2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P) | 978-1-6654-6512-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/EuroSP57164.2023.00032

# Coverage and Secure Use Analysis of Content Security Policies via Clustering

Mengxia Ren
*Colorado School of Mines*
*Golden, Colorado, USA*
*mengxiaren@mines.edu*

Chuan Yue
*Colorado School of Mines*
*Golden, Colorado, USA*
*chuanyue@mines.edu*

***Abstract*—Content Security Policy (CSP) is a standardized leading technique for protecting webpages against attacks such as Cross Site Scripting (XSS). However, it is often hard to properly deploy CSPs on webpages, and the deployed CSPs often contain security issues or errors. In this paper, we take the unsupervised clustering approach to analyze the security levels of the deployed CSPs from the directive coverage and secure use perspectives. To effectively protect a webpage, a deployed CSP should cover all types of resources needed on the webpage by using different directive names (or some default directive names if available), and should avoid using unsafe directive values which will allow harmful resources to be loaded into a webpage. We implemented a Google Chrome extension, designed policy features, designed a Contrastive Spectral Clustering (CSC) algorithm, and visited the Alexa top 100K websites to analyze the CSPs deployed on them. From the 13,317 homepages that deployed CSPs under the enforcement mode, we categorized their policies into 16 clusters with different characteristics. We found that 15 clusters are at the low level on the coverage and five clusters are at the low level on the secure use of directives; meanwhile, no cluster is at the high level on the coverage of directives, and nine clusters are at the high level on the secure use of directives. These results indicate that most deployed CSPs do not sufficiently protect webpages, and more importantly, clustering helps identify the corresponding common or different reasons from the directive coverage and secure use perspectives. In addition, by analyzing 110,718 subpages of the 13,317 CSP-deployed homepages, we found that most of them deployed the same CSP as in their homepages. Overall, our approach and results can be helpful for promoting the proper deployment of CSPs.**

## 1. Introduction

Content Security Policy (CSP) is a standardized leading technique for protecting webpages against code and data injection attacks especially Cross Site Scripting (XSS) attacks [2], [3], [8]. A webpage is vulnerable to XSS attacks if it allows resources such as JavaScript code or files (abbreviated as scripts) to be loaded from untrustworthy sources to manipulate its content or behavior. As a powerful technique successfully adopted by all major web browsers, CSP provides web application developers with the capability to comprehensively define the policy regarding the permissible resources (e.g., scripts) and behaviors (e.g., form submissions) on each webpage to protect against attacks such as XSS.

A CSP is composed of a set of directives, each of which is a pair of whitespace-delimited ***directive name*** and ***directive value***. The directive name is a non-empty string, while the directive value is a set (may be empty for some directives) of whitespace-delimited strings (referred to as ***directive value tokens*** in this paper). Typically, a directive name specifies a certain resource type that will be controlled, while a directive value specifies which resources can be loaded or what behaviors can be allowed on a webpage. Developers can deploy a CSP for a webpage either through an HTTP response header or an HTML <meta> tag. If deployed in the enforcement mode, a CSP will be enforced at the client side by a web browser. Unfortunately, largely due to the complexity reason, it is often hard to properly deploy CSPs, and the deployed CSPs often contain security issues or errors (Section 2.2).

In this paper, we take the clustering approach to analyze the security levels of the deployed CSPs from the directive coverage and secure use perspectives. To effectively protect a webpage, a deployed CSP should cover all types of resources needed on the webpage by using different directive names (or some default directive names if available), and should avoid using unsafe directive values which will allow harmful resources to be loaded into a webpage. We take the unsupervised clustering approach as it allows us to automatically categorize very diverse and complex CSPs based on the extracted policy features.

We define each ***policy feature*** as a (directive name, directive value token type) pair. Its value is binary, indicating the existence or nonexistence of the feature in a CSP. In the policy feature design, we include the directive name to comprehensively cover all types of directives; we include the directive value at the token type level to capture the specified value neither too coarse-grained nor overly fine-grained; in total, we define and extract 530 policy features.

We design a Contrastive Spectral Clustering (CSC) algorithm for automatically categorizing very diverse and complex CSPs based on the extracted policy features. CSC takes the leading and popular spectral clustering approach that has been theoretically analyzed and experimentally demonstrated as superior to traditional clustering approaches such as K-means [16], [25], [26], [30], [31]. CSC also integrates contrastive learning, a state-of-the-art self-supervised representation learning technique [4], [10], [38], for explicitly learning better (i.e., more informative) representations of CSPs.


DOI 10.1109/EuroSP57164.2023.00032

We implemented a Google Chrome extension and visited the Alexa top 100K websites to analyze the CSPs deployed on them. From the 13,317 homepages that deployed CSPs under the enforcement mode, we categorized their policies into 16 clusters with different characteristics. We found that 15 clusters are at the low level on the coverage and five clusters are at the low level on the secure use of directives; meanwhile, no cluster is at the high level on the coverage of directives, and nine clusters are at the high level on the secure use of directives. These results indicate that most deployed CSPs do not sufficiently protect webpages, and more importantly, clustering helps identify the corresponding common or different reasons from the directive coverage and secure use perspectives. In addition, by analyzing 110,718 subpages of the 13,317 CSP-deployed homepages, we found that most of them deployed the same CSP as in their homepages.

We make four major contributions in this paper: (1) we propose to take the clustering approach for analyzing the security levels of the deployed CSPs from the directive coverage and secure use perspectives; (2) we design and extract 530 policy features based on the latest CSP Level 3 specification; (3) we design a CSC algorithm that leverages the advantages of spectral clustering and contrastive learning for automatically categorizing CSPs; (4) we perform a large-scale measurement study on 100K websites, categorize the CSPs deployed on 13,317 homepages into 16 clusters with different characteristics, and analyze the security levels of the CSPs in each unique cluster to help promote the proper deployment of CSPs.

The rest of this paper is organized as follows. Section 2 reviews the background of CSP and the related work. Section 3 presents the design of our study from the data collection, policy feature extraction, clustering algorithm, and security level analysis perspectives. Section 4 presents and analyzes the clustering results. Section 5 summarizes the high-level takeaways and recommendations for web developers. Section 6 concludes this paper.

## 2. Background and Related work

### 2.1. Background of CSP

CSP is able to control the resource loading, navigation, and script execution environment of a webpage. There are five categories of directives: fetch, document, navigation, reporting, and other directives.

**Fetch Directives.** Fetch directives are used to control the locations from which certain types of resources can be loaded into a webpage. In total, there are 17 types of fetch directives indicated by their unique directive names. For example, the "script-src http://example.com" directive specifies that the scripts from "http://example.com" can be loaded into a webpage for execution. If the "default-src" directive is specified, its value will be considered as the default fetch policy, and all other fetch directives that are not explicitly specified will fall back to it. Other fallback relationships exist among fetch directives according to the CSP specification [15], [33]: e.g., "frame-src" and "worker-src" fall back to "child-src"; "script-src-elem", "script-src-attr", and "worker-src" fall back to "script-src".

**Document Directives.** Document directives control the properties of a webpage document and the script execution environment. There are two types of document directives: "base-uri" and "sandbox". The former is used to control the URLs that can be used in a webpage's <base> element. The latter is used to specify an HTML sandbox policy to be applied to an element or resource.

**Navigation Directives.** Navigation directives control the navigations of a webpage, and there are three types: "form-action", "frame-ancestors", and "navigate-to". The "form-action" directive is used to control the target URLs of a form submission. The "frame-ancestors" directive is used to control the URLs that can embed a webpage through <frame>, <iframe>, <object>, <embed>, or <applet>. The "navigate-to" directive is used to control the URLs to which a webpage can initiate navigations.

**Reporting and Other Directives.** When a resource or a behavior violates the CSP deployed on a webpage, there will be a CSP violation. There are two types of reporting directives ("report-uri" and "report-to") that can be used to specify the hosts or servers to which violation reports will be sent. Other specifications can further extend the core CSP specification, thus there are some other customized directives such as "upgrade-insecure-requests".

CSP has two deployment modes: *enforcement* and *report-only*. In the enforcement mode, CSP violations will trigger enforcement actions (e.g., blocking certain resources or disabling script execution); in the report-only mode, CSP violations will only be reported (typically for web developers to test policies) without incurring enforcement actions. Meanwhile, there are two ways to deploy a CSP for a webpage: via an HTTP response header or an HTML <meta> tag. Developers can deploy a CSP in the enforcement mode via a "Content-Security-Policy" HTTP response header or an HTML <meta> tag, and can deploy a CSP in the report-only mode via a "Content-Security-Policy-Report-Only" HTTP response header. When a webpage response is received, a browser will load resources according to the specified CSP deployment mode and directives.

### 2.2. Related Work

CSP is very complex due to its diverse directive names, infinite directive value space, and subtle semantics (e.g., fallback relationships). This is one major reason why its adoption ratio has always been low, and why it is hard to properly deploy CSPs. Only 1% of the 100 most popular websites deployed the CSP in the enforcement mode as reported by Weissbacher et al. in 2014 [36], only 3.7% (out of 106 billion) URLs carried CSPs as reported by Weichselbaum et al. in 2016 [35], and only 1,206 (out of the top 10K) websites deployed CSPs on their homepages as reported by Roth et al. in 2020 [21]. Roth et al. recruited 12 developers for a study with a semi-structured interview, a drawing task, and a programming task [22]; they found that the complexity of CSP, inconsistencies in the support from browsers and web development frameworks, and insufficient or bad information sources are some major roadblocks for CSP deployment.

Meanwhile, security issues or errors are often found in the deployed CSPs [1], [2], [21], [35]. Weichselbaum et al. found that 94.72% of CSPs could not protect webpages against XSS attacks because of policy misconfigurations and insecure whitelisted entries [35]. Calzavara et al.

found that some CSPs were ill-formed (thus invalid) or too strict [1], and 92.4% of CSP-deployed websites were vulnerable to XSS attacks due to using an 'unsafe-eval' or 'unsafe-inline' directive value; they also backed up their findings by defining a formal semantics for CSP Level 2 [2]. To help deploy better policies, Calzavara et al. further proposed *Compositional CSP*, which extends CSP by incrementally composing a needed policy based on the runtime interaction with a webpage [32]. Roth et al. found that CSP has been increasingly used for other purposes, such as frame control and TLS enforcement, in addition to the traditional XSS defense purpose [21]; besides, they found that insecure practices (especially 'unsafe-eval' or 'unsafe-inline' uses) were present on 90% of 421 websites that deployed CSPs for restricting the script content.

These studies [1], [2], [21], [35] analyzed the overall statistics of CSP security issues based on some specific rules (or heuristics), while we take a clustering approach to automatically categorize CSPs and analyze their common or different security issues. Meanwhile, they mainly analyzed the vulnerabilities of specific directives (e.g., "script-src", "object-src", and "default-src"), but did not estimate the overall or combined protection capabilities of CSPs. For example, Roth et al. coarsely analyzed some specific directives related to script content restriction, TLS enforcement, and framing control [21]; they did not analyze detailed directive values for "frame-ancestors" and did not analyze the combinations of different directive types. A CSP that does not contain a vulnerable (or unsafe) directive may not cover all needed resources, thus leading to the insufficient protection of a webpage. A CSP that contains a vulnerable directive may still be able sufficiently protect a webpage as long as the remaining directives can cover all resources. We analyze the protection capabilities of a CSP based on all its directives.

To promote the CSP adoption, some researchers focused on automatically generating CSPs typically for some specific resources or controls [8], [9], [18]. Fazzini et al. designed AutoCSP to generate a CSP for a webpage by using dynamic taint analysis and identifying resources that should be whiltelisted [9]. Pan et al. designed CSPAutoGen to generate a CSP to protect scripts in webpages by importing inline and dynamic scripts as external scripts [18]. Eriksson et al. designed AutoNav to generate a CSP to control navigations in a website based on creating and analyzing a map of where webpages can navigate to [8]. Our effort and results in this paper could be helpful to the CSP auto-generation research.

In addition, some researchers studied the weaknesses of the CSP standard [8], [20], [28]. Roth et al. found that CSPs can be bypassed by open redirects and script gadgets that would turn non-script data into code [20]. Eriksson et al. found that the "navigate-to" directive may induce several vulnerabilities to a webpage such as resource probing, history sniffing, and bypassing third-party cookie blocking [8]. Somé et al. found that CSP may be vulnerable when embedded iframes from the same origin are contained in a webpage [28]. To strengthen the security of CSP, Somé et al. also proposed four extensions to disallow redirections to partially whitelisted origins, check URL parameters, selectively exclude whitelisted content, and enable efficient feedback reporting [29].

