1 Methodology

We accessed the homepage of each website using both HTTP and HTTPS, and we collected the content security policies received in the corresponding HTTP(S) responses. We then implemented a policy transformation procedure, which replaces variable policy elements like nonces and report URIs with fixed placeholders and sorts directive names in alphabetical order, and we ran a deduplication procedure on the transformed data. We finally built a dataset containing the first policy in enforcement mode and the first policy in report-only mode sent by each website, if any. (The other collected policies are used in Section 7.)

An important caveat applies to our dataset: since content security policies are deployed per-page and we only crawled the homepages of the websites, it is possible that we missed policies deployed on internal pages, e.g., used to protect private areas, or located at sub-domains. However, being more comprehensive would require a significant engineering effort and the creation of personal accounts at the crawled websites, a process which is notoriously hard to automate [6].

## 5.2 Current Adoption of CSP

Overall, we found 10,684 distinct content security policies in 16,353 websites. We only found a dozen websites defining their policies via meta elements, while all the other websites used CSP headers. The policies are divided as follows: 2,505 policies in enforcement mode and 8,179 policies in report-only mode. Out of the 16,353 websites, we found 10,310 websites running CSP in enforcement mode and 10,729 websites using the report-only mode of CSP; we thus have 4,686 websites implementing both modes, most of which are affiliated to the popular blogging service Blogger. Though the existence of such websites may be unexpected, combining enforcement and report-only mode is actually encouraged by the CSP specification as a good way to enforce a relatively weak policy, while monitoring the possibility of enforcing a stricter one.

It is interesting to observe that an earlier study [35] conducted in March 2014 identified only 850 websites using CSP in the Alexa Top 1M, hence the CSP adoption has significantly expanded in the last two years, approximately of a ten factor. An inspection of our dataset shows that a number of popular hosting services have deployed CSP nowadays, including Blogger, Tumblr and Shopify among others. This justifies such a significant increase of the CSP popularity.

## 5.3 Common Practices in CSP Adoption

### *5.3.1 Unsafe Inline and Unsafe Eval.*

Web developers are strongly encouraged to remove all the inline scripts from their websites to reap the biggest benefits out of CSP and limit the risks of XSS [36]. However, previous studies assessed that moving inline scripts to external resources is not a trivial task [34] and showed that the large majority of the websites deploying CSP just resorts to activating `'unsafe-inline'` [35]. Nonces and hashes have been introduced in CSP Level 2 to give web developers the possibility of white-listing individual inline scripts and stylesheets. These mechanisms were designed to simplify a large-scale adoption of CSP and it is important to understand whether or not they have been successful so far in replacing the insecure `'unsafe-inline'`. We only focus on the 2,505 policies in enforcement mode, since for them we can safely assume a deliberate and fully-aware adoption of CSP, which is not obvious for report-only policies.

Out of 2,505 policies, 1,664 include `'unsafe-inline'` in a `script-src` directive (66.4%) without making use of nonces or hashes. Only 4 policies employ hashes to white-list their inline scripts (0.2%), while 38 policies rely on nonces (1.5%). This shows that the majority of the web developers still enables arbitrary inline scripts in their policies and does not use the new tools available in CSP Level 2 to white-list individual inline scripts, although they were designed to minimise code changes to existing websites and simplify the CSP adoption. Nonces appear to be more popular than hashes in the wild, most likely because they are easier to deploy: a single nonce can be used to white-list all the inline elements of a web page and nonces do not need to be changed when the code of a white-listed inline script is updated. Somewhat similar findings apply to stylesheets: 1,578 policies include `'unsafe-inline'` in a `style-src` directive (63.0%) without including nonces or hashes, while only 2 policies (0.1%) use hashes to white-list stylesheets and none relies on nonces. Notice the drop in popularity of nonces with respect to scripts, probably because the threats posed by inline styles are typically less serious than those posed by inline scripts, though practical attacks based on stylesheet injection have been reported in the past [10].

Finally, we found 1,621 policies (64.7%) including `'unsafe-eval'` in a `script-src` directive and 136 policies (5.4%) including `'unsafe-eval'` in a `style-src` directive. This suggests that the majority of the websites using CSP still resorts to dynamically transforming strings into code for generic reasons, despite the well-recognized dangers of this programming practice.

*5.3.2 Use Cases of CSP.* The original goal of CSP is defining “restrictions that give web application authors control over the content embedded on their site” [25]. However, the CSP specification has recently evolved to include features which are orthogonal to content restriction and it seems that these recent additions to CSP are extremely popular among web developers. In particular, we observed that only 3,832 out of 10,310 websites running CSP in enforcement mode (37.2%) are actually using CSP to implement some form of content restriction, i.e., their policies contain at least one directive of the form *t*`-src`. The remaining 6,478 websites essentially use just the following features of CSP:

- `upgrade-insecure-requests`: this newly proposed directive is not present in the CSP specification, but it is an official addition to the standard [31]. The directive asks web browsers to upgrade to HTTPS a number of HTTP requests sent by CSP-protected web pages, so as to simplify their full transition to HTTPS while avoiding mixed content error. Out of 6,478 websites which do not use CSP for implementing content restrictions, we found 4,985 websites (77.0%) whose content security policy only includes the `upgrade-insecure-requests` directive. The majority of these websites is hosted by Blogger;
- `frame-ancestors`: this directive is used to implement frame busting by giving control on whether browsers should be allowed to embed a CSP-protected web page inside other documents by means of iframes. Out of 6,478 websites which do not use CSP for implementing content restrictions, we found 915 websites (14.1%) using CSP just to implement frame busting. These websites deploy very simple content security policies like `frame-ancestors 'self'`, which restricts framing to same-origin pages.

## 6 MISCONFIGURATION OF CSP POLICIES

To evaluate whether web developers can correctly write useful content security policies, we performed an in-depth analysis of the policies collected from the Alexa Top 1M [2], looking for different types of misconfigurations.

### 6.1 Methodology

Systematically detecting misconfigurations in content security policies is challenging, as one needs to define a meaningful notion of misconfiguration, independent of the specific use case and which does not require to speculate on whether web developers actually enforced what they wanted to enforce. We focus on five classes of inadequate configurations:

(1) *typos and negligence*: these policies include trivial syntactic errors in their specification. In these cases it is completely clear what web developers wanted to enforce, but they specified it incorrectly, e.g., the name of a directive was misspelled;

(2) *ill-formed policies*: these policies have an unclear intended meaning, e.g., they contain contradictory or unexpected information;
(3) *lack of reporting*: these policies do not leverage the monitoring facilities of CSP and do not report the presence of CSP violations to web developers. This may lead to policy problems being undetected for a long time;
(4) *harsh policies*: these policies are too strict and trigger CSP violations upon normal navigation of the website;
(5) *vulnerable policies*: these policies are too liberal and void the benefits of CSP, since they are vulnerable to arbitrary script injection by web attackers.

We defined these classes of problems after a preliminary manual investigation of our dataset and we devised appropriate queries to automatically catch them in all the websites we visited. For the first three classes of misconfigurations, we focused on the dataset including only the first policies delivered by each website during our weekly crawls (10,684 policies). For the last two classes of problems, which are specific to content restriction, we only focused on the subset of the policies in enforcement mode which actually restrict content inclusion in some way, i.e., they contain at least one directive of the form $t$-`src` (2,130 policies).

### 6.2 Typos and Negligence

Syntactic errors in content security policies are very easy to catch and fix, but their import on security is significant, because all major web browsers ignore unknown directives and just output a warning in the JavaScript console, which may go unnoticed. If web developers are not careful enough, they may deploy unexpectedly weak content security policies on their websites.

In our analysis, we found 14 content security policies containing unknown directives, due to obvious typos like:

```
defalt-src 'self'
nfont-src www.myfonts.com
report-uri/csp-report
```

We clarify the security import of these kinds of trivial mistakes: the typo in the first directive leads to the `default-src` directive being missing from the policy, actually white-listing every website as a valid provider of contents without an explicit directive in the policy. Similar considerations apply to errors like the second one, which allows browsers to load fonts from any website (assuming the absence of a stricter default directive). Errors like the third one prevent the generation of CSP violation reports, which may lead to attacks and policy issues going undetected for a long time.

We also noticed 20 content security policies including formatting errors, e.g., misusing punctuation symbols next to directives or erroneously including CSP header information. A few representative examples are:

```
"default-src 'self'; ..."
default-src: 'self'
default-src='self'
Content-Security-Policy default-src 'self';
```

All these cases lead to (a portion of) the content security policy being skipped by the browser and not correctly enforced, with the risks described above.

Misquoting special source expressions like `’self’` or missing the terminating colon when writing a scheme like `http:` is another kind of error, resulting in the white-listing of a non-existing host. This may lead to the deployment of content security policies which are more restrictive than intended. The impact of these errors on security is thus limited, though they may lead to severe usability issues for website users: for instance, white-listing `self` rather than `’self’` prevents the browser from loading same origin contents on a CSP-protected web page. Notice that this may even convince uncaring web developers to abandon CSP to prevent further usability issues. We found 40 policies with such errors in source expressions.

