TOWARD SECURE CONTENT SECURITY POLICY DEPLOYMENT

by

Mengxia Ren

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Computer Science).

Golden, Colorado

Date ________________________

Signed: ________________________

Mengxia Ren

Signed: ________________________

Dr. Chuan Yue
Thesis Advisor

Golden, Colorado

Date ________________________

Signed: ________________________

Dr. Iris Bahar
Professor and Department Head
Department of Computer Science

ABSTRACT

Content Security Policy (CSP) is a standardized leading technique for protecting webpages against attacks such as Cross Site Scripting (XSS). 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. A CSP is composed of a set of directives, and each directive is typically composed of a directive name with a set of directive values. To effectively protect a webpage, a CSP should be carefully designed according to resources and behaviors on the protected webpage. However, it is often hard to properly deploy CSPs on webpages, and the deployed CSPs often contain security issues or errors. Therefore, helping developers properly deploy CSPs on their websites is important to the enhancement of web security.

This dissertation concentrates on promoting the proper deployment of CSPs by investigating the protection capability of deployed CSPs, analyzing CSP deployment issues, and exploring the feasibility of adopting secure CSP solutions on websites. We first investigated the security levels of the deployed CSPs from the directive coverage and secure use perspectives by taking an unsupervised clustering approach which can automatically categorize very diverse and complex CSPs. Next, we focused on investigating CSP deployment issues related to third-party scripts (i.e., JavaScript code or script files) among different websites by conducting a measurement study. We analyzed the usage of third-party scripts on websites based on the collected CSP violations triggered by the accesses of resources used on websites under our inserted CSP. Furthermore, we evaluated the feasibility of taking Google's secure CSP approach on websites. The evaluation is based on CSP violations that are triggered under the four inserted CSPs of the secure CSP approach. A large-scale web measurement usually relies on a crawler. We also studied two browser-based crawler implementation approaches in the context of the Google Chrome browser and further built a Google Chrome extension named WebMea as a baseline Google Chrome extension that can measure multiple types of web data.

TABLE OF CONTENTS

ABSTRACT iii

LIST OF FIGURES x

LIST OF TABLES xii

ACKNOWLEDGMENTS xiii

CHAPTER 1 INTRODUCTION 1

1.1 Overview of CSP and Its Deployment Issues 1

1.2 Objectives 1

1.3 Contributions 3

1.3.1 Coverage and Secure Use Analysis of CSPs via Clustering 3

1.3.2 Analysis of Third-Party Scripts Related CSP Deployment Issues among Different Websites 3

1.3.3 Analysis of the Feasibility of Taking Google's Secure CSP Approach on Websites 4

1.3.4 Study of Browser-Based Crawler Implementation Approaches 4

1.4 Related Publications 5

CHAPTER 2 COVERAGE AND SECURE USE ANALYSIS OF CONTENT SECURITY POLICIES VIA CLUSTERING 6

2.1 Introduction 6

2.2 Background and Related work 8

2.2.1 Background of CSP 8

2.2.2 Related Work 9

2.3 Design and Methodology 11

2.3.1 Data Collection Tool and Process 11

2.3.2 Policy Feature Design and Extraction 11

2.3.3 Clustering Algorithm Design 14

2.3.3.1 Contrastive Spectral Clustering (CSC) Algorithm 15

2.3.3.2 Contrastive Learning (CL) Algorithm 17

2.3.4 CSP Security Level Analysis Methods . . . 18
2.4 Results and Analysis . . . 19
2.4.1 CSP Dataset . . . 19
2.4.2 CSC Algorithm Evaluation . . . 21
2.4.2.1 Experiments and Evaluation Metrics . . . 21
2.4.2.2 Implementation of the Algorithms . . . 22
2.4.2.3 Evaluation Results . . . 22
2.4.3 CSP Clustering Results and Analysis . . . 23
2.4.3.1 Group 1 . . . 24
2.4.3.2 Group 2 . . . 26
2.4.3.3 Group 3 . . . 26
2.4.3.4 Group 4 . . . 27
2.4.3.5 Potential Reasons for Some CSP Patterns . . . 28
2.4.3.6 Summary of the Analysis Results . . . 30
2.4.4 CSP Deployment on Subpages . . . 32
2.5 Takeaways and Recommendations . . . 32
2.5.1 High-level Takeaways from Our Study . . . 32
2.5.2 Recommendations for Web Developers . . . 33
2.6 Summary . . . 33
CHAPTER 3 ANALYSIS OF THIRD-PARTY SCRIPTS RELATED CONTENT SECURITY POLICY DEPLOYMENT ISSUES . . . 34
3.1 Introduction . . . 34
3.2 Background and Related Work . . . 37
3.2.1 Background of CSP . . . 37
3.2.2 Related Work . . . 38
3.3 Methodology . . . 39
3.3.1 Website Crawling and Data Collection Framework . . . 39
3.3.2 Data Collection Process and Dataset . . . 41

3.3.3 Data Processing and Analysis . . . 42
3.3.3.1 Webpage Type Identification . . . 42
3.3.3.2 Third-party Script Type Identification . . . 42
3.3.3.3 Data Analysis Techniques . . . 44
3.3.4 Ethical Considerations and Reproducibility . . . 45
3.4 Results and Analysis . . . 45
3.4.1 Third-Party Script Resources in Builder-generated and Other Websites . . . 45
3.4.1.1 Popularity of the 10 Types of Third-Party Scripts . . . 45
3.4.1.2 Domains of the Included Third-Party Scripts . . . 47
3.4.1.3 Websites Built on Different Web Builders . . . 48
3.4.1.4 Third-party Scripts' Domain Changes . . . 49
3.4.1.5 CSP Support in Web Builders . . . 50
3.4.2 Third-Party Script Resources in CDN-enabled Websites and Other Websites . . . 51
3.4.2.1 Popularity of the 10 Types of Third-Party Scripts . . . 51
3.4.2.2 Domains of the Included Third-Party Scripts . . . 52
3.4.2.3 Websites Enabled by Different CDNs . . . 53
3.4.2.4 Third-party Scripts' Domain Changes . . . 54
3.4.3 Potential CSP Improvement or Deployment on Different Types of Websites . . . 55
3.4.3.1 CSP Improvement on CSP-deployed Websites . . . 55
3.4.3.2 Safe CSP Deployment on Websites without CSPs . . . 57
3.5 Discussion . . . 58
3.5.1 Summary of Key Results . . . 58
3.5.2 Recommendations . . . 59
3.5.3 Limitations and Future Work . . . 60
3.6 Summary . . . 60
CHAPTER 4 ANALYZING THE FEASIBILITY OF TAKING GOOGLE'S SECURE CSP APPROACH ON WEBSITES . . . 61
4.1 Introduction . . . 61

4.2 Background and Related Work . . . 64

4.2.1 Background of CSP . . . 64

4.2.2 Related Work . . . 65

4.3 Design of Our Study . . . 67

4.3.1 Website Crawling and Data Collection Framework . . . 68

4.3.2 Data Collection and Dataset . . . 70

4.3.3 Evaluation Methods for the Feasibility of Adopting Nonce-Based CSP Solutions . . . 70

4.3.3.1 Feasibility Evaluation of Adopting Nonce-based CSP Solutions on Websites . . . 71

4.3.3.2 Evaluation for the Highest Security Level of CSP Solutions on Webpages . . . 71

4.3.3.3 Third-party Resource Identification . . . 72

4.3.4 Ethical Considerations and Reproducibility . . . 72

4.4 Results and Analysis . . . 73

4.4.1 Adoptability of the Nonce-Based CSP solutions . . . 74

4.4.1.1 Adoption Patterns of Nonce-Based CSP Solutions Across Websites . . . 74

4.4.1.2 Potential CSP Improvement . . . 76

4.4.2 Adoption Issues of Nonce-Based CSP Solutions . . . 77

4.4.2.1 Upgrade to the Weaker Nonce-Based CSP Solutions . . . 77

4.4.2.2 Upgrade to the Stronger Nonce-Based CSP Solutions . . . 78

4.4.2.3 Promotion of Adopting Nonce-Based CSP Solutions by Addressing Adoption Issues Caused by First-Party Resources . . . 79