# 3. Design and Methodology

In this section, we describe the design and methodology of our study in terms of data collection, policy feature, clustering algorithm, and security level analysis.

## 3.1. Data Collection Tool and Process

We construct a Google Chrome browser extension to automatically visit websites and collect data. On each website, our extension first visits the homepage. If the homepage deploys a CSP either through an HTTP response header or an HTML <meta> tag, the extension will further randomly selects 10 subpages (with the same domain or subdomain) of the homepage to visit. On each webpage, our browser extension scrolls down to the bottom and stays 60 seconds to collect all HTTP(s) requests and responses, collect CSPs, and save the loaded HTML document. The collected CSPs will be used for policy feature extraction (Section 3.2) and clustering (Section 3.3). The collected HTTP(s) requests and HTML documents will be used for analyzing the resource type coverage of CSPs (Section 3.4).

## 3.2. Policy Feature Design and Extraction

We take the unsupervised clustering approach to categorize CSPs based on our designed policy features. Alternatively, one may consider to categorize CSPs by using pre-defined rules (or heuristics). However, due to the diverse directive names, infinite directive value space, and subtle semantics of CSPs, a ***rule-based approach*** would need much more effort from security experts to manually define rules, would not be able to comprehensively capture the CSP directive use practices (proper or improper ones) in the wild, would need frequent rule updates, and would be error-prone.

Instead, we design policy features to be ***stable*** based on the CSP specification [15], [33], and the clustering approach will automatically categorize very diverse and complex CSPs based on the automatically extracted policy features. Meanwhile, we design our policy features to be ***interpretable***, ***comprehensive***, ***efficient***, and ***extensible***. Recall that we define each *policy feature* as a (directive name, directive value token type) pair, and define its value as binary to indicate the existence or nonexistence of the feature in a CSP. For example, the (script-src, self) pair is a policy feature extracted from the "script-src 'self' " directive that allows the loading of scripts from the same origin sources. Therefore, each policy feature in the feature vector is interpretable at the bit level. In the policy feature design, we include the directive name to comprehensively cover all types of directives; we include the directive value at the token type level and leverage binning whenever appropriate as detailed below to capture the specified value neither too coarse-grained nor overly fine-grained; we design a feature value as binary because binary feature representation is efficient in computation and is popularly used in clustering tasks. In addition, when new directives are introduced to the CSP specification in the future, our policy feature vectors can be easily extended by appending new bits at the end.

Table 1 lists the 28 directive names included in our policy feature design. They cover all the directive names

Table 1: Directive Names and the Allowed Directive Value Types (defined in Table 2) or Values

| *Directive Category* | *Directive Name* | *Allowed Directive Value Types or Values* |
|---|---|---|
| Fetch Directives | default-src | Types I to V |
| | child-src | Types I to V |
| | connect-src | Types I to V |
| | font-src | Types I to V |
| | frame-src | Types I to V |
| | img-src | Types I to V |
| | manifest-src | Types I to V |
| | media-src | Types I to V |
| | prefetch-src | Types I to V |
| | object-src | Types I to V |
| | worker-src | Types I to V |
| | script-src | Types I to V |
| | script-src-attr | Types I to V |
| | script-src-elem | Types I to V |
| | style-src | Types I to V |
| | style-src-attr | Types I to V |
| | style-src-elem | Types I to V |
| Document Directives | base-uri | Types I to V |
| | sandbox | Type VI |
| Navigation Directives | form-action | Types I to V |
| | frame-ancestors | Types I and II, 'self', 'none' |
| | navigate-to | Types I to V |
| Other Directives | block-all-mixed-content | No value is needed |
| | upgrade-insecure-requests | No value is needed |
| | trusted-types | 'none', 'allow-duplicates', policyname |
| | plugin-types | Type VII |
| | require-sri-for | script, style |
| | require-trusted-types-for | 'script' |

Table 2: Directive Value Types and Value Token Types

| *Directive Value Type#* | *Directive Value Type* | *Directive Value Token Types* |
|---|---|---|
| I | scheme-source | 7 in total: https, http, wss, ws, data, blob, other schemes |
| II | host-source | 5 in total: a host-source value is specified with the syntax: *[ scheme-part "://" ] host-part [ ":" port-part ] [ path-part ]*; we bin possible values into five types: *, *.external.domain (*.exdo), *.same.domain (*.sado), external domain (exdo), same domain (sado) |
| III | keyword-source | 8 in total: 'self', 'unsafe-inline', 'unsafe-eval', 'strict-dynamic', 'unsafe-hashes', 'none', 'report-sample', 'unsafe-allow-redirects' |
| IV | nonce-source | 1 in total: 'nonce-<base64-value>' |
| V | hash-source | 3 in total: 'sha256-<base64-value>', 'sha512-<base64-value>', 'sha384-<base64-value>' |
| VI | sandbox values | 15 in total: "", allow-downloads, allow-downloads-without-user-activation, allow-forms, allow-modals, allow-orientation-lock, allow-same-origin, allow-scripts, allow-storage-access-by-user-activation, allow-top-navigation, allow-top-navigation-by-user-activation, allow-pointer-lock, allow-popups, allow-popups-to-escape-sandbox, allow-presentation |
| VII | plugin-types values | 1 in total: all MIME type <type>/<subtype> tokens are binned into one type |
| VIII | customized values | 3 in total: style, script or 'script', 'allow-duplicates' |

of four categories as defined in the CSP Level 3 specification [15], [33]. We do not include the two reporting directive names (Section 2.1) because they are used for sending CSP violation reports and not for directly protecting webpages. Based on the specification, we identify the allowed directive value types or values for each directive name, and list them in the third column of Table 1. Most directive names generally allow multiple types of directive values. For example, "default-src" allows value types from I to V. Some directive names allow certain types of directive values and some specific directive values. For example, "frame-ancestors" allows value types I and II besides two keyword values 'self' and 'none'.

Meanwhile, in our policy feature design we include the directive value at the token type level, and identify 43 directive value token types as shown in the third column of Table 2. There are seven directive value token types for scheme-source, five for host-source, eight for keyword-source, one for nonce-source, three for hash-source, 15 for sandbox values, one for plugins-types values, and three for customized values. There are more than 200 official scheme-source values, so we identify six of them that are often related to security attacks as individual types while binning all the rest as the seventh type "other schemes". Similarly, there are more than 50 MIME type tokens for plugin-types values; we simply bin all of them into a single type because plugins (e.g., java-applet and flash) have generally become deprecated in modern browsers.

Infinite number of possible value tokens exist for host-source, while the nonce values for nonce-source and hash values for hash-source are all in gigantic spaces; therefore, we also leverage the binning techniques to identify a small number of directive value token types for them. In more details, we identify five token types for host-source (roughly from the least restrictive to the most restrictive) as: allowing any host source through a wildcard character '*' (*), whitelisting some external domain source(s) with a domain combination by '*' (*.exdo), whitelisting some same domain source(s) with a domain combination by '*' (*.sado), whitelisting some external domain source(s) without a domain combination (exdo), and whitelisting some same domain source(s) without a domain combination (sado). We bin all nonce values into one token type for nonce-source, and bin hash values into three token types for hash-source based on the currently supported hash algorithms. For keyword-source, sandbox values, and customized values, we simply use their exact values as the token types because they all correspond to a small set of important and well-defined values.

We then pair the 28 directive names (Table 1) with the 43 directive value token types (Table 2) by following the third column of Table 1 to generate our policy features. For directive names that do not need a value to be specified (e.g., "upgrade-insecure-requests"), we only have one feature with an empty directive value token type for each of them. In total, we define and extract 530 (directive name, directive value token type) pairs as the complete set of policy features.

Our policy feature design is comprehensive as it includes all the 28 directive names and all the allowed 43 directive value token types. It is not too coarse-grained because we do not just consider directive names and the presence of an entire directive value. It is not overly fine-grained because we leverage binning techniques to reduce the infinite or gigantic directive value token spaces into a small set of directive value toke types. All the 530 policy features are binary features which will be sufficient for our CSP directive coverage and secure use analysis. In other words, for each CSP-deployed webpage, a 530-dimensional binary feature vector will be extracted to represent its CSP.

Note that multiple CSPs could be deployed (via response headers and/or <meta> tags) for a single webpage, and their directives will be merged by a browser following the strictest directives to protect the webpage based on the W3C specification [15], [33]. With our policy feature design, feature values from multiple CSPs for a single webpage will be simply merged into a single 530-dimensional binary feature vector (by using the binary OR operation) to accurately represent the overall policy of this webpage. Logically, sometimes an AND while sometimes an OR operation should be applied to merge CSPs. For one example, CSP1 contains "script-src 'none' ", while CSP2 contains "img-src 'none' "; a browser will use the OR operation so that the merged CSP will contain both directives. For another example, CSP1 contains "script-src 'none' ", while CSP2 contains "script-src 'self' "; a browser will use the AND operation so that the merged CSP will only contain the strictest "script-src 'none' ". Our policy feature design allows us to programmatically use the OR operation to accurately capture all information in both types of examples into feature vectors as those four directives are different features. The interpretation of a merged policy will be the accurate AND or OR for each feature vector. Thus, the dataset for our feature vectors will be correctly constructed and not biased.

### 3.3. Clustering Algorithm Design

Leveraging our designed policy features, different algorithms can be explored to cluster CSPs. We initially experimented with two popular algorithms, K-means and DBSCAN, implemented in the Scikit-learn machine learning library [24]; K-means performed better than DBSCAN and is more flexible than DBSCAN on selecting different numbers of clusters. We then further explored the leading and popular spectral clustering approach, which embeds the data in the eigenspace of the Laplacian matrix and has been theoretically analyzed and experimentally demonstrated as superior to traditional clustering approaches such as K-means [16], [25], [26], [30], [31]. Meanwhile, recent advances in self-supervised learning especially contrastive learning [4], [10], [38], in which labels are automatically derived from unlabeled examples to train an unsupervised task in a supervised manner, inspired us to explore the integration of contrastive representation learning into spectral clustering for achieving better results.

Specifically, we design a Contrastive Spectral Clustering (CSC) algorithm (Algorithm 1), for automatically categorizing very diverse and complex CSPs based on the extracted policy features. We design a Contrastive Learning (CL) algorithm (Algorithm 2) for explicitly learning better (i.e., more informative) representations of CSPs. CSC is the main algorithm and it calls the CL algorithm at the beginning before performing the spectral clustering. We present the details of these two algorithms in Sections 3.3.1 and 3.3.2, respectively.

**Algorithm 1** Contrastive Spectral Clustering (CSC)

**Input:** $X_{n\times d}$ is a dataset of $n$ $d$-dimensional binary feature vectors
**Input:** $m$ is the dimension of the to be learned representations
**Input:** $\{k_{min},\ ...,\ k_{max}\}$ is a set of numbers of clusters of interest
**Output:** $k_{opt}$ clusters $C_1$, $C_2$, ...., $C_{k_{opt}}$ of examples in X

1: Learn the $m$-dimensional representations of $X$ using Contrastive Learning (CL) (Algorithm 2): $\bar{X}_{n\times m}$ = CL ($X_{n\times d},\ m,\ ...$)
2: Construct a similarity graph $SG$ over $\bar{X}_{n\times m}$ using the nearest neighbors method: $SG$ = Construct_SG ($\bar{X}_{n\times m}$)
3: Construct an affinity matrix: $A_{n\times n}$ = Construct_AM ($SG$)
4: Construct a degree matrix $D_{n\times n}$ based on $A$ using the formula: $D_{i,j} = \begin{cases} \sum_{l=1}^{n} a_{il} \quad where\ a_{il} \in A, & i = j \\ 0, \quad otherwise \end{cases}$
5: Construct the normalized Laplacian matrix $L_{n\times n}$ using $A$, $D$, and $I$ (the identity matrix): $L_{n\times n} = I - D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$
6: Derive the eigenvector matrix $E_{n\times n}$ of $L$ based on the equation: $L = E\Lambda E^{-1}$, where $\Lambda$ is the diagonal matrix of the eigenvalues
7: $k = k_{min}$ //the smallest number of clusters we consider
8: $S = \{\emptyset\}$ //for saving (number of clusters, spectral clustering representations, and clustering evaluation score) 3-tuples
9: **while** $k$++ <= $k_{max}$ **do**
10: $E_k = \{e_1, ..., e_k\}$, where $e_i$ is the $i^{th}$ eigenvector in $E_{n\times n}$
11: $S = S \cup \{(k,\ E_k,$ Score (K-means ($E_k,\ k$)))$\}$
12: $k_{opt}$ = Select_optimal_number_of_clusters ($S$)
13: **return** $k_{opt}$ clusters $C_1$, ...., $C_{k_{opt}}$ = K-means ($E_{k_{opt}}$, $k_{opt}$)

**3.3.1. Contrastive Spectral Clustering (CSC) Algorithm.** An unsupervised clustering problem can be converted to a graph cut problem, in which clustering the examples in a dataset into k clusters can be considered as cutting the similarity graph of the examples into k disjoint subgraphs by removing some edges. Spectral clustering is one way to solve a relaxed graph cut problem through a Laplacian matrix. Spectral clustering does not assume the convexity of the clusters, and this is one main reason why it often outperforms traditional algorithms such as K-means [25], [30], [31].