## 6.3 Ill-Formed Policies

We noticed a number of content security policies with an unclear meaning or using the CSP directives in an unexpected way. These cases are typically hard or even impossible to fix without contacting the original authors of the policies, since it is unclear what they wanted to enforce. For instance, we identified 8 content security policies with the following format:

```
script-src a.com b.com; c.com
```

There are at least two legitimate interpretations for incorrect policies like this one. The first possibility is that `c.com` should be actually part of the `script-src` directive: it is plausible that the web developer included this source expression after the semicolon by accident. The second possibility, instead, is that the developer forgot to insert, or accidentally erased, the name of the directive preceding `c.com`. Interestingly, the first error would make the policy more restrictive than intended, while the second error could also make it more liberal, e.g., in the absence of a `default-src` directive.

We also found 50 websites whose content security policy just contains the character `*`. This was surprising, since such a policy does not contain any directive and it is ignored by web browsers. The quirk was readily explained when we realised that all the 50 websites were developed using ASP.NET, so this is likely a default behaviour implemented by the web framework when CSP support is not properly configured. Moreover, we identified 62 websites sending an empty content security policy. We believe this may also be connected to the use of a web development framework, but it is also possible that the policy writers believed to get a few basic security benefits just by activating CSP, for instance assuming that an empty policy prevents the execution of inline scripts as it would be mandated by the CSP specification. Unfortunately, recall that this is not the case in the browsers we tested and inline scripts are not blocked under an empty policy.

Finally, we found 22 content security policies repeating the same directive (e.g., `script-src`) multiple times. In these cases, all the browsers we tested enforce the first occurrence of the directive and ignore the other ones, as dictated by the CSP specification. It is unclear whether web developers are really aware of how web browsers enforce such policies and why repeated directives are not just removed, so it is legitimate to deem these cases at least as bad practices.

## 6.4 Lack of Reporting

We assessed the adoption of the `report-uri` directive to collect CSP violation reports. This is important, since violations to content security policies without this directive are only logged in the JavaScript console and are much more difficult to systematically detect for web developers, because all the violations triggered by website users are lost[3]. We observed that only 706 out of 2,505 policies in enforcement mode (28.2%) specify a `report-uri` directive, hence most websites do not implement a robust monitoring of their content security policies. This is surprising, since it is very easy to activate the reporting facilities of CSP and to parse the violation reports.

We also found 51 policies in report-only mode which do not include the `report-uri` directive. This is a very small fraction (0.6%) of the report-only policies we collected, but these cases are

[3] In principle, it would be possible to catch these violations by registering JavaScript listeners for the SecurityPolicyViolation event [29]. A manual investigation of a subset of the crawled websites, however, suggests that this is far from a common practice: we were not able to find listeners for these events in any of the websites we inspected.

particularly strange to us, since the lack of `report-uri` significantly reduces the benefits of reporting and questions the purpose of these policies.

### 6.5 Harsh Policies

We developed a Chromium extension which intercepts the CSP headers of incoming HTTP(S) responses and sets the `report-uri` directive so that any CSP violation report is redirected to a web server under our control. We then used Selenium to drive Chromium into navigating all the websites deploying CSP to implement content restrictions, using the extension to automatically detect CSP violations triggered when accessing the homepage of these websites. Surprisingly, we observed that 554 out of 3,832 websites (14.5%) trigger at least one CSP violation when their homepage is accessed by our crawler. Notice that this is a subset of the real violations which may be triggered upon navigation, since the crawler does not exercise any website functionality besides page loading. It is interesting to note that 415 of these websites (74.9%) do not use the `report-uri` directive to collect CSP violation reports, so it is perfectly plausible that these violations went unnoticed by web developers. Overall, we collected 960 violation reports: we summarize the reasons for the violations in Table 6.

Table 6. Summary of CSP Violations

| *Type of Violation* | *#Violations* | *#Websites* |
|---|---|---|
| Presence of inline scripts | 12 | 9 |
| Presence of inline styles | 88 | 80 |
| Invocation to `eval` | 6 | 6 |
| Presence of `data:` or `blob:` URIs | 43 | 32 |
| Unexpected HTTP(S) inclusion | 811 | 442 |

We observed 12 inline scripts blocked by CSP in 9 websites. Most of these scripts are related to advertisement or other third-party functionalities injected in the web pages, like site metrics. Interestingly, we also found 88 inline styles blocked by CSP in 80 websites. After a manual investigation of these cases, we noticed they are due to a high number of tiny styles applied to single page elements, e.g., to draw borders around text boxes, which probably went unnoticed.

We then found 6 websites where a call to `eval` was stopped by CSP. One site used `eval` to invoke `decodeURIComponent` on the base64 encoding of an email address, which is thus not rendered correctly; one site invoked `eval` to populate some global variables needed to apply style elements to the homepage; one other site made use of `eval` to check whether the web browser accessing the site was implementing CSP correctly. The last 3 cases were more involved and harder to understand by code inspection, though we noticed that 2 of them seem to be related to the presence of AngularJS[4].

We also detected 43 violations in 32 websites due to the `data:` or `blob:` source expressions being missing in a directive. Most of these cases are related to images (16 violations) and fonts (23 violations), with probably just minor visual consequences.

We finally performed a more systematic evaluation of the 811 violations fired upon requests for HTTP(S) resources which had not been white-listed in the content security policy. We observed in particular two recurrent patterns, which cover almost half of the violations we encountered. First, we noticed 245 violations (30.2%) in 198 websites which are due to advertisement or tools loaded from websites owned by Google, i.e., whose hostname contains the strings `google`, `gstatic` or `doubleclick`. Part of these violations are due to the fact that `google.com`, often correctly

[4] http://docs.angularjs.org/api/ng/directive/ngCsp

white-listed in the content security policy, actually enforces a redirection to `google.tld`, where `tld` is a national top-level domain. There is no easy way to accommodate this use case in the current CSP specification, since the syntax of policies does not allow source expressions of the form `google.*` [29]. Second, we observed 114 violations (14.1%) in 57 websites due to requests targeted at the same domain of the site or some sub-domain of it. Besides the obvious cases where web developers forgot to include the site domain (or some sub-domain of it) in the content security policy, we noticed two other main reasons for this kind of violations:

(1) HTTPS websites requesting contents over HTTP, despite a strong content security policy which prevents this behaviour. These cases often occur when source expressions like `'self'` or `a.com` are included in the policy, since they only white-list HTTPS contents when deployed on HTTPS pages. Some of these violations have no visible import, since modern browsers already block requests for active contents sent over HTTP from HTTPS pages in accordance with the mixed content policy [30];
(2) websites like `http://www.a.com` which load contents from `http://a.com`, though only `http://www.a.com` is declared as a valid source for content inclusion (or vice-versa). These cases often occur when the policy uses the `'self'` source expression, since `'self'` only white-lists same origin contents, but `http://www.a.com` and `http://a.com` are different origins. The occurrence of these violations thus depends on the user typing the `www` prefix or not in the address bar when accessing the website, which is undesirable.

### 6.6 Vulnerable Policies

The last analysis we performed is about the vulnerability of existing content security policies to script injection, based on the theory developed in Section 3.4. Using the syntactic checks in Theorem 3.16, we observed that 1,952 out of 2,130 policies implementing some form of content restriction are vulnerable to script injection (91.6%). We report in Table 7 the main reasons for the vulnerability, based on the syntactic conditions defined in the theorem. The sum is higher than 2,130, since the same policy may satisfy more than one condition.

Table 7. Reasons for Vulnerability to Script Injection

| *Reason for Vulnerability* | *#Policies* | *#Websites* |
|---|---|---|
| `'unsafe-inline'` in `script-src` | 1,654 | 2,367 |
| other liberal src. exp. in `script-src` | 229 | 441 |
| no `script-src` + `'unsafe-inline'` in `default-src` | 221 | 497 |
| no `script-src` + other liberal src. exp. in `default-src` | 129 | 356 |
| no `script-src` + no `default-src` | 62 | 96 |

The majority of the vulnerable policies explicitly disables protection against inline script injection by including `'unsafe-inline'` in `script-src` or `default-src`, without making use of hashes or nonces: this is the case for 1,875 policies, amounting to the 96.1% of the vulnerable policies. This confirms that inline scripts are still pervasive nowadays, despite the fact that their dangers are well-known by web developers.

## 7 EVOLUTION OF CSP DEPLOYMENT

We conducted a series of experiments to estimate how the adoption of CSP and existing content security policies are evolving over time. Our goals were detecting relevant trends and assessing whether web developers keep their content security policies constantly updated.

### 7.1 Methodology

In Section 5 we performed weekly crawls of the homepages of the Alexa Top 1M [2] websites from March 2016 to August 2016, collecting their content security policies. In all the experiments performed so far, however, we only considered the first policy in enforcement mode and the first policy in report-only mode sent by each website, if any. To understand the evolution of the CSP deployment, instead, we performed a set of experiments on the full (deduplicated) dataset of policies to track interesting patterns and trends in policy changes.