4.4.3 Stability of the Nonce-Based CSP Solution Adoption . . . 80

4.5 Discussion . . . 84

4.5.1 Summary of Key Findings . . . 84

4.5.2 Recommendations . . . 84

4.5.3 Limitations and Future Work . . . 85

4.6 Summary . . . 85

CHAPTER 5 WEBMEA: A GOOGLE CHROME EXTENSION FOR WEB SECURITY AND PRIVACY MEASUREMENT STUDIES . . . 86

5.1 Introduction 86
5.2 Related Work 88
5.3 Comparing the Two Crawler Implementation Approaches 89
5.3.1 Primary Requirements for Crawlers 89
5.3.2 Google Chrome DevTools and Extension Approaches 90
5.3.3 DevTools Case Study: Puppeteer 93
5.3.4 Chrome Extension Case Study: TrackingObserver 93
5.4 Proposed Chrome Extension WebMea 94
5.4.1 Design of WebMea 94
5.4.2 Implementation of WebMea 95
5.4.3 Customization of WebMea 97
5.5 Summary 98
CHAPTER 6 CONCLUSION 99
6.1 Conclusion 99
6.2 Future Work 100
REFERENCES 101
APPENDIX A APPENDICES OF CHAPTER 2 106
A.1 Other Parameters in Our Algorithms and Combinations of Data Augmentation Operators . . 106
A.2 The 358 Ranked Policy Features 106
A.3 Some Website and CSP Examples 109
A.3.1 Example Websites Built through Web Development Platforms with Identical CSPs . . 109
A.3.2 Example Websites with Severe Problems in Their CSPs 109
A.3.3 A Secure CSP Example for Web Developers to Use as A Reference in Their CSP Deployment 110
APPENDIX B APPENDICES OF CHAPTER 3 111
B.1 The 64 Web Builders That Have a Minimum of 10 Websites Built on Them 111
B.2 The 12 Web Builders That Host and Supply Third-party Scripts 112
B.3 The 26 CDNs That Have a Minimum of 10 Websites Using Them 112

APPENDIX C APPENDICES OF CHAPTER 4 113

C.1 The Types of CSP violations that can be Resolved under Google's Nonce-Based CSP Solutions 113

LIST OF FIGURES

Figure 1.1 An Example of Deploying A CSP on A Webpage . . . 2

Figure 2.1 Ranked Popularity of the 358 Features among the 13,317 Homepages . . . 21

Figure 2.2 The Silhouette Scores of Algorithms under Different Numbers of Clusters . . . 22

Figure 2.3 The NMI Scores based on 1,000 Randomly Sampled Examples . . . 23

Figure 2.4 TSNE Visualization of the 16 Clusters . . . 24

Figure 2.5 Feature Popularity of Each Cluster . . . 25

Figure 3.1 Website Crawling and Data Collection Framework . . . 40

Figure 3.2 The Popularity of the 10 Types of Third-party Script Resources between Builder-generated Websites (n=22,105) and Other Websites (n=39,722) . . . 46

Figure 3.3 CDF Distributions of First-party Builder-generated Websites (n=22,105) and Other Websites (n=39,722) regarding the Number of Domains of the Included Third-party Scripts . . . 48

Figure 3.4 The Popularity of the 10 Types of Third-party Script Resources that Change Domains between the Two Crawdlings among Builder-generated Websites (n=19,090) and Other Websites (n=36,126) . . . 50

Figure 3.5 The Popularity of the 10 Types of Third-party Script Resources between CDN-enabled Websites (n=27,332) and Other Websites (n=34,495) . . . 51

Figure 3.6 CDF Distributions of First-party CDN-enabled Websites (n=27,332) and Other Websites (n=34,495) regarding the Number of Domains of the Included Third-party Scripts . . . 53

Figure 3.7 The Popularity of the 10 Types of Third-party Script Resources that Change Domains between the Two Crawlings among CDN-enabled Websites (n=22,110) and Other Websites (n=30,737) . . . 54

Figure 4.1 Website Crawling and Data Collection Framework . . . 68

Figure 4.2 Four Inserted CSPs for Webpages . . . 69

Figure 4.3 The Lowest Security Level of Adoptable Nonce-Based CSP Solutions across 7,032 Websites . . . 75

Figure 4.4 Distribution of the webpages visited in our study regarding what percentages of them can adopt nonce-based CSP solutions at levels 1, 2, 3, and 4, as well as cannot adopt any nonce-based CSP solutions, respectively, across 2,610 websites that cannot adopt nonce-based CSP solutions. The x marks represent the mean values. . . . 75

Figure 4.5 Distribution of the webpages visited in our study regarding what percentages of them can adopt nonce-based CSP solutions at levels 1, 2, 3, and 4, respectively, across 4,422 websites that can adopt nonce-based CSP solutions. The x marks represent the mean values. 76

Figure 4.6 The Lowest Security Level Changes of Adoptable Nonce-Based CSP Solutions across 7,032 Websites after Addressing First-Party Resources that Can Cause Adoption Issues . . 80

Figure 5.1 Major Categories of Capabilities of the Chrome DevTools Protocol (CDP) APIs and the Google Chrome extension APIs. Google Chrome extensions can also access 15 CDP API domains through the *chrome.debugger* extension API. 91

Figure 5.2 The High-level Overview of WebMea 94

Figure A.1 The CSP Deployed on *https://www.xrptoolkit.com/* 110

LIST OF TABLES

Table 2.1 Directive Names and the Allowed Directive Value Types (defined in Table 2.2) or Values . . 12

Table 2.2 Directive Value Types and Value Token Types 13

Table 2.3 Summary of the Main Aims of the CSPs and the CSP Security Level Analysis Results for the 16 Clusters 30

Table 3.1 10 Types of Third-party Script Resources Related to CSP Deployment Issues. The word "using" in the third column means that web developers may need to use certain unsafe directive values (e.g., "'unsafe-inline'", " 'unsafe-eval'", or "*") in their CSPs. 44

Table 4.1 Google's Four Nonce-based CSP Solutions Categorized by Decreasing Security Levels . . . 67

Table 5.1 The major CDP APIs and Google Chrome extension APIs of the two approaches for satisfying the primary web measurement requirements. 92

Table A.1 NMI Scores of the Experiments for Six Combinations of Three Data Augmentation Operators. 106

Table A.2 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. 107

Table B.1 Details of the 64 web builders that have a minimum of 10 websites built on them. The "*" indicates web builders with websites built on them having a tendency to deploy CSPs with an "'unsafe-inline'" directive value. The "Δ" indicates web builders with websites built on them having a tendency to deploy CSPs with an "'unsafe-eval'" directive value. . 111

Table B.2 Details of the 12 web builders that have a minimum of 10 websites built on them in the first crawling, and they supply third-party scripts to more than 50% of the websites built on them. The "*" indicates web builders that provide third-party scripts which can lead to the usage of an "'unsafe-inline'" directive value in CSPs. The "Δ" indicates web builders that provide third-party scripts which can lead to the usage of an "'unsafe-eval'" directive value in CSPs. 112

Table B.3 Details of the 26 CDNs that have a minimum of 10 websites using them. The "▲" indicates CDNs that provide third-party scripts to more than 50% of the websites enabled by them. The "*" indicates CDNs that provide third-party scripts which can lead to the usage of an "'unsafe-inline'" directive value in CSPs. The "Δ" indicates CDNs that provide third-party scripts which can lead to the usage of an "'unsafe-eval'" directive value in CSPs. 112

Table C.1 Details of the types of CSP violations that can be resolved under Google's nonce-based CSP solutions. 114

ACKNOWLEDGMENTS

First and foremost, I would like to express my deep and sincere gratitude to my research supervisor, Professor Chuan Yue for his invaluable guidance, support, and patience throughout the course of my research. I sincerely thank him for the continuous support of my Ph.D. study and research, patience, enthusiasm, and immense knowledge. His guidance helped me throughout the research and writing of this dissertation. Besides my advisor, I am deeply grateful to thank the rest of my committee, Professor Qiaohai Hu, Professor Qi Han, and Professor Mehmet Belviranli, for their generous support, valuable suggestions, and insightful comments, which significantly contributed to enhancing the quality of this dissertation. Furthermore, I would like to express my sincere appreciation to Anhao Xiang and Joshua Josey for their important contributions to my research projects. I also want to sincerely thank the amazing colleagues I met and worked with in the research group under the supervision of Professor Chuan Yue: Zhiju Yang, Weiping Pei, Ahmed Alshehri, Khalid Alkhattabi, and Anhao Xiang.

Outside the research group, I would like to extend my heartfelt gratitude to my dear friends, especially Xingyuan (I am very happy to graduate with you!), who have been integral to my Ph.D. journey. Your unwavering support and companionship have provided me with the strength to persevere through challenges and brought joy and laughter into my days. Finally, I would like to express my deepest gratitude to my parents for their countless sacrifices to provide me with the opportunities I have today, and it is their encouragement that has empowered me to pursue my passions and goals with determination.

Thank you to Mines, where I have spent a wonderful five years!

CHAPTER 1

INTRODUCTION

In this chapter, we first introduce the overview of Content Security Policy (CSP) and its deployment issues. We then outline the central theme of this dissertation and present our contributions.

1.1 Overview of CSP and Its Deployment Issues

CSP is a standardized leading technique for protecting webpages against code and data injection attacks such as Cross Site Scripting (XSS), and it has been successfully adopted by all major web browsers. Figure 1.1 illustrates an example of deploying a CSP on a webpage. First, website developers analyze the resources and behaviors on their webpages. Next, they design CSPs for their webpages according to the used resources (e.g., scripts and images) and possible user behaviors (e.g., form submissions). A CSP is composed of a set of directives, each of which is typically composed of a directive name with a set of directive values. Typically, a directive name (e.g., "script-src", "img-src", and "form-action") specifies a certain resource type that will be controlled, while a directive value (e.g., 'nonce-1', "https://img.com", and "www.example.com") specifies which resources can be loaded or what behaviors can be allowed on a webpage. For example, the "img-src https://img.com" directive in Figure 1.1 is composed of an ***img-src*** directive name with an "https://img.com" directive value. This directive allows images from ***https://img.com*** to be loaded on a webpage. Then, website developers can deploy designed CSPs on their webpages via HTTP(S) response headers or HTML <meta> tags. A browser will load resources according to the specified CSPs on webpages. When a resource or a behavior violates the CSP deployed on a webpage, there will be a CSP violation.