As shown in Algorithm 1, given a dataset $X$ of $n$ unlabeled examples (i.e., $n$ CSPs represented by their $d$-dimensional binary feature vectors), the dimension of the to be learned representations, and a set of numbers of clusters of interest as the inputs, CSC will automatically return an optimal number of $k_{opt}$ clusters as the result.

In more details, at Line 1, CSC first learns $\bar{X}_{n\times m}$, the $m$-dimensional representations of $X$, using contrastive learning (Algorithm 2). At Line 2, CSC constructs a directed similarity graph $SG$ over the learned $\bar{X}_{n\times m}$ using the standard nearest neighbors method. In this similarity graph, each node is an example and is connected to a

certain number of nodes with the similar features. At Line 3, CSC constructs a symmetric affinity matrix $A_{n\times n}$ by averaging the elements in $SG$ and the transpose of $SG$. This affinity matrix intuitively reflects the mutual similarities between each pair of examples in the dataset. At Line 4, CSC constructs a diagonal degree matrix $D_{n\times n}$ based on the affinity matrix to reflect the overall connectivity level of each example in the dataset. At Line 5, CSC constructs a normalized Laplacian matrix $L_{n\times n}$ using the affinity matrix, the degree matrix, and the identity matrix. Spectral clustering with the normalized Laplacian minimizes the between-cluster similarity and maximizes the within-cluster similarity, while that with the unnormalized Laplacian only minimizes the between-cluster similarity [31]. Here $D^{-\frac{1}{2}}$ is a diagonal matrix with each diagonal element being the inverse of the square root of the corresponding element in $D$. At Line 6, CSC derives the eigenvector matrix $E_{n\times n}$ of the normalized Laplacian matrix $L$ through eigendecomposition, so each column of $E$ is an eigenvector.

From Line 7 to Line 12, for each $k$ value (i.e., the number of clusters) of interest, CSC selects the first $k$ eigenvectors corresponding to the $k$ smallest eigenvalues as the spectral representations, which are used for clustering (typically via K-means [31]) and scoring the clustering results. We choose to use the popular Silhouette score [23] in our Score(·) method. The Silhouette score measures the mean intra-cluster distance and the mean nearest-cluster distance of a clustering result. Its value is in the interval of [-1, 1]. A higher Silhouette score indicates a better clustering performance, which means that examples in the same cluster are closer while the distances between different clusters are larger. At Line 12, CSC selects the number with the largest Silhouette score as the optimal number of clusters $k_{opt}$ as suggested in [13]. At Line 13, CSC returns $k_{opt}$ clusters.

**3.3.2. Contrastive Learning (CL) Algorithm.** Contrastive learning is a state-of-the-art self-supervised representation learning technique. It has been demonstrated to be effective in learning informative representations from unlabeled examples for performing multiple types of downstream tasks such as image, text, and graph related classification and clustering [4], [10], [38]. Following the SimCLR framework proposed by Chen et al. [4] (which is for contrastive learning of visual representations), we design our CL algorithm to maximize the agreement between differently augmented views of the same CSP via a contrastive loss in the latent space.

As shown in Algorithm 2, given $X_{n\times d}$ and $m$ (like in Algorithm 1), the batch size in training, a set of data augmentation operators, an encoder network, and a projection head as the inputs, CL will automatically return the learned representations for all examples.

In more details, at Line 1, CL selects two data augmentation operators $t^a$ and $t^b$ to be used for generating pairs of positive examples. The design of data augmentation operators is crucial in contrastive learning and is often unique (ours will be presented below) for different types of tasks. From Line 3 to Line 6, CL generates a pair of positive examples $x_i^a$ and $x_i^b$ (i.e., differently augmented views of $x_i$ via $t^a$ and $t^b$, respectively) from each example $x_i$ in the current batch, extracts their $m$-dimensional representations using an encoder network $f$, and maps the learned representations to the latent space using the projection head $g$. CL does not need to sample negative examples as the other $B-1$ pairs of generated examples in a batch are treated as negative examples. CL simply uses a multilayer perceptron (MLP) with two hidden layers as the encoder network $f$, and uses another MLP with one hidden layer as the projection head $g$. At Line 7, CL computes the contrastive loss $\mathcal{L}_{contrastive}$ averaged over all augmented examples in the current batch, where $l(z_i^a, z_i^b)$ and $l(z_i^b, z_i^b)$ are individual pairwise losses calculated using the normalized temperature-scaled cross entropy loss function as in SimCLR [4]. At Line 8, CL updates the weights in the encoder network and the projection head to minimize the contrastive loss.

After training the contrastive model over all batches in each epoch and often over multiple epochs, CL extracts and returns the final representations for all examples in the dataset using the updated encoder network (Lines 9∼11). Note that the projection head is used only in the training not in this final extraction as the learned $m$-dimensional representations are what CL needs to return to the CSC algorithm for performing the spectral clustering task.

We design a set $T$ of three data augmentation operators {*Add, Delete, Swap*} that can be used for generating pairs of positive examples in our CL algorithm. Recall that the input $X_{n\times d}$ contains $d$-dimensional binary feature vectors. Unlike how data augmentation operators were designed for other types of tasks (e.g., cropping and blurring images [4] in image related tasks, and extracting adjacent and overlapped textual segments [10] in text related tasks), we design these three operators so that the perturbations to each binary feature vector $x_i$ will be relatively small and the augmented positive examples in a pair ($x_i^a$ and $x_i^b$) will be similar. The *Add* operator randomly changes

**Algorithm 2** Contrastive Learning (CL)

**Input:** $X_{n\times d}$ is a dataset of $n$ $d$-dimensional binary feature vectors
**Input:** $m$ is the dimension of the to be learned representations
**Input:** $B$ is the batch size in training
**Input:** $T$ is a set of data augmentation operators
**Input:** $f$ is an encoder network while $g$ is a projection head
**Output:** the learned contrastive learning representations $\bar{X}_{n\times m}$

//Training a contrastive model
1: Select two data augmentation operators $t^a$ and $t^b$ from $T$
2: **for** each batch $\{x_i\}_{i=1}^{B}$ in $X$ **do**
3: **for** i=1 to $B$ **do**
4: $x_i^a$=$t^a(x_i)$; $x_i^b$=$t^b(x_i)$ //generate two positive examples
5: $h_i^a$=$f(x_i^a,\ m)$; $h_i^b$=$f(x_i^b,\ m)$ //extract representations
6: $z_i^a$=$g(h_i^a)$; $z_i^b$=$g(h_i^b)$ //map to the latent space
7: Compute the contrastive loss $\mathcal{L}_{contrastive}$ over this batch using the formula: $\mathcal{L}_{contrastive}$=$\frac{1}{2B}\sum_{i=1}^{B}[l(z_i^a, z_i^b)+ l(z_i^b, z_i^b)]$
8: Update the weights in $f$ and $g$ to minimize $\mathcal{L}_{contrastive}$
//Extracting the final representations for all examples
9: **for** each $x_i$ in $X$ **do**
10: Extract the $m$-dimensional representation: $\bar{x}_i = f(x_i,\ m)$
11: **return** $\bar{X}_{n\times m}$

one bit in $x_i$ from '0' to '1', thus the augmented example will contain a new feature. The *Delete* operator randomly changes one bit in $x_i$ from '1' to '0', thus the augmented example will lose an existing feature. The *Swap* operator randomly swaps a '1' bit and a '0' bit in $x_i$, thus the augmented example will contain a new feature while losing an existing feature. Later we experiment with their combinations and select the best combination for the two operators $t^a$ and $t^b$ at Line 1 in our CL algorithm.

### 3.4. CSP Security Level Analysis Methods

Based on the clustering results, we analyze the security levels of the deployed CSPs from the directive coverage and secure use perspectives. To be effective, a deployed CSP should cover all types of resources needed on a webpage by using different directive names (or some default directive names if available), and should avoid using unsafe directive values which will allow harmful resources to be loaded into a webpage.

Analyzing the secure use of directives is relatively straightforward as it is specific to each individual policy feature. In short, we label each of our designed policy features as a *safe*, *unsafe*, or *uncertain* feature according to the CSP specification [15], [33] and guidelines such as the OWASP XSS Prevention Cheat Sheet [17]. If a policy feature clearly provides some control on resources or behaviors and would not incur potential risks, it is considered safe; if a policy feature clearly incurs potential risks, e.g., by allowing the loading of resources from any place, it is considered unsafe; all other policy features whose safeness would depend on some specific context are conservatively labeled as uncertain.

Analyzing the coverage of a CSP is about inspecting the extent to which the directives of the CSP will protect the actual resources requested or permissible on a webpage. In more details, a Google Chrome extension can directly identify 13 types of resource types in requests: "main_frame", "sub_frame", "stylesheet", "script", "image", "font", "object", "xmlhttprequest", "ping", "csp_report", "media", "websocket", and "other" [5]. We need to map these resource types to the corresponding resource types in CSPs for performing the coverage analysis. For example, "xmlhttprequest", "ping", and "websocket" resource types will be mapped to the "connect" resource type. We also extract other resource types (e.g., "form-action", "manifest", and "prefetch") from the saved HTML document of a webpage, and map them to the resource types in CSPs. On the other hand, we identify from policy features the types of resources controlled by a CSP. Some directive names such as "script-src" and "img-src" directly indicate that one type of resources can be controlled. Some directive names such as "default-src" and "child-src" imply that multiple types of resources can be controlled. Some directive names such as "upgrade-insecure-requests", "frame-ancestors", and "block-all-mixed-content" do not control any types of resources since they are not used for resource control. For other policy features, we identify the type of resources from the directive value token type. For example, the "require-sri-for script" directive indicates that scripts can be controlled.

## 4. Results and Analysis

In this section, we first introduce our CSP dataset and CSC algorithm evaluation. We then present and analyze the detailed CSP clustering results. Finally, we briefly analyze the CSP deployment on subpages.

### 4.1. CSP Dataset

Using Google Chrome and our browser extension, we visited the Alexa top 100K (dated Nov. 9th 2021) websites from Nov. in 2021 to Apr. in 2022. In total, 81,476 homepages were successfully visited, and 14,451 (17.74%) of them contain CSPs. In more details, 13,604 homepages contain CSPs in the enforcement mode and 209 of them contain multiple CSPs, while 847 homepages only contain CSPs in the report-only mode. We further filtered out 287 homepages from those 13,604 homepages because their CSP directives are simply empty, leaving the remaining 13,317 homepages in our dataset. We also successfully visited 110,718 subpages (Section 3.1) of the 13,317 CSP-deployed homepages. To be consistent across different websites, our clustering is based on the CSPs deployed on the 13,317 homepages. Note that Calzavara et al. showed in 2021 that 73% of websites deployed the same CSP between a homepage and its subpages [3]; therefore, in our clustering, excluding the CSPs of subpages also helps mitigate the bias that can be incurred from the identical CSPs of the same website. Instead, we separately analyze the CSPs of subpages at the end of this section.

Our data collection process does not raise obvious ethical issues. It does not involve human subjects or potentially sensitive data (e.g., user behavior or social network information, evaluation of censorship, etc.). Our Google Chrome extension only visited the public homepages and 10 of each homepage's public subpages. On each webpage, our browser extension scrolls down to the bottom and stays 60 seconds to collect HTTP(s) requests and responses, collect CSPs, and save the loaded HTML document; it does not click on links or buttons, thus there is no change to the click-through rate of ads. There is no modification to any accessed webpage. The traffic burden incurred to each accessed website is also minimal.