To carry out our experiments, we relied on two additional artifacts:

(1) a *policy comparison tool* based on the theory developed in Section 3.5. We used this tool to systematically analyze the effects on permissiveness of the observed policy changes and we make it available online[5];
(2) a *dataset of CSP violations* collected during our weekly crawls. This was built using the Chromium extension presented in Section 6.5 and iterating the same procedure we applied there over the different weeks. We used this dataset to assess whether the observed policy changes were effective at fixing existing CSP violations.

### 7.2 Changes in CSP Adoption

Let $t_1, \ldots, t_n$ be the snapshots of the content security policies collected in our weekly crawls. We say that a website $w$ *commits* to CSP if there exists a crawl $t_i$ such that $w$ does not enforce any policy in $t_1, \ldots, t_{i-1}$, but $w$ enforces a policy in $t_i, \ldots, t_n$. Conversely, a website $w$ *abdicates* from CSP if there exists a crawl $t_i$ such that $w$ enforces a policy in $t_1, \ldots, t_{i-1}$, but $w$ does not enforce any policy in $t_i, \ldots, t_n$. We plot the number of committing and abdicating websites over the different weeks in Figure 1.

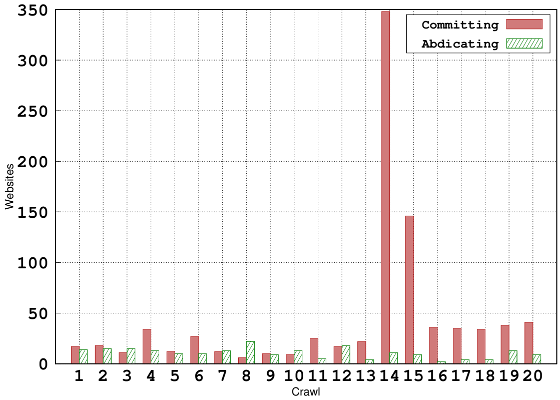

Fig. 1. Committing and Abdicating Websites in 22 Weeks

We observed many more websites committing to CSP rather than abdicating from it during our weekly crawls, which testifies a constant growth in the CSP deployment, especially in the last weeks. Overall, we found 898 committing websites and 213 abdicating websites in the considered timespan, leading to a net result of 685 new websites adopting CSP over 22 weeks. We observed a relevant peak of 348 committing websites in a single week, most of which were related to Tumblr, a well-known micro-blogging platform. Interestingly, we also noticed that 68 abdicating websites

[5] http://www.dais.unive.it/~csp/csp-comparison-tool/

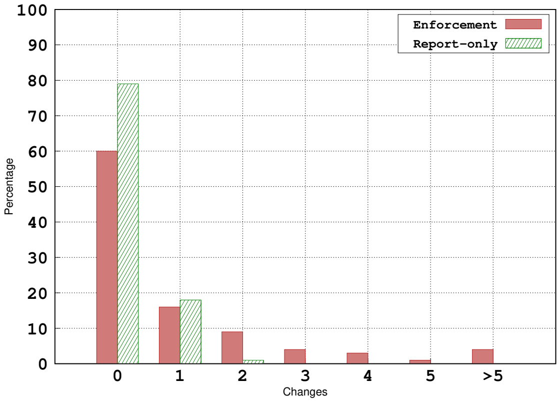

Fig. 2. Changes to Content Security Policies over 22 Weeks

(31.9%) triggered at least one CSP violation during our crawls. We believe that this non-negligible amount of policy violations may quite possibly have influenced the choice of abdicating from CSP, likely due to the challenges of configuring CSP correctly.

Another relevant trend we analyzed in the CSP adoption is the transition from report-only to enforcement mode, which should be the most desirable outcome of a preliminary reporting phase. Overall, we found 52 websites changing their policies from report-only to enforcement mode during our crawls, thus fully embracing CSP, while only 14 websites switched their policies from enforcement mode to report-only. We also found 13 websites moving to report-only just temporarily, most likely to fix issues with their enforced policy. Only 6 websites attempted to enforce a report-only policy at some point, but eventually resorted to switching it back to report-only mode. All these numbers are quite small compared to the size of our study.

## 7.3 Changes in Content Security Policies

*7.3.1 Frequency of Changes.* We evaluated how frequently existing websites change their content security policies. To get uniform and unbiased results, we only focused on the 7,884 websites deploying CSP in all our weekly crawls. Figure 2 reports the distribution (in percentage) of the considered websites with respect to the number of observed policy changes. Though the majority of the websites we analyzed never changed their content security policies in 22 weeks, there is also a significant number of websites which updated their policies at least once. This is the case for 1,032 websites running CSP in enforcement mode (39.9%) and for 1,078 websites running CSP in report-only mode (20.3%). In general, policies deployed in enforcement mode undergo a major number of changes than policies deployed in report-only mode: we believe this is reasonable, because policies in enforcement mode may break website functionality, hence they require a more urgent and frequent maintenance. Moreover, policies in report-only mode may have been deployed just for a preliminary testing or as part of the default configuration of a web development framework, with no further update or maintenance by part of the web developers.

The most surprising cases we observed in our crawls are websites which changed policy basically every week and contribute to populating the tail of the plot: a manual investigation revealed several pornographic websites including apparently random strings as valid hostnames for content inclusion. In these websites, it seems that the same contents are regularly relocated on different domains, possibly due to legal reasons or to the implementation of load balancing techniques.

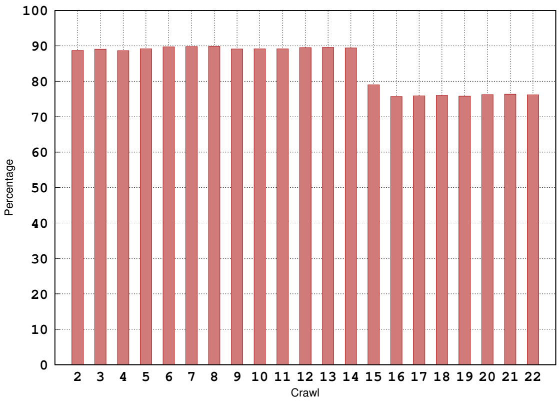

Fig. 3. Policies Vulnerable to Script Injection over 22 Weeks

*7.3.2 Security Improvements.* Another point we wanted to understand is whether web developers are trying to improve their policies by making them robust against script injection attacks. We did not observe significant changes in this respect on the 2,584 websites which enforced CSP during all the weekly crawls: enforced policies which are vulnerable against script injection are not commonly patched to improve their security, for instance by removing `'unsafe-inline'`. However, we observed that the overall percentage of websites whose content security policies are vulnerable to script injection has decreased over time: we show the trend of this percentage in Figure 3.

At the beginning of our weekly crawls, we noticed that the 89.0% of the websites enforcing some kind of content restriction were vulnerable to script injection, while only the 76.2% of such websites turned out to be vulnerable at the end of our crawls. This decrease is mostly due to a major player, Tumblr, which deployed CSP in July. Remarkably, the policies written by Tumblr use nonces to restrict the execution of inline scripts. We believe the introduction of nonces in CSP Level 2 may have encouraged the adoption of the standard by a big company like Tumblr.

*7.3.3 Policy Permissiveness.* We analyzed the general trend of the maintenance operations performed by web developers. Our goal was understanding whether changes to existing content security policies typically make these policies more restrictive, more permissive, equivalent (refactoring) or incomparable (some content inclusions are enabled, other ones are disabled). Overall, we found 6,237 policy changes in our dataset: 4,472 performed on policies in enforcement mode (71.7%) and 1,765 performed on policies in report-only mode (28.3%). The distribution of the effect of the policy changes is shown in Table 8. In the case of policies in enforcement mode, most changes were intended to make policies more permissive (42.4%). This suggests that web developers perform policy maintenance operations mostly to enable blocked functionalities, rather than to improve security. In the case of policies in report-only mode, instead, there is an abundance of interventions which make policies incomparable (60.4%). This is likely due to the fact that these policies are under active development and monitoring, and less refined than policies which are enforced on production websites. Interventions which make policies more restrictive are generally uncommon for both kind of policies.

We also looked for websites with a *monotonic* behaviour, i.e., whose policy changes were always aimed at making policies more permissive or more restrictive. Among 2,584 websites running CSP in enforcement mode in all the weekly crawls, we found 520 websites whose policies became

Table 8. Effect of Policy Changes

| *Effect of Intervention* | *Enforcement* | | *Report-only* | |
|---|---|---|---|---|
| More restrictive | 125 | 2.8% | 68 | 3.9% |
| More permissive | 1,896 | 42.4% | 144 | 8.1% |
| Equivalent | 733 | 16.4% | 487 | 27.6% |
| Incomparable | 1,718 | 38.4% | 1,066 | 60.4% |
| Total | 4,472 | 100.0% | 1,765 | 100.0% |

consistently more permissive during our weekly crawls (20.1%), while only only 10 websites always made their policies stricter over time (0.4%). As to the 5,300 websites running CSP in report-only mode in all the weekly crawls, we found less widespread monotonic behaviours: 21 websites always made their policies more permissive, while only 1 website made its policy more restrictive over time. This is due to the fact that changes making policies incomparable are much more common in website running CSP in report-only mode.