However, the CSP adoption ratio has always been low, and it is hard to properly deploy CSPs. For example, CSPs appeared on only 17.74% of the Alexa top-100K websites in 2022 [1]. Meanwhile, security issues or errors are often found in the deployed CSPs. For example, Weichselbaum et al. found that 94.72% of CSPs could not protect webpages against XSS attacks because of policy misconfigurations and insecure whitelisted entries [2]. Meanwhile, the root causes of CSP deployment issues can be complex. For example, the legacy code, third-party code, and the complexity of the CSP mechanism could all block a successful CSP deployment [3, 4].

1.2 Objectives

The aim of this dissertation is to promote the proper deployment of CSPs by investigating CSP deployment issues and solutions. Specifically, we focus on investigating the following research questions:

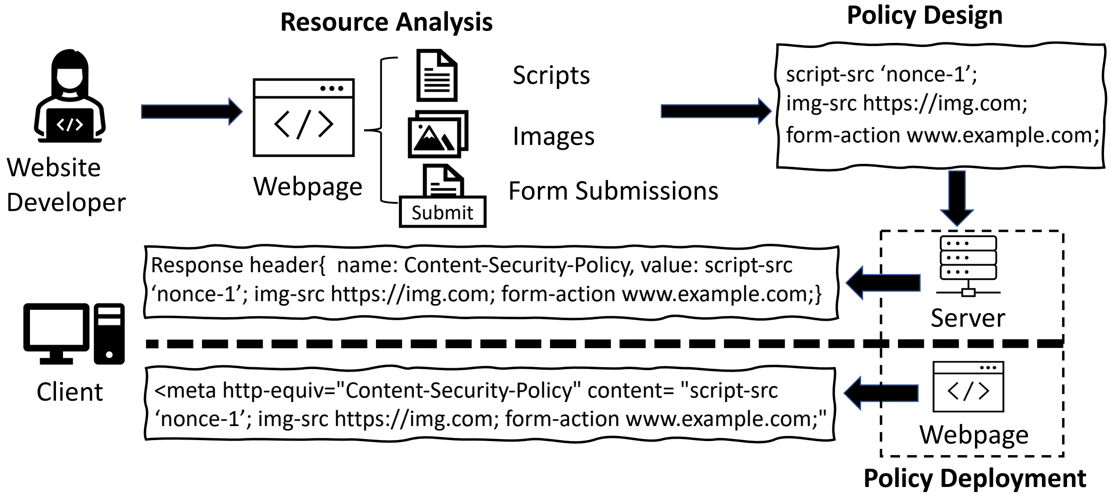


Figure 1.1 An Example of Deploying A CSP on A Webpage

Can deployed CSPs protect websites effectively? What CSP deployment issues would websites encounter? How can web developers properly deploy CSPs on their websites? The first research question focuses on evaluating the protection capability of deployed CSPs. The second research question focuses on investigating potential CSP deployment issues among websites. The third research question focuses on exploring solutions for websites to properly deploy CSPs. Each work in this dissertation is conducted to answer one of these three research questions. By answering these questions, we hope to provide a better understanding of the CSP deployment issues faced by websites and offer feasible CSP deployment solutions to promote the proper deployment of CSPs.

As for analyzing the protection capability of deployed CSPs, we investigated their security levels from the directive coverage and secure use perspective by taking the unsupervised clustering approach in Chapter 2. As for analyzing potential CSP deployment issues among websites, we conducted a measurement study to analyze third-party scripts related CSP deployment issues among different websites including builder-generated websites and CDN-enabled websites in Chapter 3. As for exploring the solutions for websites to properly deploy CSPs, we conducted a measurement study to evaluate the feasibility of taking Google’s secure CSP approach on websites and identified related adoption issues in Chapter 4. We also studied two browser-based crawler implementation approaches in the context of the Google Chrome browser and further built a Google Chrome extension named WebMea as a baseline Google Chrome extension that can measure multiple types of web data in Chapter 5.

1.3 Contributions

The contributions of this dissertation include investigating coverage and secure use of CSPs via clustering, analyzing third-party scripts related CSP deployment issues among different websites, exploring the feasibility of taking Google's secure CSP approach on websites, and studying browser-based crawler implementation approaches.

1.3.1 Coverage and Secure Use Analysis of CSPs via Clustering

Security issues or errors are often found in the deployed CSPs. 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. However, existing studies analyzed the overall statistics of CSP security issues based on some specific rules (or heuristics). Meanwhile, they mainly analyzed the vulnerabilities of specific directives (e.g., "script-src", "object-sr", and "default-src"), but did not estimate the overall or combined protection capabilities of CSPs.

In this work, we proposed to take the clustering approach for analyzing the security levels of the deployed CSPs from the directive coverage and secure use perspectives. We analyzed the protection capabilities of a CSP based on all its directives. Specifically, we designed and extracted policy features based on the latest CSP Level 3 specification, and designed an algorithm that leverages the advantages of spectral clustering and contrastive learning for automatically categorizing CSPs. We performed a large-scale measurement study, categorized the CSPs into clusters with different characteristics, and analyzed the security levels of the CSPs in each unique cluster to help promote the proper deployment of CSPs. We detail this work in Chapter 2.

1.3.2 Analysis of Third-Party Scripts Related CSP Deployment Issues among Different Websites

Recent studies found that it is often hard to properly deploy CSPs on websites that include third-party scripts due to the need for the incorporation of third parties. The improper blockage of essential third-party scripts often breaks website functionalities, and third-party resource changes also make it difficult to maintain or update CSPs. However, the usage of third-party scripts can be influenced by web technologies. Existing studies did not categorize the websites with different web technologies and did not study the impacts of web technologies on the resource usage and thus the proper deployment of CSPs.

In this work, we analyzed CSP deployment issues related to third-party scripts among websites with different web technologies including builder-generated websites and CDN-enabled websites by performing a large-scale measurement study. Specifically, we designed a browser extension to directly insert a specified CSP under the report-only mode to each homepage to collect CSP violations triggered by the accesses of resources. This provides a straightforward way for us to analyze the resource usage and CSP deployment issues related to third-party scripts on websites. We found that the choice of web builders and CDNs can impact the CSP deployment on the corresponding websites. Additionally, most CSP-deployed websites can deploy safer CSPs, and most websites without CSPs can deploy safe CSPs. We detail this work in Chapter 3.

1.3.3 Analysis of the Feasibility of Taking Google's Secure CSP Approach on Websites

Google recently explored a state-of-the-art secure CSP approach promoting four nonce-based CSP solutions that are simpler and more secure compared to traditional whitelisting-based CSP solutions. They successfully implemented this approach in more than 160 services, effectively covering 62% of all outgoing Google traffic. This secure CSP approach provides fine-grained control of web resources and can help address CSP deployment problems. However, existing studies mainly focus on investigating whitelisting-based CSP solutions, and no study has explored the feasibility of taking this nonce-based secure CSP approach on other websites yet.

In this work, we investigated the feasibility of taking Google's secure CSP approach on websites by conducting a large-scale measurement study. Specifically, we constructed a crawling tool to automatically visit landing pages and subpages of websites and insert specific CSPs under the report-only mode to webpages to collect corresponding CSP violations. The analysis of the feasibility of taking Google's secure CSP approach on websites is based on CSP violations triggered under the inserted CSPs. We found that a significant proportion of websites can take Google's secure CSP approach. Websites that cannot take the secure CSP approach are majorly due to third-party resources. Meanwhile, most deployed whitelisting-based CSPs can be converted to nonce-based ones which are more secure. We detail this work in Chapter 4.

1.3.4 Study of Browser-Based Crawler Implementation Approaches

Many large-scale web security and privacy measurement studies rely on crawlers for data collection. However, it is often not clear to researchers which is the more appropriate approach to constructing a crawler since the analysis of these two approaches is limited. In this work, we compare the web measurement capability of two Google Chrome browser-based crawler implementation approaches to help researchers more easily choose or take the appropriate approach. Meanwhile, we build WebMea as a baseline Google Chrome extension for researchers to construct their extension-based crawlers. We detail this work in Chapter 5.

1.4 Related Publications

(1). Chapter 2: Mengxia Ren, and Chuan Yue. "Coverage and Secure Use Analysis of Content Security Policies via Clustering." in *Proceedings of IEEE European Symposium on Security and Privacy (EuroS&P)*, 2023. [1]

(2). Chapter 3: Mengxia Ren, and Chuan Yue. "Analysis of Third-Party Scripts Related Content Security Policy Deployment Issues." to be submitted to a conference, 2024. [5]

(3). Chapter 4: Mengxia Ren, Anhao Xiang, and Chuan Yue. "Analyzing the Feasibility of Taking Google's Secure CSP Approach on Websites." to be submitted to a conference, 2024. [6]

(4). Chapter 5: Mengxia Ren, Joshua Josey, and Chuan Yue. "WebMea: A Google Chrome Extension for Web Security and Privacy Measurement Studies." in *Proceedings of International Conference on Science of Cyber Security (SciSec)*, 2023. [7]

CHAPTER 2
COVERAGE AND SECURE USE ANALYSIS OF CONTENT SECURITY POLICIES VIA CLUSTERING

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 chapter, 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]

2.1 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 [8–10]. 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.