From each of the 13,317 homepages, we extracted its CSP policy features into a 530-dimensional binary feature vector as described in Section 3.2. Recall that if multiple policies are deployed for a webpage, they will be combined into a single feature vector; thus we simply refer to the CSPs of the 13,317 homepages as 13,317 CSPs. We filtered out 172 features that have the zero value in all 13,317 CSPs because they will not contribute to the clustering, and we reserved the remaining 358 features. In other words, in the inputs to our CSC and CL algorithms, $n$=13,317 and $d$=358. Most removed features are unusual directive name and value (or value token type) combinations in practice, although they are not explicitly prohibited based on the CSP specification [15], [33]. For example, "img-src 'unsafe-allow-redirects' " is unusual as the value 'unsafe-allow-redirects' was designed for the "navigate-to" directive name. Similar examples are "base-uri 'strict-dynamic' " and "prefetch-src 'nonce-<base64-value>' ".

Overall, the reserved 358 features cover all 28 directive names listed in Table 1. Among these 358 features, 219, 70, and 69 are labeled as safe, unsafe, and uncertain, respectively (Section 3.4). Figure 1 shows the ranked popularity of the 358 features among the 13,317 homepages. The details of all these 358 ranked features (with the index from 1 to 358) are shown in Table 5 in Appendix B.

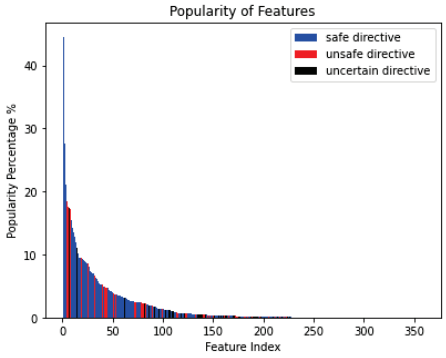



Figure 1: Ranked Popularity of the 358 Features among the 13,317 Homepages

The most popular (on 44.57% of homepages) feature is "upgrade-insecure-requests", which is a safe feature (with an empty directive value token type) used to upgrade insecure *http* URLs to the corresponding secure *https* URLs. The next five most popular features are: "frame-ancestors 'self' " (a safe feature) which specifies that only the webpages with the current URL's origin can be the valid parents of (i.e., can embed) this current webpage, "block-all-mixed-content" (a safe feature with an empty directive value token type) which prevents loading any content over *http* when the webpage uses *https*, "frame-ancestors 'none' " (a safe feature) which specifies that no webpages can be the valid parents of this current webpage, "script-src 'unsafe-inline' " (an unsafe feature) which allows inline scripts to be executed, and "style-src 'unsafe-inline' " (an unsafe feature) which allows inline stylesheets to be loaded. We can see that two unsafe features are in the top-6 list, and they all popularly appear on more than 17% of homepages.

## 4.2. CSC Algorithm Evaluation

**4.2.1. Experiments and Evaluation Metrics.** We evaluate the clustering performance of our CSC algorithm, and compare that with the performance of the K-means algorithm. Meanwhile, to measure the usefulness of contrastive learning in clustering, we also evaluate the performance of a variant of CSC (referred to as *CSC without CL*) and a variant of K-means (referred to as *K-means with CL*); the former is CSC without using our CL algorithm, i.e., CSC without its Line 1 and working directly on the original $d$-dimensional binary feature vectors, while the latter is K-means working on the $m$-dimensional representations learned by our CL algorithm.

The number of clusters $k$ that we are interested in is from five (not too small) to 20 (not too large). We run each of the four algorithms five times on our dataset of 13,317 CSPs. We obtain the optimal number of clusters from each algorithm based on the the largest Silhouette score (Section 3.3.1) averaged over the five runs. While the Silhouette score can be used to automatically quantify the clustering performance, we further verify the clustering performance by manually inspecting the clustering results of 1,000 CSPs randomly sampled from our dataset and by calculating the normalized mutual information (NMI) score [14].

The NMI score is in the interval of [0, 1], and a higher score indicates a better agreement between two clustering results. NMI is also popularly used in evaluating the quality of clustering results such as in [11], [39], but its calculation needs the ground-truth. Basically, given an optimal $k$ value obtained from an algorithm, we manually categorize the sampled 1,000 examples into $k$ clusters as the ground-truth. We then calculate the NMI score between the $k$ clusters produced by the algorithm on the 1,000 examples and the corresponding ground-truth.

**4.2.2. Implementation of the Algorithms.** We implemented our CSC and Cl algorithms using the PyTorch Lightning deep learning framework [19] and the Scikit-learn machine learning library [24]. We used the default K-means implementation in the Scikit-learn library. In CSC and CL algorithms, the dimension of the to be learned representations is $m$=128. In the CL algorithm, each hidden layer of the encoder network $f$ has 256 hidden units, while the hidden layer of the projection head $g$ has 128 units. Each hidden layer in either $f$ or $g$ is followed by a ReLU activation function. While we also explored other parameter values such as a large number of hidden units, these implementation settings worked the best in our experiments. In the CL algorithm, the best combination for the two data augmentation operators $t^a$ and $t^b$ is *Add* and *Swap*. The detailed NMI scores of the experiments for all combinations of data augmentation operators are shown in Table 4 in Appendix A. Some other implementation details such as the learning rate in the CL algorithm are also provided in Appendix A.

**4.2.3. Evaluation Results.** Figure 2 shows the Silhouette scores of the four algorithms under different numbers of clusters. It is obvious that the largest Silhouette score (averaged over the five runs) of each algorithm occurs after $k$ >15, indicating that the optimal number of clusters is greater than 15 for all four algorithms. The optimal number of clusters is 16 for our CSC algorithm. Based on the Silhouette scores, CSC is the best performer once after $k$ >7, demonstrating the overall effectiveness of our algorithm design. Meanwhile, CSC always outperforms *CSC without CL* while *K-means with CL* always outperforms K-means, demonstrating the usefulness of contrastive learning in clustering.

We further calculated the NMI scores for each algorithm after $k$ >15 as shown in Figure 3. We can see that CSC always outperforms *CSC without CL*, which further outperforms both K-means and *K-means with CL* (although it is hard to conclude which one of them is better). These NMI results further demonstrate the overall effectiveness of our algorithm design. The CSP clustering results and analysis presented in the rest of the paper are all based on our CSC algorithm and its optimal number of clusters $k$=16.

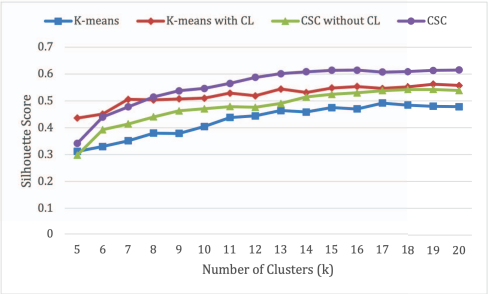



Figure 2: The Silhouette Scores of Algorithms under Different Numbers of Clusters

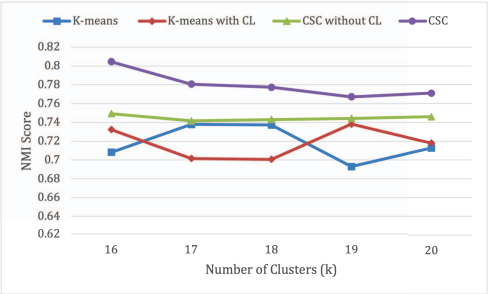



Figure 3: The NMI Scores based on 1,000 Randomly Sampled Examples

### 4.3. CSP Clustering Results and Analysis

Our CSC algorithm categorized the 13,317 CSPs into 16 clusters. Figure 4 shows the TSNE visualization of the 16 clusters. TSNE visualization provides intuitive information in terms of the within and between cluster distributions of datapoints. Using the same feature indexes as in Figure 1 and Table 5 (in Appendix B), Figure 5 shows the feature popularity of each cluster. Note that in each subfigure, the popularity percentage is calculated based on the size (i.e., number of CSPs or homepages) of the corresponding cluster. These subfigures will help us characterize and interpret the clustering results. We now analyze the CSPs in the 16 clusters based on their feature popularity patterns and their security levels from the directive coverage and secure use perspectives. For the ease of presentation, we put the 16 clusters into four groups based on the high-level visual perception of the similar or different feature popularity patterns among the 16 subfigures; meanwhile, within each specific cluster, we refer to a feature as a *popular feature* if above 50% of CSPs in this cluster have this feature. Note that seemingly contradictory directives may co-exist in a CSP for backwards compatibility reasons. For example, 'unsafe-inline', 'nonce-abcdefg', and 'strict-dynamic' may co-exist as values in the "script-src" directive but will act differently in browsers that support different CSP versions [6]. In our analysis, we examine whether such backwards compatible directive values co-occur in a CSP and meanwhile are in popular features. However, backwards compatibility is not a common case in our results, for example, "script-src 'unsafe-inline' " is a popular feature in many clusters but the "script-src 'strict-dynamic' " feature rarely appears in CSPs (Sections 4.3.1-4.3.4).

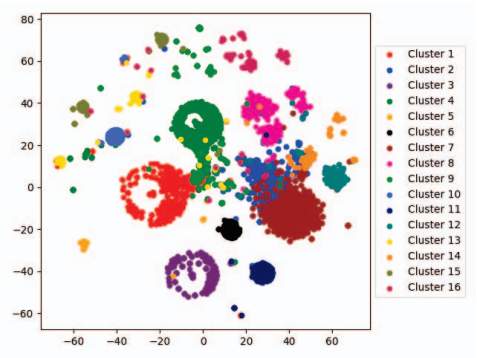
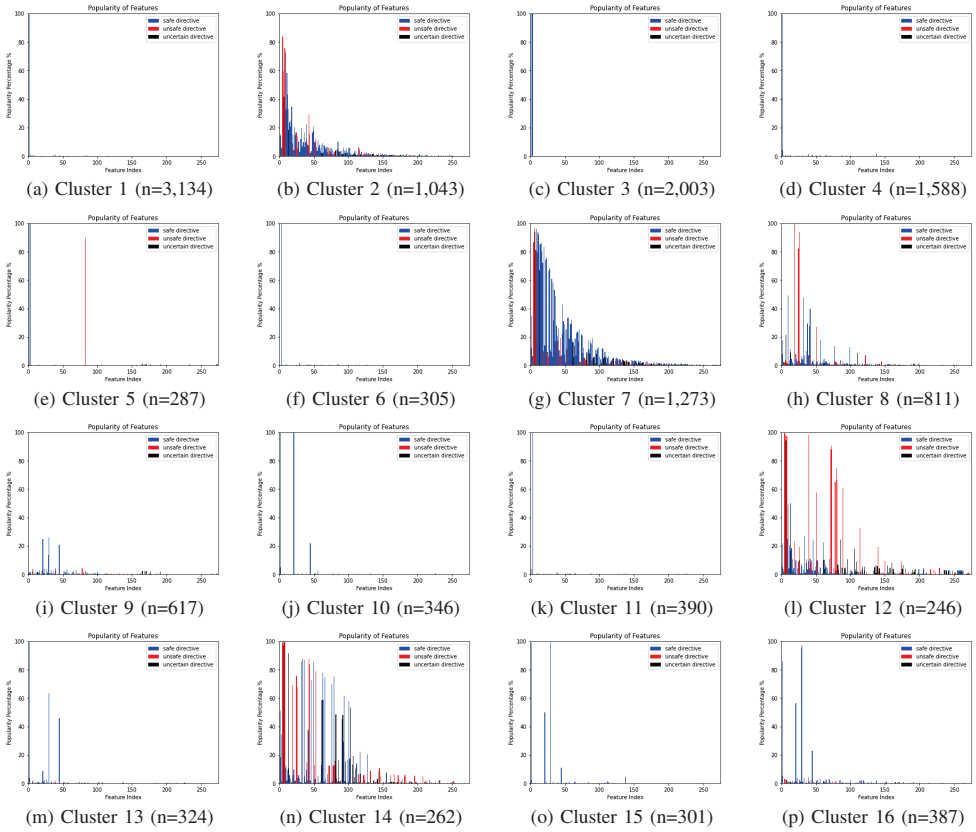



Figure 4: TSNE Visualization of the 16 Clusters

**4.3.1. Group 1.** This group includes Clusters 1, 3, 4, 6, and 11 since the features in these five clusters are sparse (with 54, 9, 63, 19, and 30 features, respectively). In Clusters 1, 4, 6, and 11, there is only one popular feature which is "upgrade-insecure-requests", "frame-ancestors 'self' ", "block-all-mixed-content", and "frame-ancestors 'none' ", respectively; meanwhile, the popularity of all other features in these clusters is less than 10%. In Cluster 3, there are only three features whose popularity is above 10%, and two of them are popular features; meanwhile, all CSPs in this cluster contain "block-all-mixed-content" and "frame-ancestors 'none' ".