*7.3.4 Fixing CSP Violations.* Finally, we looked for correlations between changes to existing content security policies and website functionality being reduced by policy enforcement. We detected 5,072 violations during our crawls which disappeared at some point in time. For these cases, we compared the content security policy $p$ deployed after the disappearance of the violation and a synthetic policy $p_v$ which only white-lists the originally blocked contents: if $p_v \leq p$, the changes performed by the web developers were relevant to fix the violation. We identified only 645 interventions (12.7%) making policies more liberal to enable a blocked functionality, while in 4,427 cases (87.3%) the changes were not related to the violations we collected. It is interesting to note that the very large majority of the violations disappeared though the underlying content security policy was not actually changed to prevent them: this is likely due to the dynamic nature of modern websites and to the widespread practice of including advertisement. The volume of these transient violations is worrisome, since it means that it is difficult to keep content security policies constantly updated.

Our last experimental finding is about the existence of *persistent* violations on website running CSP in enforcement mode. We found 1,294 violations in 506 websites being triggered at every visit of our crawler since they were first encountered (and for at least one month). These cases call for policy changes, but apparently web developers are not aware of them or have been unable to fix them correctly. We think that the first possibility is the most likely, since 365 out of the 506 websites (72.1%) do not make use of the `report-uri` directive.

## 8 PERSPECTIVE

As a result of our investigation, we observed a few important classes of problems for CSP in its current form. We discuss them in the following, highlighting recent trends and research proposals which go in the right direction to address them. We believe that solving these problems is paramount to a larger and more effective CSP deployment.

### 8.1 CSP Monitoring

The first class of problems we found comes from a lack (or loss) of useful feedback for web developers when writing content security policies. Though the reporting facilities of CSP are excellent, the large majority of the web developers do not benefit of them, since the 71.8% of the policies in enforcement mode we collected lack a `report-uri` directive. A simple change we propose is making browsers output a warning in the JavaScript console when parsing a policy lacking the `report-uri` directive: none of the browsers we tested provides this warning. We think that recommending the usage of `report-uri` would be very helpful to make web developers aware of the importance of reporting and to prevent the deployment of policies which are too strict to work correctly (Section 6.5). Moreover, we propose that the `report-uri` directive should also be leveraged to collect CSP violation reports whenever unknown directives or ill-formed policies are parsed by the browser. This would be useful to prevent the errors discussed in Sections 6.2 and 6.3. These errors are not widespread, but they are a serious problem in practice, because the syntax of CSP is very liberal and browsers are tolerant when parsing policies for the sake of backward compatibility. Reporting these issues while enforcing the well-formed portion of the policy could be a solution which combines backward compatibility with a better assistance for web developers.

### 8.2 Inline Elements

Unfortunately, the second class of problems affecting CSP is more rooted into its design. Banning inline scripts is certainly important to mitigate code injection, but too many web developers still activate `'unsafe-inline'` on their web pages: this is the case for the 88.0% of the policies implementing some form of content restriction. Nonces and hashes are a step in the right direction, but their adoption is minuscule: roughly, only the 1.5% of the websites running CSP in enforcement mode use nonces or hashes. Moreover, inline scripts are not the only attack vector for code injection: the 16.8% of the policies enforcing some content restriction directly includes a liberal source expression different from `'unsafe-inline'` in its white-list for script inclusion.

### 8.3 White-Lists

This leads us to the more general observation that white-lists require web developers to strike a very delicate balance between security and usability. Carefully designed white-lists are difficult to write and to maintain, as testified by the large number of CSP violations we encountered on existing websites: as a result, web developers resort to white-listing liberal source expressions to prevent functionality issues. It seems that security researchers have different feelings on this important problem: a recent study by [33] questioned the security of white-lists and proposed a full transition to nonce-based policies, while other efforts like CSPAutoGen [22] and the Mozilla Laboratory[6] aim at developing tools for synthesizing automatically accurate content security policies based on observable browser behaviours.

### 8.4 Dynamic Nature of the Web

A very delicate issue we observed is that CSP violations are often due to elements which are not totally under the control of the author of the content security policy. In our analysis, we noticed that redirects and advertisement systems are particularly troublesome in this respect. Redirects trigger security violations when a white-listed origin forces a redirection to an origin which is not white-listed. Advertisement systems, instead, have a very dynamic and unpredictable behaviour which hardly fits the nature of a white-list, hence they end up triggering transient security violations. In very recent work, we proposed Compositional CSP, an extension of CSP designed to tackle these issues by assembling content security policies at runtime in the browser [5]. The core idea of Compositional CSP is to start from a simple content security policy which only includes static dependencies which are easy to identify for page developers, while giving to the providers of the imported contents the ability to relax the initial policy to support their dynamic behaviour.

[6] https://addons.mozilla.org/en-US/firefox/addon/laboratory-by-mozilla/

## 9 RELATED WORK

### 9.1 CSP Semantics

In concurrent independent work, Liu et al. [16] formalized a core of the CSP 1.0 semantics. The authors used the semantics to reason on policy permissiveness and to design algorithms for removing redundant information from content security policies. However, they did not use the semantics to draw conclusions on the current state of the CSP deployment and to reason on the security of existing practices and implementations, which is the distinguishing feature of the present paper. It is also worth mentioning that their semantics is not as comprehensive and accurate as ours. Specifically, it does not represent inline elements and the corresponding source expressions available in CSP, like `'unsafe-inline'` and hashes. Also, it does not support the conjunction of policies, source expressions without an explicit scheme and the `'self'` source expression: these features are commonly used by existing websites and make the permissiveness analysis more complicated to formalize.

### 9.2 CSP Deployment

Patil and Braun [23] presented an analysis of the CSP adoption on the Alexa Top 100k in October 2013. After assessing a limited adoption of the standard, the authors proposed a tool, UserCSP, to automatically synthesise content security policies for existing websites. This is not the only available study, because Weissbacher, Lauinger and Robertson [35] proposed a more recent and in-depth analysis of the CSP deployment on the Alexa Top 1M in March 2014. The authors then discussed challenges in the CSP adoption and techniques for semi-automated policy generation.

There are many important differences between the present paper and this previous work [23, 35]. First, the focus of the works is quite different, since we are only interested in assessing the trends and the effectiveness of the current CSP adoption, while [23, 35] put great emphasis on semi-automated policy generation. Finding effective ways to generate content security policies is definitely an important and intriguing research challenge, which we plan to pursue in future work. On the other hand, the evaluation of the CSP effectiveness in [23, 35] is not nearly as comprehensive and systematic as ours: [23, 35] do not include any evaluation of the CSP support in existing browsers, nor any analysis of common errors in policy specification. Also the security analysis in [23, 35] is much more limited than ours, less rigorous and not as exhaustive in covering the possible attack vectors for XSS. Only [35] briefly touches on the point of the evolution of CSP, but it is limited to three websites (BBC, CNN, Twitter).

Besides these methodological aspects, there are also good technological reasons motivating further, up-to-date research on CSP. When the studies in [23, 35] were performed, CSP Level 2 did not yet exist, so there is no published research on the latest additions to the CSP standard, e.g., hashes and nonces. Moreover, the adoption of CSP has significantly increased in the last two years: [23] identified only 27 websites running CSP in enforcement mode, [35] found 815 websites, while the present work identified 10,310 websites. Such a larger scale calls for a more systematic evaluation, like the one pursued in the present paper.

Concurrently to the publication of the original version of our study [4], Weichselbaum et al. [33] presented a large-scale analysis of the CSP deployment based on a search engine corpus of around 100 billion pages. The focus of their research is much more specific than ours, namely providing a perspective on the insecurity of white-lists in content security policies. The authors analyzed in particular the presence of JSONP endpoints and libraries for symbolic execution on white-listed hosts, which allow script injection attacks, although the underlying content security policy does not relax the default restrictions of CSP on inline scripts. These attacks are not covered by our research, which only focuses on simpler bypasses directly enabled by the CSP semantics, rather than by the insecurity of white-listed hosts. Their study also analyzed how nonces and the `'strict-dynamic'` source expression can be used to write content security policies which are more robust against the aforementioned attacks. However, the analysis in [33] is more vertical than ours and does not discuss a number of points which are covered by our research, like the browser implementations of CSP, other possible misconfigurations of content security policies and the analysis of the evolution of the CSP deployment in the wild. Moreover, their work does not include any formal analysis.

### 9.3 Other Works on CSP

Van Acker, Hausknecht and Sabelfeld [26] studied the current inability of CSP at preventing data exfiltration attacks. The paper provides empirical evidence that no major web browser implements defenses against data exfiltration in presence of DNS and resource prefetching, even when the strongest content security policy is put in place, and proposes mitigation techniques.

Hausknecht, Magazinius and Sabelfeld [9] focused on the tension between content security policies and browser extensions. Since browser extensions can modify the DOM, they may end up making web pages request external resources which are not white-listed by the underlying content security policy. The paper proposes a mechanism to endorse CSP modifications performed by browser extensions, so as to strike a good balance between security and extensions functionality.