[1]This is a joint work with Chuan Yue [1]. Chuan is an associate professor of computer science at the Colorado School of Mines.

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 chapter). 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 2.2.2).

In this chapter, 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 [11–15]. CSC also integrates contrastive learning, a state-of-the-art self-supervised representation learning technique [16–18], for explicitly learning better (i.e., more informative) representations of CSPs.

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 work: (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 chapter is organized as follows. Section 2.2 reviews the background of CSP and the related work. Section 2.3 presents the design of our study from the data collection, policy feature extraction, clustering algorithm, and security level analysis perspectives. Section 2.4 presents and analyzes the clustering results. Section 2.5 summarizes the high-level takeaways and recommendations for web developers. Section 2.6 concludes this chapter.

2.2 Background and Related work

2.2.1 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 [19, 20]: 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.

2.2.2 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 [21], only 3.7% (out of 106 billion) URLs carried CSPs as reported by Weichselbaum et al. in 2016 [2], and only 1,206 (out of the top 10K) websites deployed CSPs on their homepages as reported by Roth et al. in 2020 [22]. Roth et al. recruited 12 developers for a study with a semi-structured interview, a drawing task, and a programming task [3]; 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 [2, 8, 22, 23]. Weichselbaum et al. found that 94.72% of CSPs could not protect webpages against XSS attacks because of policy misconfigurations and insecure whitelisted entries [2]. Calzavara et al. found that some CSPs were ill-formed (thus invalid) or too strict [23], 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 [8]. 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 [24]. 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 [22]; 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 [2, 8, 22, 23] 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 [22]; 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 [9, 25, 26]. Fazzini et al. designed AutoCSP to generate a CSP for a webpage by using dynamic taint analysis and identifying resources that should be whiltelisted [25]. Pan et al. designed CSPAutoGen to generate a CSP to protect scripts in webpages by importing inline and dynamic scripts as external scripts [26]. 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 [9]. Our effort and results in this chapter could be helpful to the CSP auto-generation research.

In addition, some researchers studied the weaknesses of the CSP standard [9, 27, 28]. Roth et al. found that CSPs can be bypassed by open redirects and script gadgets that would turn non-script data into code [27]. 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 [9]. 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].

2.3 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.

2.3.1 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 2.3.2) and clustering (Section 2.3.3). The collected HTTP(s) requests and HTML documents will be used for analyzing the resource type coverage of CSPs (Section 2.3.4).

2.3.2 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 [19, 20], 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

Table 2.1 Directive Names and the Allowed Directive Value Types (defined in Table 2.2) 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'

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 2.1 lists the 28 directive names included in our policy feature design. They cover all the directive names of four categories as defined in the CSP Level 3 specification [19, 20]. We do not include the two reporting directive names (Section 2.2.1) 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 2.1. 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 2.2. 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.

Table 2.2 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'

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 2.1) with the 43 directive value token types (Table 2.2) by following the third column of Table 2.1 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 [19, 20]. 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.

2.3.3 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 [30]; 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 [11–15]. Meanwhile, recent advances in self-supervised learning especially contrastive learning [16–18], 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 2.3.3.1 and 2.3.3.2, respectively.

2.3.3.1 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 [13–15].

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

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: $D_{i,j} = \begin{cases} \sum_{l=1}^{n} a_{il} & where\ a_{il} \in A, \quad i = j \\ 0, & otherwise \end{cases}$
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 Λ 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: $\quad E_k = \{e_1, ..., e_k\}$, where e_i is the i^{th} eigenvector in $E_{n \times n}$
11: $\quad S = S \cup \{(k,\ E_k,$ 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 $(E_{k_{opt}}, k_{opt})$

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 [13]. 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 [13]) and scoring the clustering results. We choose to use the popular Silhouette score [31] in our Score($\cdot$) 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 [32]. At Line 13, CSC returns k_{opt} clusters.

2.3.3.2 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 [16–18]. Following the SimCLR framework proposed by Chen et al. [16] (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 [16]. 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 [16] in image related tasks, and extracting adjacent and

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** $\bar{X}_{n \times m}$

overlapped textual segments [17] 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 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.

2.3.4 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 [19, 20] and guidelines such as the OWASP XSS Prevention Cheat Sheet [33]. 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" [34]. 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.

2.4 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.

2.4.1 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 2.3.1) 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 [10]; 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 2.3.2. 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 [19, 20]. 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 2.1. Among these 358 features, 219, 70, and 69 are labeled as safe, unsafe, and uncertain, respectively (Section 2.3.4). Figure 2.1 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 A.2 in Appendix A.2.

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

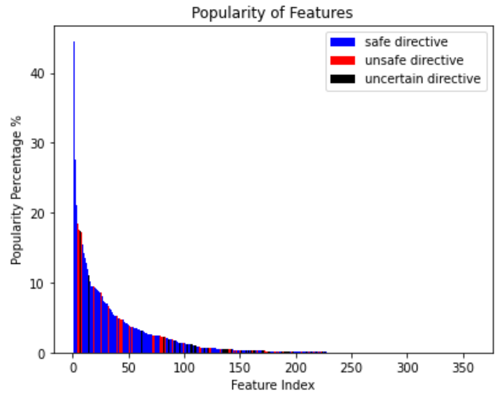


Figure 2.1 Ranked Popularity of the 358 Features among the 13,317 Homepages

that two unsafe features are in the top-6 list, and they all popularly appear on more than 17% of homepages.

2.4.2 CSC Algorithm Evaluation

2.4.2.1 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 2.3.3.1) 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 [35].

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 [36, 37], 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.

2.4.2.2 Implementation of the Algorithms

We implemented our CSC and Cl algorithms using the PyTorch Lightning deep learning framework [38] and the Scikit-learn machine learning library [30]. 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 A.1 in Appendix A.1. Some other implementation details such as the learning rate in the CL algorithm are also provided in Appendix A.1.

2.4.2.3 Evaluation Results

Figure 2.2 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.

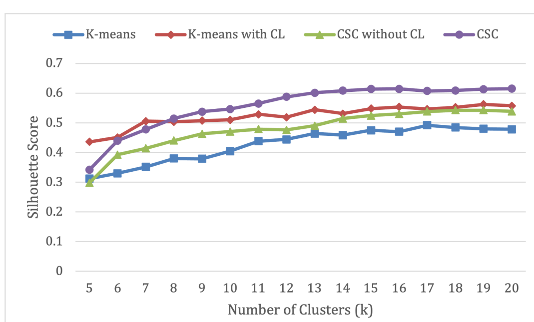


Figure 2.2 The Silhouette Scores of Algorithms under Different Numbers of Clusters

We further calculated the NMI scores for each algorithm after k >15 as shown in Figure 2.3. 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 chapter are all based on our CSC algorithm and its optimal number of clusters k=16.

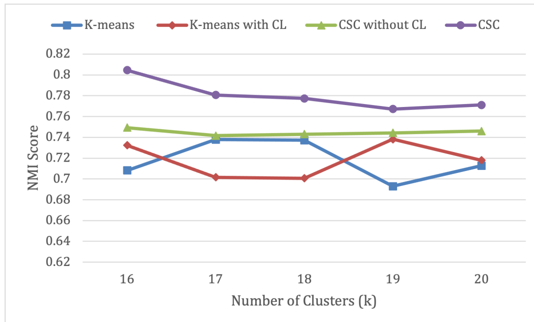


Figure 2.3 The NMI Scores based on 1,000 Randomly Sampled Examples

2.4.3 CSP Clustering Results and Analysis

Our CSC algorithm categorized the 13,317 CSPs into 16 clusters. Figure 2.4 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 2.1 and Table A.2 (in Appendix A.2), Figure 2.5 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 [39]. 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 2.4.3.1-2.4.3.4).