Cluster 1 is unique especially because the "upgrade-insecure-requests" directive is the only directive in most CSPs (99.29%), and the popularity of the rest 53 features is less than 1.5%. CSPs in Cluster 1 are mainly used for TLS enforcement. In Cluster 3, all CSPs contain "block-all-mixed-content" and "frame-ancestors 'none' ", but less than 1% of CSPs contain the rest eight policy features. Thus, most CSPs in Cluster 3 are mainly used for TLS enforcement and framing control. In Cluster 4, most CSPs contain the "frame-ancestors 'self' (99.31%) directive, and less than 4.5% of CSPs contain the rest 62 features. Thus, most CSPs in Cluster 4 are mainly used for framing control. All CSPs in Cluster 6 contain "block-all-mixed-content", and less than 2% of CSPs contain the rest 18 features. Thus, CSPs in Cluster 6 are mainly used for TLS enforcement. All CSPs in Cluster 11 contain "frame-ancestors 'none' ", and less than 4% of CSPs contain the rest 29 features. Thus, CSPs in Cluster 11 are mainly used for framing control.

We further analyze the security levels of the CSPs in each cluster from the directive coverage and secure use perspectives. Popular features in Clusters 1, 4, 6, and 11, only cover one category of directives, and popular features in Cluster 3 only cover two categories of directives. Especially, fetching directives are completely missing in the majority of the CSPs in all these five clusters, leaving their webpages vulnerable to XSS attacks. Furthermore, by analyzing the actual resources requested on each webpage (Section 3.4), we verified that only 0.70%, 0.13%, 0.33%, and 0.25% of CSPs in Clusters 1, 4, 6, and 11, respectively, can fully cover or control the actually requested resources on their corresponding webpages. No CSP in Cluster 3 can fully cover or control the actually requested resources on its corresponding webpages. Therefore, we




(a) Cluster 1 (n=3,134) (b) Cluster 2 (n=1,043) (c) Cluster 3 (n=2,003) (d) Cluster 4 (n=1,588)

(e) Cluster 5 (n=287) (f) Cluster 6 (n=305) (g) Cluster 7 (n=1,273) (h) Cluster 8 (n=811)

(i) Cluster 9 (n=617) (j) Cluster 10 (n=346) (k) Cluster 11 (n=390) (l) Cluster 12 (n=246)

(m) Cluster 13 (n=324) (n) Cluster 14 (n=262) (o) Cluster 15 (n=301) (p) Cluster 16 (n=387)

Figure 5: Feature Popularity of Each Cluster

consider that the CSPs in all these five clusters are generally at the low-level on directive coverage. Since all popular features in Clusters 1, 3, 4, 6, and 11 are safe features, we consider that the CSPs in them are generally at the high-level on secure use.

**4.3.2. Group 2.** This group includes Clusters 2 and 7 since the popularity of features gradually decreases in the two subfigures. Clusters 2 and 7 both contain above half of the features (257 and 327, respectively) in total, but their popular features are less than 7.5% (5 and 23, respectively). The CSPs in both clusters often contain a "style-src" or "script-src" directive name with an 'unsafe-inline' or 'unsafe-eval' value to allow any scripts or stylesheets to be loaded, and often whitelist some domain sources. In Cluster 7, above 50% of CSPs also contain at least one of five other fetch directives (i.e., "default-src", "img-src", "font-src", "frame-src", and "connect-src"), while "style-src" or "script-src" are the only two directive names contained by above 50% of CSPs in Cluster 2. Since all popular features in Clusters 2 and 7 are fetch directives, CSPs in both clusters are mainly used for protecting against XSS attacks.

We now analyze the security levels of their CSPs. Popular features in Cluster 2 only include "style-src" and "script-src" directives, and less than 35% of CSPs contain a "default-src" directive which can cover almost all types of resources. We also found that less than 20% of CSPs include navigation directives and document directives in Cluster 2. Popular features in Cluster 7 include seven fetch directives including a "default-src" directive, but less than 35% of CSPs include navigation directives, and less than 20% of CSPs include document directives. Note that missing navigation or document directives in a CSP will make a webpage vulnerable to *UI Redressing* attacks, and malicious resources can be inserted through the document type of resources such as a <base> tag. Meanwhile, only 28.28% of CSPs in Cluster 2 can fully control the requested resources in their corresponding webpages, while that percentage is 54.91% in Cluster 7. Therefore, we consider that the CSPs in Cluster 2 are at the low-level on directive coverage, while the CSPs in Cluster 7 are generally at the medium-level on directive coverage. In Cluster 2, 40% of popular features are safe thus we consider that its CSPs are generally at the low-level on secure use. In Cluster 7, 78.26% of popular features are safe while less than 20% of popular features are unsafe, thus we consider that its CSPs are generally at the medium-level on secure use.

**4.3.3. Group 3.** This group includes Clusters 5, 9, 10, 13, 15, and 16 since their popular features (excluding Cluster 9 which does not have a popular feature) contain some features that are not top-ranked in Table 5 (in Appendix B). Among the total 18 features in Cluster 5, only three are popular features; there is no popular feature in Cluster 9 among 136 features; only two out of the total 18 features in Cluster 10 are popular features; only two out of the total 48 features in Cluster 13 are popular features; only two out of the total 20 features in Cluster 15 are popular features; only four out of the total 94 features in Cluster 16 are popular features.

All CSPs in Cluster 5 contain "upgrade-insecure-requests" and "block-all-mixed-content" directives, and 89.89% of them contain "frame-ancestors *" which allows any webpages to embed a current webpage. The popularity of the rest features is less than 1%. Therefore, CSPs in Cluster 5 are mainly used for TLS enforcement and framing control. The most popular feature in Clusters 9 and 16 is "frame-ancestors *.sado" (26.09% and 97.41%, respectively). About 15% of CSPs in Cluster 9 use 'frame-ancestors *.exdo" to allow webpages from external sources to embed a current webpage by using a domain combination scheme, while that percentage in Cluster 16 is 95.61%. The most popular feature in Cluster 10 is "frame-ancestors exdo" which allows webpages from external sources to embed a current webpage without using a domain combination scheme. Most CSPs in Cluster 10 also allow webpages from the same origin to embed a current webpage. The most popular feature in Cluster 13 is "frame-ancestors 'self' ", Most CSPs in Cluster 13 also allow webpages from the same domain to embed a current webpage. The most popular feature in Cluster 15 is "frame-ancestors *.exdo". The directive name of features whose popularity is above 10% in Clusters 9, 10, 13, 15, and 16 is "frame-ancestors". Therefore, CSPs in Clusters 9, 10, 13, 15, and 16 are mainly used for framing control.

We now analyze the security levels of their CSPs. Popular features in Cluster 5 only include one navigation directive "frame-ancestors" and two other directives (i.e., "upgrade-insecure-requests" and "block-all-mixed-content"); popular features in Clusters 10, 13, 15, and 16 only include one navigation directive "frame-ancestors"; there is no popular feature in Cluster 9. We can see that popular features in all these six clusters do not include any fetch or document directives. We consider that the CSPs in Clusters 5, 9, 10, 13, 15, and 16 are generally at the low-level on directive coverage because in each cluster, less than 2% of CSPs (0%, 0.97%, 0%, 0%, 0% and 1.81%, respectively) can cover requested resources. Since all popular features in Clusters 10, 13, 15, and 16 are safe, we consider that their CSPs are generally at the high-level on secure use. Since 66.67% of popular features are unsafe in Cluster 5 and there is no safe popular feature in Cluster 9, we consider that the CSPs in these two clusters are generally at the low-level on secure use.

**4.3.4. Group 4.** This group includes Clusters 8, 12, and 14 since all these clusters contain many features whose popularity is above 20%. In Cluster 8, three out of the total 191 features are popular features, and the popularity of two popular features is above 90%; in Cluster 12, 12 out of the total 211 features are popular features, and the popularity of six popular features is above 90%; in Cluster 14, 26 out of the total 191 features are popular features, and the popularity of five popular features is above 90%.

In Cluster 8, containing a "default-src" directive in a policy is common; meanwhile CSPs in Cluster 8 often use a "data:" scheme and an 'unsafe-inline' or 'unsafe-eval' value to allow any scripts or stylesheets to be loaded. CSPs in Cluster 12 commonly deploy eight fetch directives: "style-src", "img-src", "script-src", "default-src", "connect-src", "font-src", "frame-src", and "media-src"; meanwhile, they control the resources through '*' value or through a "data:" scheme. The wildcard '*' allows resources from any sources to be loaded in a webpage. CSPs in Cluster 14 often deploy 10 directives: "font-src", "upgrade-insecure-requests", "media-src", "style-src", "connect-src", "script-src", "object-src", "default-src", "img-src", and "child-src". Most CSPs in Cluster 14 control resources through an "https:", "data:", or "blob:" scheme, and all CSPs contain "img-src data: ". They also often control scripts and stylesheets by using "script-src" and "style-src" directive names with an 'unsafe-eval' or 'unsafe-inline' value. Since popular features in Clusters 8, 12, and 14 are fetch directives, CSPs in these three clusters are mainly used for XSS mitigation.

We now analyze the security levels of CSPs in these clusters. Popular features in all these three clusters do not cover any navigation or document directives. In all these three clusters, less than 50% (32.55%, 35.77%, and 35.69%, respectively) of CSPs can cover all requested resources in the corresponding webpages, so we consider that the CSPs in all three clusters are generally at the low-level on directive coverage. Since above 90% of popular features are unsafe in Clusters 8 and 12 (100% and 91.67%, respectively), we consider that the CSPs in Clusters 8 and 12 are generally at the low-level on secure use. Since 57.67% of popular features in Cluster 14 are safe, we consider that the CSPs in Cluster 14 are generally at the medium-level on secure use.

**4.3.5. Potential Reasons for Some CSP Patterns.** We consider a group of popular directives in a cluster as a CSP pattern for the cluster. Analysis of popular directives in each cluster shows that some popular directives are shared in several CSP patterns while some popular directives are uniquely contained by a certain cluster. To figure out how webpage properties contribute to CSP patterns of clusters, we further analyzed development settings and CSP contents of webpages.

Most webpages deployed an identical CSP in Clusters 3 and 5. We observed that 1,995 out of 2,003 webpages in Cluster 3 deployed the identical CSP which is *"block-all-mixed-content; frame-ancestors 'none'; upgrade-insecure-requests;"*, and 256 out of 287 webpages in Cluster 5 deployed the identical CSP which is *"block-all-mixed-content; frame-ancestors *; upgrade-insecure-requests;"*. What are the reasons for more than 90% webpages of different websites to deploy an identical CSP in Clusters 3 and 5? Why popular directives are so similar in Clusters 3 and 5? By analyzing development settings of webpages, we found that 1,985 (examples are in Appendix C.1) out of those 1,995 webpages in Cluster 3 were built by using Shopify (which is a platform for building shopping websites [27]), and all those 256 webpages in Cluster 5 were built by using Shopify. We also checked development settings of webpages in other clusters, and found that only ten webpages were built by using Shopify.

One possible reason is that there is a default CSP provided by Shopify, thus identical CSPs are commonly used in Clusters 3 and 5, respectively. However, it is difficult to check all website templates provided by Shopify because most of them are not free. Based on the replies in the Shopify community (community.shopify.com), we figured out that for protecting websites against clickjacking attacks, Shopify deploys a default CSP which is the one used in those 1,985 webpages in Cluster 3. However, those

256 webpages in Cluster 5 modified the default CSP to allow any frame ancestors, thus they may not be able to sufficiently prevent clickjacking attacks. It seems that the developers of those 256 webpages in Cluster 5 realized that there is a default CSP deployed in their webpages. On the other hand, it is unclear if the developers of those 1,985 webpages in Cluster 3 were aware of that default CSP deployed in their webpages and its purposes. The default CSP provided by Shopify also explains why most CSPs in Clusters 3 and 5 do not contain fetch directives.