Hothersall-Thomas, Maffeis and Novakovic [11] presented BrowserAudit, a web application implementing a series of more than 400 automated security tests for web browsers. Notably, BrowserAudit also includes a set of 226 tests for CSP 1.0 divided in 10 main families. The compliance tests for CSP implemented in BrowserAudit are simple and quite low-level, likely because the scope of BrowserAudit is not limited to CSP, but rather embraces browser security as a whole.

Johns [13] identified three limitations of CSP leaving room for dangerous code injections: no prevention of insecure server-side assembly of JavaScript code, lack of control over the content of white-listed external scripts, and lack of control over the number and the appearance order of script tags. His paper proposes a framework, called PreparedJS, which complements CSP with solutions (or mitigations) against these attack vectors.

Some, Bielova and Rezk [24] investigated the security import of the delicate interactions between CSP and the Same Origin Policy. If a web page embeds an iframe from its same origin, the SOP does not isolate the iframe and the distinction between the two entities is immaterial. Since iframes can enforce content security policies independently from their embedders, the security of a page embedding a same-origin iframe is downgraded to the security of the most permissive content security policy between those of the embedder and the iframe.

### 9.4 Large-Scale Analysis of the Web

The present paper positions itself in the popular research line of large-scale security evaluations of the Web [27]. Just to mention a few relevant works, previous evaluations focused on other aspects of web security, like remote JavaScript inclusion [19], DOM-based XSS [15], mixed content websites [7], authentication cookies [6] and HSTS [14].

## 10 CONCLUSION

We performed a large-scale, systematic analysis of four key factors to the effectiveness of CSP: browser support, website adoption, correct configuration and constant maintenance. Though browser support is largely satisfactory, with the exception of few notable issues, the other three points present significant shortcomings. The deployment of CSP is still quite limited in practice and, more importantly, there are many errors and weaknesses in existing content security policies, which leave room for security or usability issues. Moreover, content security policies are not regularly updated to ban insecure practices and remove unintended security violations. We argue that many of the problems we found can be fixed by better exploiting the reporting facilities of CSP, but other issues deserve additional research, being more rooted into the CSP design.

Overall, CSP is growing, but not nearly as fast and effectively as desirable. Given the still relatively limited adoption of the standard, this could be an excellent moment for a retrospective look at its design and motivations based on the main observations we collected.

## ACKNOWLEDGMENTS

This research was supported by the MIUR project ADAPT. We thank Daniel Hausknecht, Artur Janc, Sebastian Lekies, Andrei Sabelfeld, Michele Spagnuolo and Lukas Weichselbaum for the many lively discussions about the current state of CSP.

## REFERENCES

[1] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John C. Mitchell, and Dawn Song. 2010. Towards a formal foundation of web security. In *CSF*. 290–304.

[2] Alexa. 2016. Alexa top sites. (2016). http://www.alexa.com/topsites.

[3] Adam Barth. 2011. The web origin concept. (2011). https://tools.ietf.org/html/rfc6454.

[4] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. 2016. Content Security Problems? Evaluating the effectiveness of Content Security Policy in the wild. In *CCS*. 1365–1375.

[5] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. 2017. CCSP: Controlled relaxation of content security policies by runtime policy composition. In *USENIX Security Symposium*.

[6] Stefano Calzavara, Gabriele Tolomei, Andrea Casini, Michele Bugliesi, and Salvatore Orlando. 2015. A supervised learning approach to protect client authentication on the web. *TWEB* 9, 3 (2015), 15.

[7] Ping Chen, Nick Nikiforakis, Christophe Huygens, and Lieven Desmet. 2013. A dangerous mix: large-scale analysis of mixed-content websites. In *ISC*. 354–363.

[8] Matthew Van Gundy and Hao Chen. 2012. Noncespaces: using randomization to defeat cross-site scripting attacks. *Computers & Security* 31, 4 (2012), 612–628.

[9] Daniel Hausknecht, Jonas Magazinius, and Andrei Sabelfeld. 2015. May I? - Content Security Policy endorsement for browser extensions. In *DIMVA*. 261–281.

[10] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. 2014. Scriptless attacks: Stealing more pie without touching the sill. *Journal of Computer Security* 22, 4 (2014), 567–599.

[11] Charlie Hothersall-Thomas, Sergio Maffeis, and Chris Novakovic. 2015. BrowserAudit: automated testing of browser security features. In *ISSTA*. 37–47.

[12] Trevor Jim, Nikhil Swamy, and Michael Hicks. 2007. Defeating script injection attacks with browser-enforced embedded policies. In *WWW*. 601–610.

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

[14] Michael Kranch and Joseph Bonneau. 2015. Upgrading HTTPS in mid-air: an empirical study of strict transport security and key pinning. In *NDSS*.

[15] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: large-scale detection of DOM-based XSS. In *CCS*. 1193–1204.

[16] Shukai Liu, Xuexiong Yan, Qingxian Wang, and Qi Xi. 2016. A systematic analysis of Content Security Policy in web applications. *Security and Communication Networks* (2016). In press.

[17] Mike Ter Louw and V. N. Venkatakrishnan. 2009. Blueprint: robust prevention of cross-site scripting attacks for existing browsers. In *S&P*. 331–346.

[18] Yacin Nadji, Prateek Saxena, and Dawn Song. 2009. Document Structure Integrity: a robust basis for cross-site scripting defense. In *NDSS*.

[19] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You are what you include: large-scale evaluation of remote javascript inclusions. In *CCS*. 736–747.

[20] OWASP. 2013. OWASP Top 10 Threats. (2013). https://www.owasp.org/index.php/Top_10_2013-Top_10.

[21] OWASP. 2017. XSS Prevention Cheat Sheet. (2017). https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet.

[22] Xiang Pan, Yinzhi Cao, Shuangping Liu, Yu Zhou, Yan Chen, and Tingzhe Zhou. 2016. CSPAutoGen: Black-box Enforcement of Content Security Policy upon Real-world Websites. In *CCS*. 653–665.

[23] Kailas Patil and Frederik Braun. 2016. A measurement study of the Content Security Policy on real-world applications. *I. J. Network Security* 18, 2 (2016), 383–392.

[24] Dolière Francis Some, Nataliia Bielova, and Tamara Rezk. 2017. On the Content Security Policy Violations due to the Same-Origin Policy. (2017). To appear.

[25] Sid Stamm, Brandon Sterne, and Gervase Markham. 2010. Reining in the web with Content Security Policy. In *WWW*. 921–930.

[26] Steven Van Acker, Daniel Hausknecht, and Andrei Sabelfeld. 2016. Data exfiltration in the face of CSP. In *ASIA CCS*.

[27] Tom Van Goethem, Ping Chen, Nick Nikiforakis, Lieven Desmet, and Wouter Joosen. 2014. Large-scale security analysis of the web: challenges and findings. In *TRUST*. 110–126.

[28] W3C. 2012. Content Security Policy 1.0. (2012). https://www.w3.org/TR/2012/CR-CSP-20121115/.

[29] W3C. 2015. Content Security Policy Level 2. (2015). https://www.w3.org/TR/CSP2/.

[30] W3C. 2015. Mixed content. (2015). https://www.w3.org/TR/mixed-content/.

[31] W3C. 2015. Upgrade Insecure Requests. (2015). https://www.w3.org/TR/upgrade-insecure-requests/.

[32] W3C. 2016. Content Security Policy Level 3. (2016). https://w3c.github.io/webappsec-csp/.

[33] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. 2016. CSP is dead, long live CSP! On the insecurity of whitelists and the future of Content Security Policy. In *CCS*. 1376–1387.

[34] Joel Weinberger, Adam Barth, and Dawn Song. 2011. Towards client-side HTML security policies. In *HotSec*.

[35] Michael Weissbacher, Tobias Lauinger, and William K. Robertson. 2014. Why is CSP failing? Trends and challenges in CSP adoption. In *RAID*. 212–233.

[36] Mike West. 2015. An introduction to Content Security Policy. (2015). http://www.html5rocks.com/en/tutorials/security/content-security-policy/.

## A PROOFS

### A.1 Proof of Lemma 3.7

We show the three points of the lemma separately. All of them are proved by induction on the structure of the policy $p$:

(1) we first prove that, for all source expressions $se$ and subjects $s$, $\langle se \rangle_s$ is a normal directive value. This is done by observing that all the non-normal source expressions `self`, $str$ and $*.str$ are transformed into a set of normal source expressions by the normalization operator in Definition 3.6, while all the normal source expressions are transformed into the singleton including them.