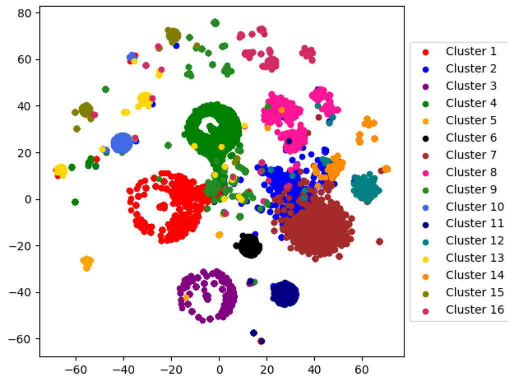


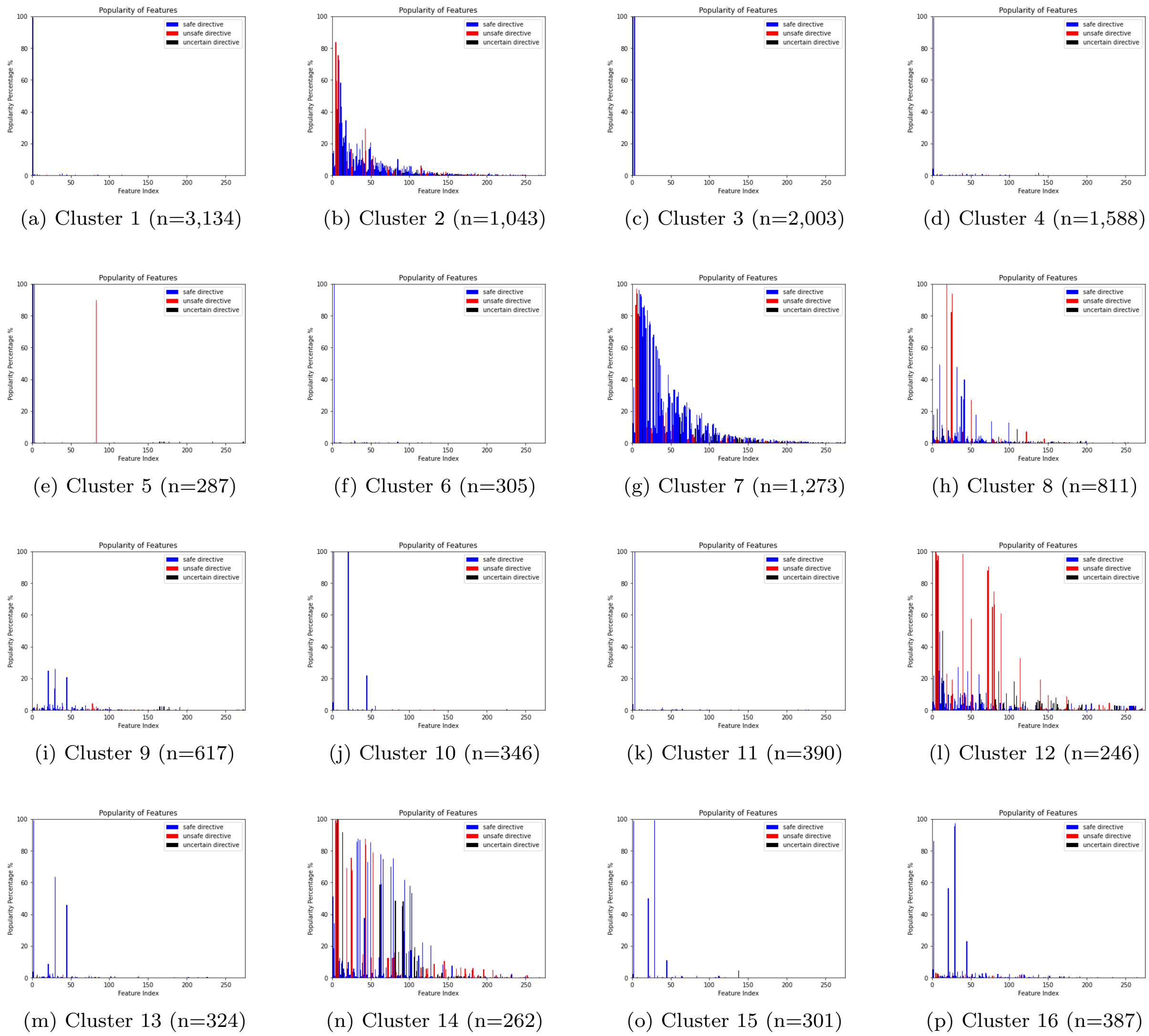
Figure 2.4 TSNE Visualization of the 16 Clusters

2.4.3.1 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



(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 2.5 Feature Popularity of Each Cluster

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 2.3.4), 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 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.

2.4.3.2 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.

2.4.3.3 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 A.2 (in Appendix A.2). 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.

2.4.3.4 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.

2.4.3.5 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 A.3.1) out of those 1,995 webpages in Cluster 3 were built by using Shopify (which is a platform for building shopping websites [40]), 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 [41]). 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 [42]), 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 [43]), 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 [3, 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

Table 2.3 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

clear evidence to show website builders' contribution of these two directives.

2.4.3.6 Summary of the Analysis Results

Overall, based on the 16 subfigures in Figure 2.5 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 2.3. 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 2.3, 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 [8], 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 [2]. Nonces and hashes are not commonly used either, and they are not even in the top-50 features in Table A.2; 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 [3, 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 [2, 8, 22, 23] 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, 8, 21, 22]) 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 A.3.2.

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 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 [10]. 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.

2.5 Takeaways and Recommendations

2.5.1 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 2.4); 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 2.2.2 and 2.3.2. Second, it is important to analyze CSPs from both directive coverage and secure use perspectives (Sections 2.2.2 and 2.3.4); as we summarized in Section 2.4.3.6, 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 2.4.3.1-2.4.3.4); (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 2.4.3.6); (3) web development platforms contributed to the specific CSP patterns of many websites (Section 2.4.3.5 and Appendix A.3.1); (4) several severe problems exist in specific clusters (Section 2.4.3.6 and Appendix A.3.2).

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 2.4.3.6. We further provide a secure CSP example in Appendix A.3.3 for developers to use as a reference in their CSP deployment.

2.6 Summary

In this chapter, 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 [44].

CHAPTER 3

ANALYSIS OF THIRD-PARTY SCRIPTS RELATED CONTENT SECURITY POLICY DEPLOYMENT ISSUES

Content Security Policy (CSP) is a leading security mechanism for mitigating content injection attacks such as Cross-Site Scripting (XSS). However, CSPs are not widely deployed on websites, and deployed CSPs often have security issues or errors. In this chapter, we investigate CSP deployment issues related to third-party scripts among different websites including builder-generated websites and CDN-enabled websites by conducting a web measurement study on Tranco top 100K websites. We construct a Google Chrome extension to automatically visit websites, simulate user interactions, and insert a specified CSP to collect CSP violations triggered by the accesses of resources used on websites. We focus on analyzing the usage of third-party scripts and the CSP deployment issues on websites based on the CSP violations triggered under the inserted CSP. We found that most analyzed websites would not encounter CSP deployment issues. Compared to other websites, builder-generated websites and CDN-enabled websites are more likely to contain third-party scripts which can lead to CSP deployment issues. Meanwhile, some web builders and CDNs provide scripts that can lead to CSP deployment issues, which indicates that the choice of web builders and CDNs can impact the CSP deployment on the corresponding websites. Furthermore, most CSPs that were already deployed on websites can be improved to be safer. For websites that did not deploy CSPs yet, most of them can deploy safe CSPs. Overall, our results can be helpful for promoting the proper deployment of CSPs.[2]

3.1 Introduction

Content Security Policy (CSP) is a leading security mechanism for protecting webpages against content injection attacks, primarily against Cross-Site Scripting (XSS) attacks [19, 20]. With the adoption of CSP by major web browsers, website developers are able to deploy CSPs to allow permissible resources (e.g., scripts including JavaScript code or files) to be loaded or executed and behaviors (e.g., form submissions) to occur on their websites. Resources that violate CSPs deployed under the enforcement mode will be blocked, while resources that violate CSPs deployed under the report-only mode will not be blocked but will be reported as CSP violations.

A CSP is composed of directives, and each directive is typically composed of a directive name with a set of directive values. The directive name specifies a certain type of resources to be controlled, while the

[2]This is a joint work with Chuan Yue [5]. Chuan is an associate professor of computer science at the Colorado School of Mines.

directive values specify the allowed resources or behaviors on a webpage. Therefore, different types of resources (e.g., frames, scripts, and fonts) are controlled by the corresponding directives. For example, the *"script-src 'none'"* directive is composed of a *script-src* directive name with a *'none'* directive value. This directive does not allow any scripts to be loaded or executed on a webpage. Developers can deploy CSPs on their webpages either via HTTP(S) response headers or via HTML <meta> tags.

Well-designed CSPs can help protect websites against content injection attacks. Unfortunately, CSPs are not widely deployed on websites yet (e.g., they appeared only on 17.74% websites as reported in a 2023 study [1]), and most deployed CSPs have security issues or errors [2, 8, 22]. Especially, it is often hard to properly deploy CSPs on websites that include third-party scripts due to the need for the cooperation from third parties [3, 4]; the improper blockage of essential third-party scripts often breaks website functionalities, and third-party resource changes also make it difficult to maintain or update CSPs. The usage of third-party scripts can be influenced by web technologies especially ***web builders*** (i.e., content management systems and web frameworks that provide standard and major ways for developers to build web applications) and Content Delivery Networks (***CDNs***). Specifically, some web builders and CDNs host third-party resources on their servers for websites to include and use. Prior studies (e.g., [1, 3, 4, 22, 23]) on CSP deployment issues did not pay sufficient attention to the impacts of different web technologies on the deployment of CSPs.