Similarly, most webpages deployed an identical CSP in Clusters 1, 4, and 6, respectively. We observed that 3,030 out of 3,134 webpages in Cluster 1 only deployed an "upgrade-insecure-requests" directive, 1,440 out of 1,588 webpages in Cluster 4 only deployed a "frame-ancestors 'self' " directive, and 293 out of 305 webpages in Cluster 6 only deploy a "block-all-mixed-content" directive. We found that 1,242 out of those 3,030 webpages in Cluster 1 and 218 out of those 293 webpages in Cluster 6 were built by using WordPress (which is a platform for building any websites [37]). In other clusters, 751 webpages were built by using WordPress, but less than 16% webpages in each cluster were built by using this platform. Most webpages built by using WordPress focus on controlling *http* URLs by deploying an "upgrade-insecure-requests" directive or "block-all-mixed-content" directive. In Cluster 4, 333 out of those 1,440 webpages were built by using Webflow (which is a platform like WordPress for building any websites [34]), and only 46 webpages were built by using this platform in other clusters. Meanwhile, 337 out of those 3,030 webpages in Cluster 1 were built by using HubSpot (which is also a platform for building any websites [12]), and only five webpages were built by using this platform in other clusters. Unlike Shopify, WordPress, HubSpot, and Webflow did not provide a default CSP for webpages built by using them.

CSPs in Clusters 2, 7, 8, 12, and 14 commonly contain fetch directives. Meanwhile, 'unsafe-inline' and 'unsafe-eval' are popular in these five clusters likely because plugins and event handlers cannot be easily whitelisted by using hashes or nonces [21], [22]. We wondered whether default CSPs provided by website builders also contribute to the common use of 'unsafe-inline' and 'unsafe-eval'. However, we did not find clear evidence to show website builders' contribution of these two directives.

**4.3.6. Summary of the Analysis Results.** Overall, based on the 16 subfigures in Figure 5 and our detailed descriptions, we summarized the main aims of the CSPs and the CSP security level analysis results for the 16 clusters in Table 3. We can see that all these 16 clusters have their own unique characteristics, and CSPs in each cluster often have limited aims or even a single aim. Furthermore, based on those unique characteristics and by analyzing the 16 clusters in four groups, we identified many common or different CSP directive use practices within and between clusters. All these further helped us analyze the CSP security levels in each cluster from the directive coverage and secure use perspectives.

**Main Aims of the CSPs**. From Table 3, we can see that CSPs in Clusters 1 and 6 are mainly used for TLS enforcement; CSPs in Clusters 2, 7, 8, 12, and 14 are mainly used for protecting against XSS attacks; CSPs in Clusters 3 and 5 are mainly used for TLS enforcement and framing control; CSPs in Clusters 4, 9, 10, 11, 13, 15, and 16 are mainly used for framing control. Similar to what Calzavara et al. found in 2018 [2], CSP is no longer used mainly for XSS mitigation. In those clusters whose CSPs are used mainly for protecting against XSS attacks, the "object-src" directive name is unfortunately not in any popular feature. However, to better prevent script injection attacks, developers should use both "script-src" and "object-src" directives [35]. Nonces and hashes are not commonly used either, and they are not even in the top-50 features in Table 5; however, 'unsafe-inline' and 'unsafe-eval' directive values are still commonly used and in the top-10 features. The low adoption of nonces and hashes may be attributed to the insufficient support from event handlers, web development frameworks, and plugins [21], [22].

**Security Levels of the CSPs**. Unfortunately, no cluster has its CSPs at the high-level on directive coverage. CSPs in one cluster are generally at the medium-level on directive coverage, while CSPs in 15 clusters are generally at the low-level on directive coverage. It is worth noting that the popular features in most low-level directive coverage clusters only cover at most two types of directives with limited aims or limited controls on resource types (e.g., only on scripts). From the directive secure use perspective, CSPs in nine, two, and five clusters are generally at the high-level, medium-level, and low-level, respectively. It is worth noting that all the nine high-level secure use clusters contain less than 100 features, which are all unfortunately at the low-level on directive coverage. It is also worth noting that CSPs in the medium-level secure use clusters popularly contain unsafe directives "script-src 'unsafe-inline' ", "script-src 'unsafe-eval' ", and "style-src 'unsafe-inline' ". We conclude that most deployed CSPs do not sufficiently protect webpages from either the directive coverage or the directive secure use perspective.

Previous studies such as [1], [2], [21], [35] mainly analyzed the vulnerabilities of specific directives, but did not estimate the overall protection capabilities of CSPs, especially from the directive coverage perspective. Meanwhile, we found that severe problems (unreported in previous studies such as [2], [21], [35], [36]) exist in some specific clusters: (1) fetch directives are completely missing in the majority of CSPs in Clusters 1, 3, 4, 6, and 11; (2) 89.90% of CSPs in Cluster 5 contain "frame-ancestors *" which allows any webpages to embed a current webpage; (3) CSPs with a "default-src" fallback directive in Clusters 8, 12, and 14 often contain unsafe directive values such as 'unsafe-inline', 'unsafe-eval', or "*". Some example websites with such problems are in Appendix C.2.

## 4.4. CSP Deployment on Subpages

We further analyzed the CSP deployment on the 110,718 subpages of the 13,317 CSP-deployed homepages. We found that 12,373 homepages contain same domain or subdomain subpages, while the rest 944 homepages do not. Among those 12,373 homepages, 9,862 of them have all their sampled subpages containing CSPs, 2,365 of them have portions of their sampled subpages containing CSPs, and 146 of them do not contain any CSP on their sampled subpages. In total, we found that

Table 3: Summary of the Main Aims of the CSPs and the CSP Security Level Analysis Results for the 16 Clusters

| Cluster No. | Main Aims of the CSPs in the Cluster | Number of Websites in the Cluster | Overall Level on Directive Coverage | Overall Level on Directive Secure Use |
|---|---|---|---|---|
| 1 | TLS enforcement via "upgrade-insecure-requests" | 3,134 | low-level | high-level |
| 2 | XSS mitigation via "script-src" and "style-src" directives | 1,043 | low-level | low-level |
| 3 | TLS enforcement via "block-all-mixed-content"; Framing control via "frame-ancestors 'none' " | 2,003 | low-level | high-level |
| 4 | Framing control via "frame-ancestors 'self' " | 1,588 | low-level | high-level |
| 5 | TLS enforcement via "upgrade-insecure-requests" and "block-all-mixed-content"; Framing control via "frame-ancestors *" | 287 | low-level | low-level |
| 6 | TLS enforcement via "block-all-mixed- content" | 305 | low-level | high-level |
| 7 | XSS mitigation via fetch directives with external domain combinations and a "self" value | 1,273 | medium-level | medium-level |
| 8 | XSS mitigation via a "default-src" directive | 811 | low-level | low-level |
| 9 | Framing control via whitelisting sources | 617 | low-level | low-level |
| 10 | Framing control via "frame-ancestors exdo" and "frame-ancestors 'self' " | 346 | low-level | high-level |
| 11 | Framing control via "frame-ancestors 'none' " | 390 | low-level | high-level |
| 12 | XSS mitigation via fetch directives with a "*" value | 246 | low-level | low-level |
| 13 | Framing control via "frame-ancestors *.sado" and "frame-ancestors 'self' " | 324 | low-level | high-level |
| 14 | XSS mitigation via fetch directives with blob:, data:, and https: schemes | 262 | low-level | medium-level |
| 15 | Framing control via "frame-ancestors *.exdo" | 301 | low-level | high-level |
| 16 | Framing control via "frame-ancestors *.exdo" and "frame-ancestors 'self' " | 387 | low-level | high-level |

the subpages of 9,726 homepages all contain CSPs in the enforcement mode, while the subpages of 136 homepages contain CSPs in both the enforcement and report-only modes. We focused on analyzing the CSPs deployed on subpages in the enforcement mode, and comparing if they are identical to the CSPs deployed on their corresponding homepages. Overall, subpages of 9,441 homepages have identical policy feature vectors as their corresponding homepages, and subpages of 9,269 of these homepages further have the identical CSP content as their corresponding homepages. In summary, most of subpages deployed the same CSP as in their homepages. This result is largely consistent with what Calzavara et al. found in [3]. It is worth noting that although deploying the same CSP on multiple webpages of a website is convenient for web developers, it should be clear to them that the directive coverage and secure use problems are also common across all their webpages that share a common CSP.

# 5. Takeaways and Recommendations

## 5.1. High-level Takeaways from Our Study

We have three high-level takeaways from our study. First, taking the clustering approach allows us to automatically categorize CSPs and effectively analyze common or different security problems (Section 4); if we statistically analyze the data based on specific rules or heuristics, much more effort would be needed, and the analysis would not be comprehensive as we discussed in Sections 2.2 and 3.2. Second, it is important to analyze CSPs from both directive coverage and secure use perspectives (Sections 2.2 and 3.4); as we summarized in Section 4.3.6, nine CSP clusters often contain secure directives but have a poor directive coverage, and CSPs in most clusters do not sufficiently protect webpages from one of the two perspectives. Third, we have the following new findings beyond existing studies: (1) each of those 16 clusters has its unique CSP patterns, and most CSPs have limited aims or even a single aim (Sections 4.3.1-4.3.4); (2) 15 clusters are at the low-level on the directive coverage (a major problem) and five clusters are at the low-level on the secure use of directives, while previous studies did not analyze the overall protection capabilities of CSPs (Section 4.3.6); (3) web development platforms contributed to the specific CSP patterns of many websites (Section 4.3.5 and Appendix C.1); (4) several severe problems exist in specific clusters (Section 4.3.6 and Appendix C.2).

## 5.2. Recommendations for Web Developers

Based on our findings, we provide three recommendations that may help developers deploy CSPs or improve their currently deployed CSPs. First, for developers of CSP-deployed websites, we recommend them to improve CSPs from both the directive coverage and the secure use perspectives, and especially, to put more effort on improving the directive coverage of CSPs to comprehensively protect a website. Second, for web developers who would like to build their websites through a web development platform, we recommend them to ascertain and leverage the CSP support (if exists) of the web development platform; meanwhile, they should upgrade (instead of downgrading) the protection capability of their CSPs when they further customize the policies. Third, we recommend web developers to avoid those severe problems in specific clusters as summarized in Section 4.3.6. We further provide a secure CSP example in Appendix C.3 for developers to use as a reference in their CSP deployment.

# 6. Conclusion

In this paper, we took the unsupervised clustering approach to analyze the security levels of the deployed CSPs from the directive coverage and secure use perspectives. We designed policy features and a Contrastive Spectral

Clustering (CSC) algorithm to automatically categorize very diverse and complex CSPs. From the 13,317 homepages that deployed CSPs under the enforcement mode, we categorized their policies into 16 clusters with different characteristics. We found that 15 clusters are at the low level on the coverage and five clusters are at the low level on the secure use of directives; meanwhile, no cluster is at the high level on the coverage of directives, and nine clusters are at the high level on the secure use of directives. These results indicate that most deployed CSPs do not sufficiently protect webpages; more importantly, clustering helps identify the corresponding common or different reasons from the directive coverage and secure use perspectives.

While our detailed findings depend on the specific CSP dataset that we constructed, our design and methodology on policy feature, clustering algorithm, and security level analysis can be generally applied to any dataset. Meanwhile, while the interpretations of the clustering results still need human efforts, everything else is largely automatic. Overall, our approach and results can be helpful for promoting the proper deployment of CSPs. We suggest that developers should improve CSPs from both the directive coverage and secure use perspectives. Our dataset and source code are available at [7].

## Acknowledgments

We sincerely thank our shepherd Dr.-Ing. Aurore Fass and anonymous reviewers for their valuable suggestions. This research was partially supported by the NSF grant OIA-1936968.

## References

[1] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. Content Security Problems?: Evaluating the Effectiveness of Content Security Policy in the Wild. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS), 2016.

[2] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. Semantics-Based Analysis of Content Security Policy Deployment. ACM Transactions on the Web (TWEB), 12(2), 2018.

[3] Stefano Calzavara, Tobias Urban, Dennis Tatang, Marius Steffens, and Ben Stock. Reining in the Web's Inconsistencies with Site Policy. In Proceedings of the Network and Distributed System Security Symposium (NDSS), 2021.

[4] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A Simple Framework for Contrastive Learning of Visual Representations. In Proceedings of the International Conference on Machine Learning (ICML), 2020.