An immediate corollary of this result is that, for all directive values $v$ and subjects $s$, $\langle v \rangle_s$ is a normal directive value, because $\langle v \rangle_s$ is defined by applying $\langle se \rangle_s$ to any $se \in v$. Now, let $p = \vec{d}$ for some list of directives $\vec{d}$: since $\langle p \rangle_s$ is obtained by applying $\langle v \rangle_s$ to any directive value $v$ occurring in $\vec{d}$, we have that $\langle p \rangle_s$ is normal. If instead $p = p_1 + p_2$ for some $p_1, p_2$, then $\langle p_1 \rangle_s$ and $\langle p_2 \rangle_s$ are normal by induction hypothesis, hence $\langle p \rangle_s = \langle p_1 \rangle_s + \langle p_2 \rangle_s$ is normal, because the conjunction of two normal policies is normal (only directive values are relevant for normality);

(2) the key lemma is that, for all source expressions $se$ and subjects $s$, we have $se \rightsquigarrow_s L$ if and only if $\langle se \rangle_s \rightsquigarrow_s L$. This is proved by a case analysis on the structure of $se$:

- if $se =$ `self`, then $se \rightsquigarrow_s \{\pi_1(s)\}$, $\langle se \rangle_s = \{\pi_1(s)\}$ and $\langle se \rangle_s \rightsquigarrow_s \{\pi_1(s)\}$;
- if $se = str$, then $se \rightsquigarrow_s \{(sc, str) \mid \pi_1(\pi_1(s)) \rhd sc\}$, $\langle se \rangle_s = \{(sc, str) \mid \pi_1(\pi_1(s)) \rhd sc\}$ and $\langle se \rangle_s \rightsquigarrow_s \{(sc, str) \mid \pi_1(\pi_1(s)) \rhd sc\}$;
- if $se = *.str$, then $se \rightsquigarrow_s \{(sc, str') \mid \pi_1(\pi_1(s)) \rhd sc \wedge \exists str'' : str' = str''.str\}$, $\langle se \rangle_s = \{(sc, *.str) \mid \pi_1(\pi_1(s)) \rhd sc\}$ and $\langle se \rangle_s \rightsquigarrow_s \{(sc, str') \mid \pi_1(\pi_1(s)) \rhd sc \wedge \exists str'' : str' = str''.str\}$;
- in all the other cases, we have $\langle se \rangle_s = \{se\}$ and the result is immediate.

An immediate corollary of this result is that, for all directive values $v$ and subjects $s$, we have $v \rightsquigarrow_s L$ if and only if $\langle v \rangle_s \rightsquigarrow_s L$, because $\langle v \rangle_s$ is defined by applying $\langle se \rangle_s$ to any $se \in v$. Now, let $p = \vec{d}$ for some list of directives $\vec{d}$ and let $p \vdash s \leftarrowtail_t O$ for some set of objects $O$. The judgement can only be proved by rule (D-Val), so there exists a directive value $v$ such that $\vec{d} \downarrow t = v$ and $v \rightsquigarrow_s L$ for some $L$ such that $O = \{o \in \mathcal{O}_t \mid \pi_1(o) \in L\}$. We then observe that the same rule must also be used to build the judgement $\langle p \rangle_s \vdash s \leftarrowtail_t O'$ for some set of objects $O'$. However, $\langle \vec{d} \rangle_s \downarrow t = \langle v \rangle_s$ by definition and we proved that $\langle v \rangle_s \rightsquigarrow_s L$, hence we conclude that $O = O'$. If instead $p = p_1 + p_2$ for some $p_1, p_2$, let $p \vdash s \leftarrowtail_t O$ for some set of objects $O$. The judgement can only be proved by rule (D-Conj), so there exist two set of objects $O_1, O_2$ such that $p_1 \vdash s \leftarrowtail_t O_1$, $p_2 \vdash s \leftarrowtail_t O_2$ and $O = O_1 \cap O_2$. By induction hypothesis, we have $\langle p_1 \rangle_s \vdash s \leftarrowtail_t O_1$ and $\langle p_2 \rangle_s \vdash s \leftarrowtail_t O_2$, so we conclude $\langle p \rangle_s \vdash s \leftarrowtail_t O$ by rule (D-Conj), because $\langle p \rangle_s = \langle p_1 \rangle_s + \langle p_2 \rangle_s$;

(3) the key lemma is that, for all normal source expressions *se* and subjects $s_1, s_2$, we have that $se \rightsquigarrow_{s_1} L_1$ and $se \rightsquigarrow_{s_2} L_2$ implies $L_1 = L_2$. This is proved by a case analysis on the structure of *se* and an inspection of the rules in Table 3, observing that the subjects $s_1, s_2$ are only used in the rules for the non-normal source expressions `self`, *str* and $*.str$.
An immediate corollary of this result is that, for all normal directive values $v$ and subjects $s_1, s_2$, we have $v \rightsquigarrow_{s_1} L$ if and only if $v \rightsquigarrow_{s_2} L$. Now, let $p = \vec{d}$ for some list of directives $\vec{d}$ and let $p \vdash s_1 \leftarrowtail_t O$ for some set of objects $O$. The judgement can only be proved by rule (D-Val), so there exists a normal directive value $v$ such that $\vec{d} \downarrow t = v$ and $v \rightsquigarrow_{s_1} L$ for some $L$ such that $O = \{o \in \mathcal{O}_t \mid \pi_1(o) \in L\}$. However, we proved that also $v \rightsquigarrow_{s_2} L$, hence $p \vdash s_2 \leftarrowtail_t O$ by rule (D-Val). If instead $p = p_1 + p_2$ for some $p_1, p_2$, let $p \vdash s_1 \leftarrowtail_t O$ for some set of objects $O$. The judgement can only be proved by rule (D-Conj), so there exist two sets of objects $O_1, O_2$ such that $p_1 \vdash s_1 \leftarrowtail_t O_1$, $p_2 \vdash s_1 \leftarrowtail_t O_2$ and $O = O_1 \cap O_2$. By induction hypothesis, we have $p_1 \vdash s_2 \leftarrowtail_t O_1$ and $p_2 \vdash s_2 \leftarrowtail_t O_2$, so we conclude $p \vdash s_2 \leftarrowtail_t O$ by rule (D-Conj).

## A.2 Proof of Lemma 3.8

Lemma A.1. *Let $se_1, se_2$ be two normal source expressions. If $se_1 \sqsubseteq_{src} se_2$, then for all subjects $s$ we have $se_1 \rightsquigarrow_s L_1$ and $se_2 \rightsquigarrow_s L_2$ with $L_1 \subseteq L_2$.*

Proof. By a case analysis on the derivation of $se_1 \sqsubseteq_{src} se_2$, observing that the subject used to build the sets of locations is immaterial for normal source expressions:

- $sc \sqsubseteq_{src} *$ with $sc \notin \{\texttt{data}, \texttt{blob}, \texttt{fsys}, \texttt{inl}\}$: we have $L_1 = \{l \mid \pi_1(l) = sc\}$ and $L_2 = \{l \mid \pi_1(l) \notin \{\texttt{data}, \texttt{blob}, \texttt{fsys}, \texttt{inl}\}\}$, hence $L_1 \subseteq L_2$;
- $(sc, he) \sqsubseteq_{src} *$ with $sc \notin \{\texttt{data}, \texttt{blob}, \texttt{fsys}, \texttt{inl}\}$: for all the locations $l \in L_1$ we have $\pi_1(l) = sc$ and $L_2 = \{l \mid \pi_1(l) \notin \{\texttt{data}, \texttt{blob}, \texttt{fsys}, \texttt{inl}\}\}$, hence $L_1 \subseteq L_2$;
- $sc \sqsubseteq_{src} (sc, *)$: we have $L_1 = L_2 = \{l \mid \pi_1(l) = sc\}$, hence $L_1 \subseteq L_2$;
- $(sc, he) \sqsubseteq_{src} sc$: for all the locations $l \in L_1$ we have $\pi_1(l) = sc$ and $L_2 = \{l \mid \pi_1(l) = sc\}$, hence $L_1 \subseteq L_2$;
- $(sc, str) \sqsubseteq_{src} (sc, *)$: we have $L_1 = \{(sc, str)\}$ and $L_2 = \{l \mid \pi_1(l) = sc\}$, hence $L_1 \subseteq L_2$;
- $(sc, *.str) \sqsubseteq_{src} (sc, *)$: we have $L_1 = \{l \mid \pi_1(l) = sc \wedge \exists str' : \pi_2(l) = str'.str\}$ and $L_2 = \{l \mid \pi_1(l) = sc\}$, hence $L_1 \subseteq L_2$;
- $(sc, str'.str) \sqsubseteq_{src} (sc, *.str)$: we have $L_1 = \{(sc, str'.str)\}$ and $L_2 = \{l \mid \pi_1(l) = sc \wedge \exists str' : \pi_2(l) = str'.str\}$, hence $L_1 \subseteq L_2$;
- $\texttt{hash}(str) \sqsubseteq_{src} \texttt{unsafe-inline}$: we have $L_1 = \{(\texttt{inl}, str)\}$ and $L_2 = \{l \mid \pi_1(l) = \texttt{inl}\}$, hence $L_1 \subseteq L_2$.