In this chapter, we focus on analyzing CSP deployment issues related to script resources (in short, scripts) among websites with different web technologies. Specifically, we answer the following three research questions: **RQ1:** Would builder-generated websites encounter the same CSP deployment issues related to third-party scripts as other websites?, **RQ2:** Would CDN-enabled websites encounter the same CSP deployment issues related to third-party scripts as other websites?, and **RQ3:** How can CSPs be improved on different types of ***CSP-deployed websites*** (i.e., websites that already deployed CSPs), and what safe CSPs can be deployed on different types of ***websites without CSPs*** (i.e., websites that did not deploy CSPs yet)? By answering these three research questions, we aim to help developers properly deploy CSPs on their websites.

We implemented a Google Chrome extension to automatically visit the homepages of Tranco [45] top 100K websites. To collect and analyze third-party scripts used on homepages, the constructed extension inserted our specified CSP (i.e., *"default-src 'none'; report-uri https://ourserver/report;"*) under the report-only mode to each homepage, and simulated user interactions. The specified CSP does not allow any resources to be loaded or executed on webpages, and it directs the browser to send CSP violations triggered by the accesses of resources used on webpages to our server. User interactions are simulated as in realistic browsing to trigger event handlers for more resource accesses and CSP violations. Our analysis of the usage of third-party scripts and the CSP deployment issues on websites is based on the CSP violations triggered under the specified CSP; we take this approach because CSP violations directly reflect browsers' policy enforcement and provide a more straightforward way for us to answer our research questions than using HTTP(S) requests.

To identify builder-generated websites and CDN-enabled websites, we used a web technology detection tool which is also a Google Chrome extension. To analyze the impacts of domain changes of third-party scripts on CSP deployment, we conducted website crawling twice. We finally collected 61,827 and 61,510 homepages that triggered CSP violations on third-party scripts in the first crawling and the second crawling, respectively. Meanwhile, 60,299 homepages appeared in both crawlings. Our analysis is mainly based on the 61,827 homepages visited in the first crawling and the 60,299 common homepages visited in both crawlings.

Overall, we found that third-party scripts used on most websites do not contain or load any other scripts which can lead to CSP deployment issues. However, a greater percentage of builder-generated websites and CDN-enabled websites contain third-party scripts which can lead to CSP deployment issues than other websites. Meanwhile, the choice of web builders or CDNs can impact the deployment of CSPs on the corresponding websites. Furthermore, builder-generated CSP-deployed websites and CDN-enabled CSP-deployed websites tend to deploy CSPs that are less safe than other websites. Nonetheless, most CSPs on the CSP-deployed websites can be improved to be safer. For the websites without CSPs, most of them can deploy safe CSPs.

We make three major contributions in this chapter:

- We design a browser extension to directly insert a specified CSP under the report-only mode to each homepage to collect CSP violations triggered by the accesses of resources. This provides a straightforward way for us to analyze the resource usage and CSP deployment issues related to third-party scripts on websites.
- We perform a large-scale measurement study on 100K websites and analyze CSP deployment issues related to third-party scripts among websites with different web technologies to help promote the proper deployment of CSPs.
- We found that a greater percentage of builder-generated websites and CDN-enabled websites use third-party scripts which can lead to CSP deployment issues than other websites. Some web builders and CDNs provide scripts that can lead to CSP deployment issues, indicating that the choice of web builders and CDNs can impact the CSP deployment on the corresponding websites. Furthermore, most CSP-deployed websites can deploy safer CSPs, and most websites without CSPs can deploy safe

CSPs.

The rest of this chapter is organized as follows. Section 3.2 reviews the background of CSP and the related work. Section 3.3 describes the design of our study from the data collection, processing, and analysis perspectives. Section 3.4 analyzes the usage patterns of third-party scripts among builder-generated, CDN-enabled, and other websites and analyzes the corresponding CSP deployment issues. Section 3.5 summarizes the key findings of our work, makes recommendations, and further discusses the limitations of this work as well as potential future work. Section 3.6 concludes this chapter.

3.2 Background and Related Work

In this section, we review the background of CSP and the related work.

3.2.1 Background of CSP

We introduce the CSP directives for controlling scripts which are the focus of our work. We also introduce CSP violations, deployment modes, and ways of deployment.

CSP Directives for Script Control. CSP is able to control the loading and behaviors of inline and external scripts. The "script-src" directives (i.e., directives with the "script-src" name and certain values) and "default-src" directives (which serve as a fallback when more specific fetch directives are missing) are able to control all types of script resources. Developers can also use nonces and hash values as directive values in CSPs to control script resources. In such cases, scripts matched with the specified nonces or hash values are allowed to be loaded or executed. However, CSP also provides the following ***unsafe directive values***: (1) the "'unsafe-inline'" directive value allows any scripts and event handlers included in the first-party websites to be loaded or executed, (2) the "'unsafe-eval'" directive value allows any eval-like functions to be executed, (3) the "'wasm-unsafe-eval'" directive value allows any wasm-eval functions to be executed, (4) the "*" value allows resources from any domains to be loaded or executed, and (5) scheme values (e.g., an "https:" value) allow scripts from any sources with the corresponding scheme values to be executed. If a nonce or hash value and the "'unsafe-inline'" value both appear in a CSP, browsers will ignore the "'unsafe-inline'" value.

Weichselbaum et al. [2] proposed the "'strict-dynamic'" directive value to mitigate the burden of deploying CSPs in chained script inclusion situations by allowing scripts with matched nonce or hash values to further dynamically include scripts to first-party websites. If the "'strict-dynamic'" value and the "'unsafe-inline'" value both appear in a CSP, browsers will ignore the "'unsafe-inline'" value. Additionally, browsers will ignore the whitelisted resource URL directive values if the "'strict-dynamic'" value and the whitelisted resource URLs both appear in a CSP.

CSP Violations. When a resource access (e.g., retrieving a JavaScript file) or behavior (e.g., executing a JavaScript function) violates the CSP deployed on a webpage, it will trigger a CSP violation on a browser. Event handlers that violate CSPs will also trigger CSP violations when they are going to be executed. Developers can specify the hosts or servers to which violation reports will be sent via reporting directives (i.e., with "report-uri" and "report-to" directive names). CSP violation reports contain information of violating resources such as their source files and the violated CSP directives.

CSP Deployment Modes and Ways. CSP has two deployment modes: *enforcement* and *report-only*. In the enforcement mode, CSP violations will let the browsers block violating resources and disable violating behaviors; in the report-only mode, CSP violations will only be reported (typically for web developers to experiment with policies) without the enforcement actions on browsers. Meanwhile, there are two ways to deploy a CSP for a webpage: via an HTTP(S) response header or an HTML <meta> tag. Developers can deploy a CSP in the enforcement mode via a "Content-Security- Policy" HTTP(S) response header or an HTML <meta> tag. However, CSPs in the report-only mode can only be deployed via a "Content-Security-Policy-Report-Only" HTTP(S) response header. In addition, CSPs deployed via HTML <meta> tags are not applied to the content that precedes them [19].

3.2.2 Related Work

Although CSP is proposed for protecting webpages against content injection attacks, it is not widely deployed in practice [1, 2, 21, 22], and most deployed CSPs have security issues or deployment errors [1, 2, 8, 22, 23]. Weichselbaum et al. found that 94.72% of deployed policies are ineffective to protect websites against XSS attacks due to policy misconfigurations and insecure whitelist entries [2]. Calzavara et al. found that above 90% of CSP-deployed websites were vulnerable to XSS attacks because of using an "'unsafe-eval'" or "'unsafe-inline'" directive value [8]. They further proposed Compositional CSP to incrementally compose a needed policy based on the runtime interactions on a webpage [24].

Roth et al. conducted a longitudinal study to analyze the historical evolution of CSP deployment [22]. They 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, and opined that event handlers hinder websites from deploying CSPs without using an "'unsafe-inline'" value. Ren et al. took the unsupervised clustering approach to analyze the security levels of the deployed CSPs from the directive coverage and secure use perspectives, and found that most deployed CSPs do not sufficiently protect webpages from these two perspectives [1].

Several studies explored the root causes of CSP deployment issues. Steffens et al. found that most first-party websites need the cooperation with third parties to properly deploy CSPs [4]. Roth et al.

conducted a qualitative semi-structured interview study with 12 web developers to uncover the root causes of insecure CSP deployment, and found that legacy code, third-party code, and the complexity of the CSP mechanism, etc. could all block a successful CSP deployment [3].

However, all above studies did not categorize the websites with different web technologies and did not study the impacts of web technologies on the resource usage and thus the proper deployment of CSPs. In our work, we analyze the usage patterns of third-party scripts and the corresponding CSP deployment issues among builder-generated and CDN-enabled websites to help improve their CSP deployment instead of analyzing all websites indiscriminately.

To promote CSP adoption, some researchers focused on automatically generating CSPs. Fazzini et al. designed AutoCSP to identify scripts that should be whitelisted, and to automatically generate CSPs for webpages by modifying the server-side code [25]. Pan et al. designed CSPAutoGen to automatically generate CSPs for webpages by importing inline and dynamic scripts as external scripts[26]. Our work and the findings could benefit future CSP auto-generation studies by providing a good understanding of the impacts of web technologies on the resource usage and thus the proper deployment of CSPs.