[5] Google Chrome webRequest ResourceType, 2022. https://developer.chrome.com/docs/extensions/reference/webRequest.

[6] CSP script-src Backwards Compatibility, 2023. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src.

[7] Datasets and Code of This Paper, 2023. https://github.com/mengxiaren2024/CSP_via_Clustering.git.

[8] Benjamin Eriksson and Andrei Sabelfeld. AutoNav: Evaluation and Automatization of Web Navigation Policies. In Proceedings of The Web Conference, 2020.

[9] Mattia Fazzini, Prateek Saxena, and Alessandro Orso. AutoCSP: Automatically Retrofitting CSP to Web Applications. In Proceedings of the IEEE/ACM IEEE International Conference on Software Engineering (ICSE), 2015.

[10] John Giorgi, Osvald Nitski, Bo Wang, and Gary Bader. DeCLUTR: Deep Contrastive Learning for Unsupervised Textual Representations. In Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL) and the International Joint Conference on Natural Language Processing (IJCNLP), 2021.

[11] Xiaolong Gong, Linpeng Huang, and Fuwei Wang. Feature Sampling Based Unsupervised Semantic Clustering for Real Web Multi-View Content. In Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), 2019.

[12] HubSpot Free Drag-and-Drop Website Builder, 2022. https://www.hubspot.com/products/cms/drag-and-drop-website-builder.

[13] Leonard Kaufman and Peter J Rousseeuw. Finding Groups in Data: An Introduction to Cluster Analysis. John Wiley & Sons, 2009.

[14] Andrea Lancichinetti, Santo Fortunato, and János Kertész. Detecting the Overlapping and Hierarchical Community Structure in Complex Networks. New journal of physics, 11(3), 2009.

[15] MDN Web Docs: Content-Security-Policy, 2022. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy.

[16] Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. On Spectral Clustering: Analysis and an Algorithm. In Proceedings of the International Conference on Neural Information Processing Systems (NIPS), 2001.

[17] OWASP Cross Site Scripting Prevention Cheat Sheet, 2022. https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html.

[18] Xiang Pan, Yinzhi Cao, Shuangping Liu, Yu Zhou, Yan Chen, and Tingzhe Zhou. CSPAutoGen: Black-box Enforcement of Content Security Policy upon Real-world Websites. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS), 2016.

[19] PyTorch Lightning deep learning framework, 2022. https://www.pytorchlightning.ai/.

[20] Sebastian Roth, Michael Backes, and Ben Stock. Assessing the Impact of Script Gadgets on CSP at Scale. In Proceedings of the ACM Asia Conference on Computer and Communications Security (AsiaCCS), 2020.

[21] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies. In Proceedings of the Network and Distributed System Security Symposium (NDSS), 2020.

[22] Sebastian Roth, Lea Gröber, Michael Backes, Katharina Krombholz, and Ben Stock. 12 Angry Developers - A Qualitative Study on Developers' Struggles with CSP. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS), 2021.

[23] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. Journal of Computational and Applied Mathematics, 20:53–65, 1987.

[24] Scikit-learn machine learning library, 2022. https://scikit-learn.org/stable/.

[25] Uri Shaham, Kelly Stanton, Henry Li, Boaz Nadler, Ronen Basri, and Yuval Kluger. SpectralNet: Spectral Clustering using Deep Neural Networks. In Proceedings of the International Conference on Learning Representations (ICLR), 2018.

[26] Jianbo Shi and Jitendra Malik. Normalized Cuts and Image Segmentation. IEEE Transactions on Pattern Analysis and Machine Intelligence, 22(8):888–905, 2000.

[27] Shopify: Easily Build and Run Your Ecommerce Website, 2022. https://www.shopify.com/tour/ecommerce-website.

[28] Dolière Francis Somé, Nataliia Bielova, and Tamara Rezk. On the Content Security Policy Violations Due to the Same-Origin Policy. In Proceedings of the International Conference on World Wide Web (WWW), 2017.

[29] Dolière Francis Somé and Tamara Rezk. Strenghtening Content Security Policy via Monitoring and URL Parameters Filtering. In Proceedings of the Workshop on Privacy in the Electronic Society (WPES), 2020.

[30] Yaling Tao, Kentaro Takagi, and Kouta Nakata. Clustering-friendly Representation Learning via Instance Discrimination and Feature Decorrelation. In Proceedings of the International Conference on Learning Representations (ICLR), 2021.

[31] Ulrike von Luxburg. A Tutorial on Spectral Clustering. Statistics and Computing, 17(4):395–416, 2007.

[32] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. CCSP: Controlled Relaxation of Content Security Policies by Runtime Policy Composition. In Proceedings of the USENIX Security Symposium, 2017.

[33] W3C: Content Security Policy Level 3, 2022. https://www.w3.org/TR/CSP3/.

[34] Webflow Website Builder, 2022. https://webflow.com.

[35] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS), 2016.

[36] Michael Weissbacher, Tobias Lauinger, and William K. Robertson. Why Is CSP Failing? Trends and Challenges in CSP Adoption. In Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID), 2014.

[37] WordPress Website Builder, 2022. https://wordpress.com/website-builder/.

[38] Yuning You, Tianlong Chen, Yongduo Sui, Ting Chen, Zhangyang Wang, and Yang Shen. Graph Contrastive Learning with Augmentations. In Proceedings of the Conference on Neural Information Processing Systems (NeurIPS), 2020.

[39] Sihang Zhou, Xinwang Liu, Jiyuan Liu, Xifeng Guo, Yawei Zhao, En Zhu, Yongping Zhai, Jianping Yin, and Wen Gao. Multi-View Spectral Clustering with Optimal Neighborhood Laplacian Matrix. In Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), 2020.

[40] Fangyu Zou, Li Shen, Zequn Jie, Weizhong Zhang, and Wei Liu. A Sufficient Condition for Convergences of Adam and RMSProp. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2019.

## A. Other Parameters in Our Algorithms and Combinations of Data Augmentation Operators

We provide more implementation details of our CSC algorithm (Algorithm 1) and CL algorithm (Algorithm 2) in this appendix. In the CSC algorithm, 200 neighbors are used in the nearest neighbors method (Line 2 in Algorithm 1). In the CL algorithm, the batch size (Line 2 in Algorithm 2) is $B$=193; the fine-tuned temperature in the contrastive loss function $\mathcal{L}_{contrastive}$ (Line 7 in Algorithm 2) is 0.5; the training epoch is 100; the optimizer is RMSprop [40] while the learning rate is $1 \times 10^{-4}$, which are used to optimize the encoder network $f$ and the projection head $g$. Table 4 lists the NMI scores for all six combinations of the three data augmentation operators we considered in the CL algorithm. The best combination is *Add & Swap*.

Table 4: NMI Scores of the Experiments for Six Combinations of Three Data Augmentation Operators.

| Combination | NMI Score |
|---|---|
| Add & Add | 0.7882884 |
| Delete & Delete | 0.775500 |
| Swap & Swap | 0.7968726 |
| Add & Delete | 0.7721543 |
| Add & Swap | **0.8044657** |
| Delete & Swap | 0.7884328 |

## B. The 358 Ranked Policy Features

Table 5 lists all the 358 ranked policy features which are used for clustering the CSPs from the 13,317 homepages.

Table 5: Details of the 358 Ranked Policy Features. The blue color indicates safe features, the red color indicates unsafe features, and the black color indicates uncertain features.

| Rank | Feature | Rank | Feature | Rank | Feature |
|---|---|---|---|---|---|
| 1 | (upgrade-insecure-requests, “ ”) | 2 | (frame-ancestors, self) | 3 | (block-all-mixed-content, “ ”) |
| 4 | (frame-ancestors, none) | 5 | (script-src, unsafe-inline) | 6 | (style-src, unsafe-inline) |
| 7 | (img-src, data) | 8 | (script-src, unsafe-eval) | 9 | (script-src, self) |
| 10 | (default-src, self) | 11 | (script-src, exdo) | 12 | (style-src, self) |
| 13 | (img-src, self) | 14 | (font-src, data) | 15 | (font-src, self) |
| 16 | (connect-src, self) | 17 | (style-src, exdo) | 18 | (script-src, *.exdo) |
| 19 | (default-src, unsafe-inline) | 20 | (connect-src, exdo) | 21 | (frame-ancestors, exdo) |
| 22 | (img-src, exdo) | 23 | (frame-src, exdo) | 24 | (font-src, exdo) |
| 25 | (default-src, data) | 26 | (default-src, unsafe-eval) | 27 | (frame-src, self) |
| 28 | (connect-src, *.exdo) | 29 | (frame-ancestors, *.exdo) | 30 | (frame-ancestors, *.sado) |
| 31 | (img-src, *.exdo) | 32 | (default-src, https) | 33 | (frame-src, *.exdo) |
| 34 | (img-src, blob) | 35 | (style-src, *.exdo) | 36 | (img-src, https) |
| 37 | (media-src, self) | 38 | (default-src, exdo) | 39 | (object-src, none) |
| 40 | (img-src, *) | 41 | (default-src, *.exdo) | 42 | (default-src, blob) |
| 43 | (script-src, https) | 44 | (script-src, blob) | 45 | (frame-ancestors, sado) |
| 46 | (media-src, blob) | 47 | (font-src, *.exdo) | 48 | (script-src, *.sado) |
| 49 | (script-src, sado) | 50 | (style-src, https) | 51 | (default-src, *) |
| 52 | (object-src, self) | 53 | (script-src, data) | 54 | (connect-src, sado) |
| 55 | (connect-src, *.sado) | 56 | (base-uri, self) | 57 | (default-src, *.sado) |
| 58 | (img-src, sado) | 59 | (img-src, *.sado) | 60 | (media-src, exdo) |
| 61 | (worker-src, blob) | 62 | (media-src, data) | 63 | (font-src, https) |
| 64 | (child-src, blob) | 65 | (form-action, self) | 66 | (connect-src, https) |
| 67 | (child-src, self) | 68 | (style-src, *.sado) | 69 | (frame-src, *.sado) |
| 70 | (worker-src, self) | 71 | (media-src, *.exdo) | 72 | (script-src, *) |
| 73 | (style-src, *) | 74 | (frame-src, sado) | 75 | (style-src, sado) |
| 76 | (connect-src, blob) | 77 | (default-src, sado) | 78 | (frame-src, *) |
| 79 | (media-src, https) | 80 | (font-src, *) | 81 | (connect-src, *) |
| 82 | (style-src, data) | 83 | (frame-ancestors, *) | 84 | (child-src, exdo) |
| 85 | (script-src, nonce-) | 86 | (img-src, unsafe-inline) | 87 | (font-src, sado) |
| 88 | (manifest-src, self) | 89 | (media-src, *) | 90 | (font-src, *.sado) |
| 91 | (connect-src, wss) | 92 | (connect-src, data) | 93 | (style-src, blob) |
| 94 | (child-src, https) | 95 | (form-action, exdo) | 96 | (font-src, unsafe-inline) |
| 97 | (default-src, none) | 98 | (child-src, *.exdo) | 99 | (default-src, wss) |
| 100 | (base-uri, none) | 101 | (object-src, https) | 102 | (frame-src, https) |
| 103 | (child-src, data) | 104 | (media-src, sado) | 105 | (media-src, *.sado) |
| 106 | (connect-src, unsafe-inline) | 107 | (frame-src, data) | 108 | (object-src, exdo) |
| 109 | (style-src, unsafe-eval) | 110 | (default-src, other scheme) | 111 | (script-src, sha256-) |
| 112 | (frame-src, blob) | 113 | (form-action, *.exdo) | 114 | (object-src, *) |
| 115 | (script-src, http) | 116 | (script-src, strict-dynamic) | 117 | (font-src, blob) |
| 118 | (img-src, other scheme) | 119 | (script-src, report-sample) | 120 | (object-src, *.exdo) |
| 121 | (frame-src, unsafe-inline) | 122 | (default-src, http) | 123 | (script-src-elem, unsafe-inline) |
| 124 | (img-src, http) | 125 | (child-src, *.sado) | 126 | (child-src, sado) |
| 127 | (form-action, sado) | 128 | (form-action, https) | 129 | (script-src-elem, self) |
| 130 | (script-src-elem, exdo) | 131 | (form-action, *.sado) | 132 | (object-src, data) |
| 133 | (object-src, *.sado) | 134 | (prefetch-src, self) | 135 | (script-src, other scheme) |
| 136 | (frame-src, other scheme) | 137 | (img-src, unsafe-eval) | 138 | (frame-ancestors, other scheme) |
| 139 | (media-src, unsafe-inline) | 140 | (worker-src, *) | 141 | (style-src-elem, unsafe-inline) |
| 142 | (worker-src, data) | 143 | (object-src, blob) | 144 | (worker-src, sado) |
| 145 | (default-src, ws) | 146 | (object-src, sado) | 147 | (style-src, http) |
| 148 | (worker-src, exdo) | 149 | (style-src-elem, self) | 150 | (child-src, *) |
| 151 | (connect-src, other scheme) | 152 | (worker-src, *.exdo) | 153 | (worker-src, *.sado) |
| 154 | (font-src, unsafe-eval) | 155 | (worker-src, https) | 156 | (connect-src, unsafe-eval) |
| 157 | (script-src-elem, *.exdo) | 158 | (frame-src, unsafe-eval) | 159 | (style-src, report-sample) |
| 160 | (worker-src, unsafe-inline) | 161 | (connect-src, ws) | 162 | (object-src, unsafe-inline) |
| 163 | (child-src, unsafe-inline) | 164 | (style-src-elem, exdo) | 165 | (sandbox, allow-scripts) |
| 166 | (connect-src, http) | 167 | (worker-src, none) | 168 | (sandbox, allow-same-origin) |
| 169 | (prefetch-src, exdo) | 170 | (sandbox, allow-forms) | 171 | (sandbox, allow-popups) |
| 172 | (font-src, http) | 173 | (script-src-elem, unsafe-eval) | 174 | (form-action, *) |
| 175 | (object-src, unsafe-eval) | 176 | (worker-src, unsafe-eval) | 177 | (sandbox, allow-popups-to-escape-sandbox) |
| 178 | (frame-ancestors, https) | 179 | (manifest-src, sado) | 180 | (script-src, unsafe-hashes) |
| 181 | (child-src, http) | 182 | (media-src, http) | 183 | (style-src-elem, *.exdo) |
| 184 | (style-src-attr, unsafe-inline) | 185 | (media-src, none) | 186 | (manifest-src, exdo) |