□

Lemma A.2. *Let se be a normal source expression and $v$ be a normal directive value. If there exists a subject $s$ such that $se \rightsquigarrow_s L$ and $v \rightsquigarrow_s L'$ with $L \subseteq L'$, then there exists $se' \in v$ such that $se \sqsubseteq_{src} se'$.*

Proof. By a case analysis on the structure of *se*, using the observation that the number of source expressions occurring in a directive value is finite:

- $se = sc$: we have $L = \{l \mid \pi_1(l) = sc\}$. Since $L \subseteq L'$, we must have one of the following sub-cases:
  - $sc \in v$: we have $sc \sqsubseteq_{src} sc$ by reflexivity;
  - $(sc, *) \in v$: we have $sc \sqsubseteq (sc, *)$ by Table 5;
  - $* \in v$ with $sc \notin \{\texttt{data}, \texttt{blob}, \texttt{fsys}, \texttt{inl}\}$: we have $sc \sqsubseteq *$ by Table 5;
- $se = *$: we have $L = \{l \mid \pi_1(l) \notin \{\texttt{data}, \texttt{blob}, \texttt{fsys}, \texttt{inl}\}\}$. Since $L \subseteq L'$, we must have $* \in v$, but $* \sqsubseteq_{src} *$ by reflexivity;
- $se = (sc, str)$: we have $L = \{(sc, str)\}$. Since $L \subseteq L'$, we must have one of the following sub-cases:
  - $(sc, str) \in v$: we have $(sc, str) \sqsubseteq_{src} (sc, str)$ by reflexivity;
  - $(sc, *.str') \in v$ with $str = str''.str'$: we have $(sc, str) \sqsubseteq (sc, *.str')$ by Table 5;
  - $(sc, *) \in v$: we have $(sc, str) \sqsubseteq_{src} (sc, *)$ by Table 5;
  - $sc \in v$: we have $(sc, str) \sqsubseteq_{src} sc$ by Table 5;
  - $* \in v$ with $sc \notin \{\texttt{data}, \texttt{blob}, \texttt{fsys}, \texttt{inl}\}$: we have $(sc, str) \sqsubseteq_{src} *$ by Table 5;
- $se = (sc, *.str)$: we have $L = \{l \mid \pi_1(l) = sc \wedge \exists str' : \pi_2(l) = str'.str\}$. Since $L \subseteq L'$, we must have one of the following sub-cases:
  - $(sc, *.str) \in v$: we have $(sc, *.str) \sqsubseteq_{src} (sc, *.str)$ by reflexivity;
  - $(sc, *) \in v$: we have $(sc, *.str) \sqsubseteq_{src} (sc, *)$ by Table 5;
  - $sc \in v$: we have $(sc, *.str) \sqsubseteq_{src} sc$ by Table 5;
  - $* \in v$ with $sc \notin \{\texttt{data}, \texttt{blob}, \texttt{fsys}, \texttt{inl}\}$: we have $(sc, *.str) \sqsubseteq_{src} *$ by Table 5;
- $se = (sc, *)$: we have $L = \{l \mid \pi_1(l) = sc\}$. Since $L \subseteq L'$, we must have one of the following sub-cases:
  - $(sc, *) \in v$: we have $(sc, *) \sqsubseteq_{src} (sc, *)$ by reflexivity;
  - $sc \in v$: we have $(sc, *) \sqsubseteq_{src} sc$ by Table 5;
  - $* \in v$ with $sc \notin \{\texttt{data}, \texttt{blob}, \texttt{fsys}, \texttt{inl}\}$: we have $(sc, *) \sqsubseteq_{src} *$ by Table 5;
- $se = \texttt{hash}(str)$: we have $L = \{(\texttt{inl}, str)\}$. Since $L \subseteq L'$, we must have one of the following sub-cases:
  - $\texttt{hash}(str) \in v$: we have $\texttt{hash}(str) \sqsubseteq_{src} \texttt{hash}(str)$ by reflexivity;
  - $\texttt{unsafe-inline} \in v$: we have $\texttt{hash}(str) \sqsubseteq_{src} \texttt{unsafe-inline}$ by Table 5;
- $se = \texttt{unsafe-inline}$: we have $L = \{l \mid \pi_1(l) = \texttt{inl}\}$. Since $L \subseteq L'$, we have $\texttt{unsafe-inline} \in v$, but $\texttt{unsafe-inline} \sqsubseteq_{src} \texttt{unsafe-inline}$ by reflexivity.

□

We can then prove Lemma 3.8 as follows. Let $v_1 = \{se_1, \ldots, se_m\}$ with $se_i \rightsquigarrow_s L_i$ for $i \in \{1, \ldots, m\}$ and $v_2 = \{se'_1, \ldots, se'_n\}$ with $se'_j \rightsquigarrow_s L'_j$ for $j \in \{1, \ldots, n\}$. We show the two points separately:

(1) assume $v_1 \sqsubseteq v_2$ and let $v_1 \rightsquigarrow_s L_1$ and $v_2 \rightsquigarrow_s L_2$ for some $s, L_1, L_2$. By definition, we have $L_1 = \bigcup_{i \in \{1,\ldots,m\}} L_i$ and $L_2 = \bigcup_{j \in \{1,\ldots,n\}} L'_j$. Since $v_1 \sqsubseteq v_2$, we know that for all $se_i \in v_1$ there exists $se'_j \in v_2$ such that $se_i \sqsubseteq_{src} se'_j$. Hence, for all $L_i$ with $i \in \{1, \ldots, m\}$ there exists $L'_j$ with $j \in \{1, \ldots, n\}$ such that $L_i \subseteq L'_j$ by Lemma A.1. This implies $L_1 = \bigcup_{i \in \{1,\ldots,m\}} L_i \subseteq \bigcup_{j \in \{1,\ldots,n\}} L'_j = L_2$;

(2) assume $v_1 \rightsquigarrow_s L_1$ and $v_2 \rightsquigarrow_s L_2$ with $L_1 \subseteq L_2$ for some $s, L_1, L_2$. By definition, we have $L_1 = \bigcup_{i \in \{1,\ldots,m\}} L_i$ and $L_2 = \bigcup_{j \in \{1,\ldots,n\}} L'_j$. Assume by contradiction that $v_1 \not\sqsubseteq v_2$, then there exists $se_k$ with $k \in \{1, \ldots, m\}$ such that $se_k \not\sqsubseteq_{src} se'_j$ for all $j \in \{1, \ldots, n\}$. By Lemma A.2 this implies that $L_k \nsubseteq L_2$, but this is contradictory since $L_k \subseteq L_1 \subseteq L_2$.

## A.3 Proof of Lemma 3.11

Lemma A.3. *Let $se_1, se_2$ be two normal source expressions. If there exists a subject $s$ such that $se_1 \rightsquigarrow_s L_1$ and $se_2 \rightsquigarrow_s L_2$ with $L_1 \cap L_2 \neq \emptyset$, then either $se_1 \sqsubseteq_{src} se_2$ or $se_2 \sqsubseteq_{src} se_1$.*

Proof. By a case analysis on the structure of $se_1$, observing that the subject used to build the sets of locations is immaterial for normal source expressions:

- $se_1 = sc$: we have $L_1 = \{l \mid \pi_1(l) = sc\}$. Since $L_1 \cap L_2 \neq \emptyset$, we must have one of the following sub-cases:
  - $se_2 = sc$: we have $sc \sqsubseteq_{src} sc$ by reflexivity;
  - $se_2 = (sc, he)$: we have $(sc, he) \sqsubseteq_{src} sc$ by Table 5;
  - $se_2 = *$ with $sc \notin \{\texttt{data}, \texttt{blob}, \texttt{fsys}, \texttt{inl}\}$: we have $sc \sqsubseteq_{src} *$ by Table 5;
- $se_1 = *$: we have $L_1 = \{l \mid \pi_1(l) \notin \{\texttt{data}, \texttt{blob}, \texttt{fsys}, \texttt{inl}\}\}$. Since $L_1 \cap L_2 \neq \emptyset$, we must have one of the following sub-cases:
  - $se_2 = sc$ with $sc \notin \{\texttt{data}, \texttt{blob}, \texttt{fsys}, \texttt{inl}\}$: we have $sc \sqsubseteq_{src} *$ by Table 5;
  - $se_2 = (sc, he)$ with $sc \notin \{\texttt{data}, \texttt{blob}, \texttt{fsys}, \texttt{inl}\}$: we have $(sc, he) \sqsubseteq_{src} *$ by Table 5;
  - $se_2 = *$: we have $* \sqsubseteq_{src} *$ by reflexivity;
- $se_1 = (sc, str)$: we have $L_1 = \{(sc, str)\}$. Since $L_1 \cap L_2 \neq \emptyset$, we must have one of the following sub-cases:
  - $se_2 = sc$: we have $(sc, str) \sqsubseteq_{src} sc$ by Table 5;
  - $se_2 = (sc, str)$: we have $(sc, str) \sqsubseteq_{src} (sc, str)$ by reflexivity;
  - $se_2 = (sc, *.str')$ with $str = str''.str'$: we have $(sc, str) \sqsubseteq_{src} (sc, *.str')$ by Table 5;
  - $se_2 = (sc, *)$: we have $(sc, str) \sqsubseteq_{src} (sc, *)$ by Table 5;
  - $se_2 = *$ with $sc \notin \{\texttt{data}, \texttt{blob}, \texttt{fsys}, \texttt{inl}\}$: we have $(sc, str) \sqsubseteq_{src} *$ by Table 5;
- $se_1 = (sc, *.str)$: we have $L_1 = \{l \mid \pi_1(l) = sc \land \exists str' : \pi_2(l) = str'.str\}$. Since $L_1 \cap L_2 \neq \emptyset$, we must have one of the following sub-cases:
  - $se_2 = sc$: we have $(sc, *.str) \sqsubseteq_{src} sc$ by Table 5;
  - $se_2 = (sc, str')$ with $str' = str''.str$: we have $(sc, str') \sqsubseteq_{src} (sc, *.str)$ by Table 5;
  - $se_2 = (sc, *.str)$: we have $(sc, *.str) \sqsubseteq_{src} (sc, *.str)$ by reflexivity;
  - $se_2 = (sc, *)$: we have $(sc, *.str) \sqsubseteq_{src} (sc, *)$ by Table 5;
  - $se_2 = *$ with $sc \notin \{\texttt{data}, \texttt{blob}, \texttt{fsys}, \texttt{inl}\}$: we have $(sc, *.str) \sqsubseteq_{src} *$ by Table 5;
- $se_1 = (sc, *)$: we have $L_1 = \{l \mid \pi_1(l) = sc\}$. Since $L_1 \cap L_2 \neq \emptyset$, we must have one of the following sub-cases:
  - $se_2 = sc$: we have $(sc, *) \sqsubseteq_{src} sc$ by Table 5;
  - $se_2 = (sc, str)$: we have $(sc, str) \sqsubseteq_{src} (sc, *)$ by Table 5;
  - $se_2 = (sc, *.str)$: we have $(sc, *.str) \sqsubseteq_{src} (sc, *)$ by Table 5;
  - $se_2 = (sc, *)$: we have $(sc, *) \sqsubseteq_{src} (sc, *)$ by reflexivity;
  - $se_2 = *$ with $sc \notin \{\texttt{data}, \texttt{blob}, \texttt{fsys}, \texttt{inl}\}$: we have $(sc, *) \sqsubseteq_{src} *$ by Table 5;
- $se_1 = \texttt{hash}(str)$: we have $L_1 = \{(\texttt{inl}, str)\}$. Since $L_1 \cap L_2 \neq \emptyset$, we must have one of the following sub-cases:
  - $se_2 = \texttt{hash}(str)$: we have $\texttt{hash}(str) \sqsubseteq_{src} \texttt{hash}(str)$ by reflexivity;
  - $se_2 = \texttt{unsafe-inline}$: we have $\texttt{hash}(str) \sqsubseteq_{src} \texttt{unsafe-inline}$ by Table 5;
- $se_1 = \texttt{unsafe-inline}$: we have $L_1 = \{l \mid \pi_1(l) = \texttt{inl}\}$. Since $L_1 \cap L_2 \neq \emptyset$, we must have one of the following sub-cases:
  - $se_2 = \texttt{hash}(str)$: we have $\texttt{hash}(str) \sqsubseteq_{src} \texttt{unsafe-inline}$ by Table 5;
  - $se_2 = \texttt{unsafe-inline}$: we have $\texttt{unsafe-inline} \sqsubseteq_{src} \texttt{unsafe-inline}$ by reflexivity.

□

Lemma A.4 (Properties of Meet). *The following properties hold true:*

(1) *For all normal directive values $v_1, v_2$, we have $v_1 \sqcap v_2 \sqsubseteq v_1$ and $v_1 \sqcap v_2 \sqsubseteq v_2$;*
(2) *For all normal directive values $v, v_1, v_2$, if $v \sqsubseteq v_1$ and $v \sqsubseteq v_2$, then $v \sqsubseteq v_1 \sqcap v_2$.*

Proof. We show the two points separately:

(1) Let $se \in v_1 \sqcap v_2$, we show that there exists $se' \in v_1$ such that $se \sqsubseteq_{src} se'$, which proves $v_1 \sqcap v_2 \sqsubseteq v_1$. By definition of $\sqcap$, there are two possibilities:
   - let $se \in \{se \in v_1 \mid \exists se' \in v_2 : se \sqsubseteq_{src} se'\}$. Since $se \in v_1$, the conclusion follows by the reflexivity of $\sqsubseteq_{src}$;
   - let $se \in \{se \in v_2 \mid \exists se' \in v_1 : se \sqsubseteq_{src} se'\}$. Then, there exists $se' \in v_1$ such that $se \sqsubseteq_{src} se'$ by definition of this set.

   The proof of $v_1 \sqcap v_2 \sqsubseteq v_2$ is analogous.

(2) Let $v \sqsubseteq v_1$ and $v \sqsubseteq v_2$. Assume $v \rightsquigarrow_s L$, $v_1 \sqcap v_2 \rightsquigarrow_s L'$, $v_1 \rightsquigarrow_s L_1$ and $v_2 \rightsquigarrow_s L_2$ for some $s, L, L', L_1, L_2$. We show that $L \subseteq L'$, which proves $v \sqsubseteq v_1 \sqcap v_2$ by Lemma 3.8. Let $l \in L$. Since $v \sqsubseteq v_1$ and $v \sqsubseteq v_2$, we have $L \subseteq L_1$ and $L \subseteq L_2$ by Lemma 3.8, hence $l \in L_1$ and $l \in L_2$. However, this is only possible if there exist $se_1 \in v_1$ and $se_2 \in v_2$ such that $se_1 \rightsquigarrow_s L'_1$ and $se_2 \rightsquigarrow_s L'_2$ for some $L'_1, L'_2$ such that $l \in L'_1$ and $l \in L'_2$. By Lemma A.3, this implies that either $se_1 \sqsubseteq_{src} se_2$ or $se_2 \sqsubseteq_{src} se_1$. We thus perform a case distinction:
   - if $se_1 \sqsubseteq_{src} se_2$, then $se_1 \in \{se \in v_1 \mid \exists se' \in v_2 : se \sqsubseteq_{src} se'\} \subseteq v_1 \sqcap v_2$. This implies $L'_1 \subseteq L'$. Since $l \in L'_1$, we conclude $l \in L'$;
   - if $se_2 \sqsubseteq_{src} se_1$, then $se_2 \in \{se \in v_2 \mid \exists se' \in v_1 : se \sqsubseteq_{src} se'\} \subseteq v_1 \sqcap v_2$. This implies $L'_2 \subseteq L'$. Since $l \in L'_2$, we conclude $l \in L'$.

□

Lemma A.5 (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 $v_1 \sqcap v_2 \rightsquigarrow_s L_1 \cap L_2$.*

Proof. Let $v_1 \rightsquigarrow_s L_1$, $v_2 \rightsquigarrow_s L_2$ and $v_1 \sqcap v_2 \rightsquigarrow_s L$ for some subject $s$, we prove $L \subseteq L_1 \cap L_2$ and $L \supseteq L_1 \cap L_2$:

($\subseteq$) let $l \in L$. Since $v_1 \sqcap v_2 \sqsubseteq v_1$ and $v_1 \sqcap v_2 \sqsubseteq v_2$ by Lemma A.4, we have $l \in L_1$ and $l \in L_2$ by Lemma 3.8. This implies that $l \in L_1 \cap L_2$;

($\supseteq$) let $l \in L_1 \cap L_2$, then $l \in L_1$ and $l \in L_2$. Hence, by observing that $\{l\} \rightsquigarrow_s \{l\}$, we get $\{l\} \sqsubseteq v_1$ and $\{l\} \sqsubseteq v_2$ by Lemma 3.8. This implies $\{l\} \sqsubseteq v_1 \sqcap v_2$ by Lemma A.4. Hence, we get $l \in L$ by Lemma 3.8.

□

We are finally ready to prove Lemma 3.11 by induction on the structure of $p$. If $p = \vec{d}$, then $p \Downarrow t = \vec{d} \downarrow t$ and the conclusion is immediate by rule (D-Val). Otherwise, let $p = p_1 + p_2$. By induction hypothesis we have:

$$\begin{array}{c} p_1 \vdash s \leftarrowtail_t \{o \in \mathcal{O}_t \mid \exists L_1 \subseteq \mathcal{L} : p_1 \Downarrow t \rightsquigarrow_s L_1 \wedge \pi_1(o) \in L_1\} \\ p_2 \vdash s \leftarrowtail_t \{o \in \mathcal{O}_t \mid \exists L_2 \subseteq \mathcal{L} : p_2 \Downarrow t \rightsquigarrow_s L_2 \wedge \pi_1(o) \in L_2\} \end{array}$$

By rule (D-Conj), we then have:

$$p_1 + p_2 \vdash s \leftarrowtail_t \{o \in \mathcal{O}_t \mid \exists L_1, L_2 \subseteq \mathcal{L} : p_1 \Downarrow t \rightsquigarrow_s L_1 \wedge p_2 \Downarrow t \rightsquigarrow_s L_2 \wedge \pi_1(o) \in L_1 \cap L_2\}.$$

By Lemma A.5, this leads to:

$$p_1 + p_2 \vdash s \leftarrowtail_t \{o \in \mathcal{O}_t \mid \exists L \subseteq \mathcal{L} : (p_1 \Downarrow t) \sqcap (p_2 \Downarrow t) \rightsquigarrow_s L \wedge \pi_1(o) \in L\}.$$

The conclusion follows by observing that $(p_1 \Downarrow t) \sqcap (p_2 \Downarrow t) = (p_1 + p_2) \Downarrow t$ by definition.