3.3 Methodology

In this section, we describe the design of our study from the data collection, processing, and analysis perspectives. Recall that our unique goal in this study is to analyze the CSP deployment issues related to third-party script resources among websites with different web technologies. We construct a Google Chrome extension to automatically crawl the homepages of websites, directly insert a specified CSP under the report-only mode to each homepage, and collect the CSP violations triggered by the accesses of the resources. The analysis of the resource usage and the CSP deployment issues on websites is based on the CSP violations triggered under the specified CSP; we take this approach because CSP violations directly reflect browsers' policy enforcement and provide a more straightforward way for us to answer our research questions than using HTTP(S) requests. We focus on crawling the homepage of each website because a homepage is often the initial point of access for users, and it should be well protected against attacks.

3.3.1 Website Crawling and Data Collection Framework

Figure 4.1 illustrates our website crawling and data collection framework. Our constructed browser extension consists of four components: a *CSP inserter*, a *user interaction simulator*, a *request and response collector*, and an *HTML document collector*. It instructs the browser to visit the homepage of each website. It collects the original CSPs deployed on homepages (if they exist), HTTP(S) requests and responses, and HTML documents of homepages.

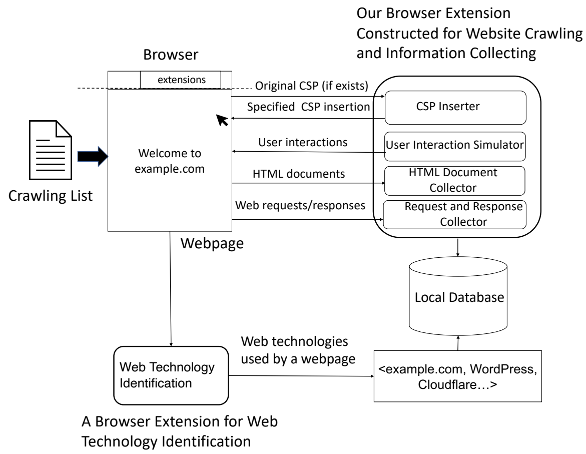


Figure 3.1 Website Crawling and Data Collection Framework

To collect resources used on webpages, the CSP inserter component inserts a specified CSP (i.e., “default-src ‘none’; report-uri https://ourserver/report;”) under the report-only mode via an HTTP(S) response header for a webpage. Specifically, the CSP inserter first checks whether there is a deployed CSP on a homepage. If there is no CSP deployed on the homepage, the CSP inserter directly inserts the specified CSP for the homepage. If there is a CSP deployed on the homepage, the CSP inserter removes the original CSP after saving it and then inserts the specified CSP for the homepage. The specified CSP aims to not allow any resources to be loaded or executed on homepages. However, because this CSP is inserted under the report-only mode, it will not block any violating resources but will only generate CSP violations for these resources. Therefore, the accesses of all resources used on a webpage will trigger CSP violations. Because of the specification of the “report-uri” directive, the CSP violations will be wrapped as HTTPS requests and sent to our server.

Each triggered CSP violation contains information about the violating resource, allowing us to directly analyze the resource that can be controlled by CSP. To trigger more event handlers and achieve a more comprehensive collection of resources, the user interaction simulator simulates two common user interactions after each webpage is fully loaded: scrolling down the webpage to the bottom and clicking on the first identified button (which can be a submit, reset, checkbox, or radio button) on the webpage. The HTML document collector saves the loaded HTML document when a webpage is fully loaded. The request and response collector intercepts all HTTP(S) requests and the corresponding responses generated for a webpage by using the *chrome.debugger* extension API and the Network Domain of the Chrome DevTools Protocol [46]. All collected data will be saved in a local database.

CSP violations triggered under our specified CSP will be used for the analysis of usage patterns of third-party scripts and the related CSP deployment issues among builder-generated websites, CDN-enabled websites, and other websites (Sections 3.4.1, 3.4.2, and 3.4.3, respectively). Note that another Google Chrome extension Wappalyzer [47] (Section 3.3.2) is used to identify builder-generated websites and CDN-enabled websites based on web technology fingerprints. The collected HTTP(S) requests and responses as well as HTML documents will be used for further validating builder-generated websites and CDN-enabled websites (Section 3.3.3). Also note that our extension does not check whether there is a CSP deployed via an HTML <meta> tag in a webpage because in pratice only a very small number of websites deployed a CSP via an HTML <meta> tag [22].

3.3.2 Data Collection Process and Dataset

We visited homepages of the Tranco [45] top 100K (dated February 8th 2023) websites in February and March of 2023 by utilizing six computers with identical configurations. Each computer is installed with a Google Chrome browser (Version 104) which is only configured with our constructed browser extension (Section 3.3.1) and a popular web technology detection browser extension, Wappalyzer (Version 6.10.54) [47], on a Ubuntu 20.04 LTS system. To analyze the impacts of domain changes for third-party scripts on CSP deployment, we performed the website crawling twice. The second crawling of a website started five days after the end of its first crawling. In total, homepages of 65,595 unique websites were successfully visited, and 13,557 homepages deployed CSPs based on the first crawling. Meanwhile, 63,352 homepages contain resources that trigger CSP violations under the inserted CSP, while 2,243 homepages are not with any CSP violations (e.g., blank webpages). In the second crawling, homepages of 65,197 unique websites were successfully visited in total, and 13,369 homepages deployed CSPs. Meanwhile, 63,034 homepages contain resources that trigger CSP violations under the inserted CSP, while 2,163 homepages are not with any CSP violations.

Our focus is on analyzing homepages with CSP violations of *script resources*. Therefore, in the first crawling, we further filtered out 1,525 out of 63,352 homepages without any CSP violations of script resources, keeping the remaining ***61,827*** homepages (among which 13,497 deployed CSPs). In the second crawling, we further filtered out 1,524 out of 63,034 homepages without any CSP violations of script resources, keeping the remaining 61,510 homepages (among which 13,253 deployed CSPs). In total, ***60,299*** homepages that triggered CSP violations of script resources were successfully accessed in both crawlings. Our analysis in Section 3.4 is mainly based on the 61,827 homepages visited in the first crawling and the 60,299 common homepages visited in both crawlings.

3.3.3 Data Processing and Analysis

To support our data analysis, we process our dataset by identifying if a homepage is builder-generated or not, and if it is CDN-enabled or not; meanwhile, we differentiate third parties from first parties, and categorize third-party scripts related to CSP deployment issues into 10 types.

3.3.3.1 Webpage Type Identification

Builder-generated Webpages. We consider Content Management Systems (CMSs) and web frameworks as web builders because both of them provide standard and major ways for developers to build web applications. The Wappalyzer web technology detection browser extension [47] is able to detect 126 CMSs and 41 web frameworks used for generating 22,105 homepages based on the first crawling. To estimate the accuracy of the detection of builder-generated homepages, we manually identified the ground truth of a set of 1,000 randomly selected homepages by analyzing their HTML documents as well as the corresponding HTTP(S) requests and responses. We found that among these 1,000 homepages, 315 are builder-generated homepages, while 685 are other homepages. Wappalyzer correctly identifies 226 builder-generated homepages and 649 other homepages. We analyzed all 125 false identifications including 36 false positives and 89 false negatives. Wappalyzer achieves 87.50% on accuracy, 86.25% on precision, 71.25% on recall, and 78.04% on F1-score in identifying builder-generated webpages. Note that a homepage can be built with multiple web builders in rare situations; we found that only 4.70% of 22,105 homepages were built with multiple CMSs, multiple frameworks, or mixed CMSs and frameworks.

CDN-enabled Webpages. Wappalyzer is able to detect 42 CDNs used for delivering 27,332 homepages based on the first crawling. Similarly to estimate the accuracy of the detection of CDN-enabled homepages, we manually identified the ground truth of the same set of 1,000 randomly selected homepages by analyzing their HTML documents as well as the corresponding HTTP(S) requests and responses. We found that 408 are CDN-enabled homepages, while 592 are other homepages. Wappalyzer correctly identifies 377 CDN-enabled homepages and 445 other homepages. We analyzed all 178 false identifications including 147 false positives and 31 false negatives. Wappalyzer achieves 82.20% on accuracy, 71.95% on precision, 92.40% on recall, and 80.90% on F1-score in identifying CDN-enabled webpages. Note that the resources on a homepage can be delivered via multiple CDNs; we found such cases on 21.23% of those 27,332 homepages.

3.3.3.2 Third-party Script Type Identification

As in studies such as [4, 10], we use the notion of an eTLD+1 (effective top-level domain+1) to differentiate third parties from first parties. If the eTLD+1 of a script resource is different from the eTLD+1 of a crawled homepage, we consider this script resource as a third-party script on the homepage. To differentiate third-party scripts more accurately by considering that a different eTLD+1 of a script may actually belong to the same party or organization as that of a homepage, we refined our results based on the domain relationships provided by the webXray domain owner list [48] and the domain owner list derived by Steffens et al. in [4]. Note that webXray was also used by Steffens et al. in [4].