| | | | | | |
|---|---|---|---|---|---|
| 187 | (base-uri, *.sado) | 188 | (media-src, other scheme) | 189 | (script-src-attr, unsafe-inline) |
| 190 | (form-action, unsafe-inline) | 191 | (sandbox, allow-presentation) | 192 | (child-src, unsafe-eval) |
| 193 | (manifest-src, unsafe-inline) | 194 | (media-src, unsafe-eval) | 195 | (script-src-elem, *.sado) |
| 196 | (object-src, http) | 197 | (base-uri, exdo) | 198 | (font-src, other scheme) |
| 199 | (default-src, unsafe-hashes) | 200 | (prefetch-src, *.exdo) | 201 | (script-src-elem, sado) |
| 202 | (style-src-attr, self) | 203 | (frame-src, none) | 204 | (manifest-src, *.sado) |
| 205 | (frame-src, http) | 206 | (style-src, unsafe-hashes) | 207 | (base-uri, sado) |
| 208 | (form-action, http) | 209 | (manifest-src, *.exdo) | 210 | (style-src, other scheme) |
| 211 | (script-src-attr, self) | 212 | (script-src-elem, https) | 213 | (script-src-attr, none) |
| 214 | (child-src, other scheme) | 215 | (child-src, none) | 216 | (prefetch-src, *) |
| 217 | (prefetch-src, *.sado) | 218 | (frame-ancestors, data) | 219 | (prefetch-src, sado) |
| 220 | (base-uri, *.exdo) | 221 | (style-src-elem, *) | 222 | (style-src-elem, *.sado) |
| 223 | (script-src-elem, *) | 224 | (base-uri, unsafe-inline) | 225 | (form-action, none) |
| 226 | (frame-ancestors, unsafe-inline) | 227 | (plugin-types, " ") | 228 | (default-src, nonce-) |
| 229 | (manifest-src, *) | 230 | (style-src, nonce-) | 231 | (script-src-elem, blob) |
| 232 | (frame-ancestors, blob) | 233 | (sandbox, allow-modals) | 234 | (style-src, sha256-) |
| 235 | (style-src-elem, sado) | 236 | (form-action, unsafe-eval) | 237 | (default-src, report-sample) |
| 238 | (img-src, unsafe-hashes) | 239 | (manifest-src, unsafe-eval) | 240 | (manifest-src, none) |
| 241 | (manifest-src, data) | 242 | (prefetch-src, unsafe-inline) | 243 | (script-src-elem, data) |
| 244 | (connect-src, unsafe-hashes) | 245 | (frame-src, nonce-) | 246 | (media-src, report-sample) |
| 247 | (script-src, wss) | 248 | (worker-src, unsafe-hashes) | 249 | (worker-src, http) |
| 250 | (style-src-elem, https) | 251 | (style-src-elem, data) | 252 | (script-src-attr, unsafe-eval) |
| 253 | (form-action, other scheme) | 254 | (frame-ancestors, unsafe-eval) | 255 | (font-src, unsafe-hashes) |
| 256 | (frame-src, unsafe-hashes) | 257 | (manifest-src, unsafe-hashes) | 258 | (media-src, unsafe-hashes) |
| 259 | (object-src, unsafe-hashes) | 260 | (prefetch-src, unsafe-eval) | 261 | (prefetch-src, unsafe-hashes) |
| 262 | (style-src-attr, *) | 263 | (form-action, unsafe-hashes) | 264 | (form-action, data) |
| 265 | (sandbox, allow-top-navigation-by-user-activation) | 266 | (connect-src, nonce-) | 267 | (font-src, none) |
| 268 | (manifest-src, https) | 269 | (style-src-elem, unsafe-eval) | 270 | (style-src-elem, blob) |
| 271 | (script-src-attr, *) | 272 | (base-uri, *) | 273 | (sandbox, allow-pointer-lock) |
| 274 | (font-src, report-sample) | 275 | (media-src, nonce-) | 276 | (object-src, report-sample) |
| 277 | (prefetch-src, https) | 278 | (worker-src, wss) | 279 | (script-src-elem, unsafe-hashes) |
| 280 | (script-src-elem, report-sample) | 281 | (script-src-elem, nonce-) | 282 | (style-src-attr, unsafe-hashes) |
| 283 | (script-src-attr, unsafe-hashes) | 284 | (script-src-attr, https) | 285 | (base-uri, https) |
| 286 | (frame-ancestors, http) | 287 | (sandbox, allow-downloads) | 288 | (sandbox, allow-top-navigation) |
| 289 | (child-src, strict-dynamic) | 290 | (child-src, wss) | 291 | (frame-src, report-sample) |
| 292 | (img-src, report-sample) | 293 | (prefetch-src, none) | 294 | (prefetch-src, data) |
| 295 | (script-src, ws) | 296 | (style-src, wss) | 297 | (worker-src, report-sample) |
| 298 | (worker-src, ws) | 299 | (style-src-elem, unsafe-hashes) | 300 | (style-src-attr, https) |
| 301 | (style-src-attr, data) | 302 | (script-src-attr, blob) | 303 | (base-uri, data) |
| 304 | (form-action, report-sample) | 305 | (sandbox, allow-orientation-lock) | 306 | (child-src, report-sample) |
| 307 | (connect-src, report-sample) | 308 | (default-src, strict-dynamic) | 309 | (font-src, wss) |
| 310 | (img-src, nonce-) | 311 | (img-src, wss) | 312 | (object-src, other scheme) |
| 313 | (object-src, nonce-) | 314 | (script-src, sha384-) | 315 | (style-src, strict-dynamic) |
| 316 | (style-src, ws) | 317 | (worker-src, other scheme) | 318 | (style-src-elem, nonce-) |
| 319 | (script-src-elem, sha256-) | 320 | (script-src-elem, http) | 321 | (style-src-attr, report-sample) |
| 322 | (style-src-attr, *.exdo) | 323 | (script-src-attr, exdo) | 324 | (base-uri, other scheme) |
| 325 | (form-action, nonce-) | 326 | (frame-ancestors, report-sample) | 327 | (require-trusted-types-for, script) |
| 328 | (child-src, nonce-) | 329 | (connect-src, strict-dynamic) | 330 | (connect-src, none) |
| 331 | (default-src, sha256-) | 332 | (font-src, nonce-) | 333 | (font-src, ws) |
| 334 | (frame-src, strict-dynamic) | 335 | (frame-src, wss) | 336 | (manifest-src, http) |
| 337 | (manifest-src, blob) | 338 | (media-src, wss) | 339 | (object-src, strict-dynamic) |
| 340 | (worker-src, strict-dynamic) | 341 | (worker-src, nonce-) | 342 | (style-src-elem, strict-dynamic) |
| 343 | (style-src-elem, report-sample) | 344 | (style-src-elem, sha256-) | 345 | (style-src-elem, http) |
| 346 | (script-src-elem, strict-dynamic) | 347 | (script-src-elem, wss) | 348 | (style-src-attr, *.sado) |
| 349 | (style-src-attr, sado) | 350 | (style-src-attr, exdo) | 351 | (script-src-attr, report-sample) |
| 352 | (script-src-attr, *.exdo) | 353 | (base-uri, unsafe-eval) | 354 | (base-uri, http) |
| 355 | (form-action, blob) | 356 | (navigate-to, *) | 357 | (require-sri-for, script style) |
| 358 | (trusted-types, policyname) | | | | |

## C. Some Website and CSP Examples

### C.1. Example Websites Built through Web Development Platforms with Identical CSPs

We analyzed in Section 4.3.5 that some websites were built through the same web development platform, and they deployed the identical CSP. For example, websites *https://huel.com*, *https://goli.com*, and *https://vessi.com/* were built through Shopify. All of them deployed the same CSP which is *"block-all-mixed-content; frame-ancestors 'none'; upgrade-insecure-requests;"*.

### C.2. Example Websites with Severe Problems in Their CSPs

We analyzed in Section 4.3.6 that websites in some clusters had severe problems in their CSPs. The following are some examples. In Cluster 1, the CSP of *https://sina.cn/* only contained the "upgrade-insecure-requests" directive, which did not cover any type of resources needed in the webpage. In Cluster 5, the CSP of *https://www.vqfit.com/* was *"block-all-mixed-content; frame-ancestors *; upgrade-insecure-requests;"*. The "frame-ancestors *" directive allows any webpages to embed the current webpage. In Cluster 8, the CSP of *https://www.cope.es/* was *"default-src 'unsafe-inline' 'unsafe-eval' 'self' data: blob: wss://* http://* https://*;"*, which included unsafe directive values. The 'unsafe-inline' directive value allows any inline scripts and stylesheets to be loaded in the webpage. The 'unsafe-eval' directive value allows any eval-like functions to be executed in the webpage. Meanwhile, the "data:", "blob:", "wss://*", "http://*", and "https://*" directive values allow resources from any external sources to be loaded.

### C.3. A Secure CSP Example for Web Developers to Use as A Reference in Their CSP Deployment

The CSP deployed on *https://www.xrptoolkit.com/* (as shown in Figure 6) is a secure CSP example from both the directive coverage and the secure use perspectives. It is used for XSS mitigation, framing control, and TLS enforcement. The policy features of the CSP are "default-src 'none' ", "object-src 'none' ", script-src 'self' ", "manifest-src 'self' ", "style-src 'self' ", "img-src 'self' ", "img-src exdo", "connect-src exdo", "frame-ancestors 'none' ", "base-uri 'none' ", "form-action 'none' ", and "block-all-mixed-content". All these policy features of this CSP are safe. With the "default-src 'none' " directive, all types of resources can be controlled by the CSP for XSS mitigation. The "block-all-mixed-content" directive is used for TLS enforcement. For framing control, the "frame-ancestors 'none' " directive does not allow any webpages to embed a current webpage.

```
Webpage:https://www.xrptoolkit.com/

default-src 'none';
object-src 'none';
script-src 'self';
manifest-src 'self';
style-src 'self';
img-src 'self' https://www.gravatar.com https://xumm.app;
connect-src wss://xrplcluster.com wss://xrpl.link
          wss://s2.ripple.com https://data.ripple.com
          https://xumm.xrptoolkit.com wss://xumm.app
          https://lookup.xrptoolkit.com
          https://api.rollbar.com;
frame-ancestors 'none';
base-uri 'none';
form-action 'none';
block-all-mixed-content;
report-to csp
```

Figure 6: The CSP Deployed on *https://www.xrptoolkit.com/*