As introduced in Section 4.2.1, using "'unsafe-inline'", "'unsafe-eval'", "*", or scheme directive values will undermine the protection provided by a CSP. On the other hand, improper blockages of third-party scripts can often break webpage functionalities. There are three major CSP deployment issues related to third-party scripts: (1) insertions of scripts and event handlers into first-party websites by third-party scripts can lead to the usage of an "'unsafe-inline'" directive value in a CSP; (2) eval-like functions contained in third-party scripts can lead to the usage of an "'unsafe-eval'" directive value in a CSP, and wasm-eval functions contained in third-party scripts can lead to the usage of a "'wasm-unsafe-eval'" or an "'unsafe-eval'" directive value in a CSP; (3) a large number of third-party script source domains and frequent source domain changes on a webpage can lead to the usage of a "*" directive value or a scheme directive value in a CSP.

First-party webpages can specify source URLs to directly include third-party scripts (referred to as ***direct third-party scripts***), which can further include other third-party scripts (referred to as ***indirect third-party scripts***) thus resulting in chained script inclusion situations. Either direct or indirect third-party scripts may even further insert new scripts into the first-party webpages, e.g., using the *document.createElement()* function; in other words, beyond providing their functions for first-party webpages to call, they may dynamically modify the Document Object Model (DOM) of first-party webpages with the added new script elements. To avoid the aforementioned CSP deployment issues, both direct and indirect third-party scripts included on a webpage should not require the webpage to use any of those unsafe directive values (e.g., an "'unsafe-inline'" value) in its CSP.

As shown in Table 3.1, we categorize third-party script resources related to CSP deployment issues into 10 types according to the CSP Level-3 specification [19] and the analyses in [2, 4, 10]. This categorization uses the information in CSP violations of webpages triggered under our inserted CSP. The "blocked-uri" and "violated-directive" fields of CSP violations generated by Google Chrome indicate whether the reported script resources are source URLs, event handlers, embedded scripts, or eval-like functions. Therefore, the sources of scripts can be correctly tracked for accurate resource type categorization. Categorizing third-party scripts into these 10 types will allow us to comprehensively analyze their specific impacts on the CSP deployment in Section 3.4.

Table 3.1 10 Types of Third-party Script Resources Related to CSP Deployment Issues. The word "using" in the third column means that web developers may need to use certain unsafe directive values (e.g., "'unsafe-inline'", " 'unsafe-eval'", or "*") in their CSPs.

Type No.	***Third-party Script Resource Type***	***Related CSP Deployment Issues***
I	Source URLs of Direct Third-party Scripts (Included into First-party Websites)	Using a "*" Value or Scheme Values
II	Event Handlers Inserted into First-party Websites by Direct Third-party Scripts	Using an "'unsafe-inline'" Value
III	Scripts Inserted into First-party Websites by Direct Third-party Scripts	Using an "'unsafe-inline'" Value
IV	Eval-like Functions Induced by Direct Third-party Scripts	Using an "'unsafe-eval'" Value
V	Wasm Eval Functions Induced by Direct Third-party Scripts	Using an "'unsafe-eval'" or a "'wasm-unsafe-eval'" Value
VI	Source URLs of Indirect Third-party Scripts (Included into First-party Websites)	Using a "*" Value or Scheme Values
VII	Event Handlers Inserted into First-party Websites by Indirect Third-party Scripts	Using an "'unsafe-inline'" Value
VIII	Scripts Inserted into First-party Websites by Indirect Third-party Scripts	Using an "'unsafe-inline'" Value
IX	Eval-like Functions Induced by Indirect Third-party Scripts	Using an "'unsafe-eval'" Value
X	Wasm Eval Functions Induced by Indirect Third-party Scripts	Using an "'unsafe-eval'" or a "'wasm-unsafe-eval'" Value

3.3.3.3 Data Analysis Techniques

In Section 3.4, we will analyze the usage (including popularity and distributions) of the 10 types of third-party scripts and the CSP deployment issues among different types of websites. We will report the distributions of the number of third-party domains changed for different types of scripts among different types of websites to explore the impacts of domain changes on CSP deployment. We will identify CSP-deployed websites that can further improve their CSPs, and websites without CSPs that can deploy safe CSPs.

We will use the Chi-square test of independence to inspect the relationship between two categorical variables, and will use the Wilcoxon rank sum test for two groups to perform the between-subjects comparison of data distributions. Specifically, we will use the Chi-square test of independence for inspecting the relationships between different types of websites and the usage of different types of third-party scripts. We will reject the null hypothesis (which states that the popularity of using a specific type of third-party scripts is the same across different types of websites) when *p-value* < 0.05. We will also use the Chi-square test of independence to inspect the relationships between different types of CSP-deployed websites and the adoption of safe CSPs. We will reject the null hypothesis (which states that the adoption of safe CSPs is the same across different types of CSP-deployed websites) when *p-value* < 0.05. We will use the Wilcoxon rank sum test to perform the comparison of the distributions of the number of domains of third-party scripts among different types of websites. We will reject the null hypothesis (which states that the corresponding distributions are the same) when p-value < 0.05.

3.3.4 Ethical Considerations and Reproducibility

Our data collection process (Section 3.3.2) does not raise obvious ethical issues. It does not involve human subjects or potentially sensitive data (e.g., user behavior or social network information, etc.). Our constructed Google Chrome extension only visited the public homepages of websites. On each homepage, our browser extension inserts the specified CSP, scrolls down to the bottom, only clicks on one button to collect HTTP(S) requests and responses as well as CSP violations triggered under our inserted CSP, and saves the loaded HTML document (Section 3.3.1). Therefore, there is minimal or even no change to the click-through rate of ads, and the traffic burden incurred to each visited website is also minimal. In web measurement studies, reproducibility is a requirement that is crucial to sustainable research [49–51]. Our work is reproducible because we provide the details of our website crawling and data collection framework, our data collection process and dataset, and our data processing and analysis techniques in this section as suggested in studies such as [49–51]; meanwhile, our constructed Google Chrome extension and dataset will be open-sourced along with the publication of this chapter.

3.4 Results and Analysis

In this section, we answer the three research questions about the usage of third-party script resources and the related CSP deployment issues among builder-generated, CDN-enabled, and other websites.

3.4.1 Third-Party Script Resources in Builder-generated and Other Websites

To answer RQ1, we explored the usage patterns of third-party script resources and the related CSP deployment issues between builder-generated and other websites from multiple perspectives.

3.4.1.1 Popularity of the 10 Types of Third-Party Scripts

Figure 3.2 illustrates the popularity of the 10 types of third-party script resources related to CSP deployment issues (Section 3.3.3.2) on the two types of websites. We can see that while the distributions of the 10 types of third-party script resources are similar between builder-generated websites and other websites, a greater percentage of builder-generated websites contain third-party scripts related to CSP deployment issues than other websites.

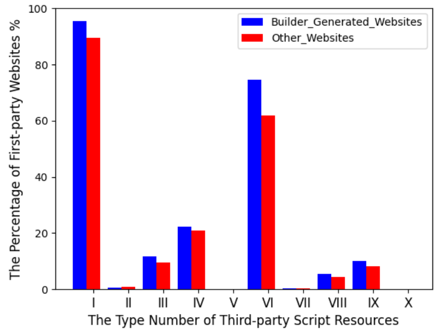


Figure 3.2 The Popularity of the 10 Types of Third-party Script Resources between Builder-generated Websites (n=22,105) and Other Websites (n=39,722)

The two most popular types of third-party script resources used on builder-generated and other websites are source URLs of direct third-party scripts (95.41% and 89.37%, respectively) and source URLs of indirect third-party scripts (74.63% and 61.95%, respectively). This indicates that most websites include third-party scripts both directly and indirectly. Web developers should carefully inspect all source URLs of third-party scripts used on their webpages, and properly design CSPs to only include scripts from trustworthy sources. The third most popular type of third-party script resources used on builder-generated and other websites are eval-like functions induced by direct third-party scripts (22.26% and 20.83%, respectively). Additionally, 10.00% of builder-generated websites and 8.18% of other websites contain eval-like functions induced by indirect third-party scripts. Eval-like functions induced by third-party scripts can make developers deploy CSPs with an "'unsafe-eval'" directive value which allows any eval-like functions to be executed on first-party websites (i.e., those websites among the top 100K that were successfully visited in our crawling). Luckily based on our results, most websites would not encounter and could avoid the CSP deployment issue of using an "'unsafe-eval'" directive value.

The four least popular types of third-party script resources used on builder-generated and other websites are: (1) event handlers inserted into first-party websites by direct third-party scripts, (2) event handlers inserted into first-party websites by indirect third-party scripts, (3) wasm eval functions induced by direct third-party scripts, and (4) wasm eval functions induced by indirect third-party scripts. The popularity of each of these four types of third-party resources is less than 1%. Prior studies opined that event handlers were one major reason for websites to deploy CSPs with an "'unsafe-inline'" directive value [4, 22]. However, here we can see that most first-party websites at least do not contain event handlers inserted into them by third-party scripts.

Scripts inserted into first-party websites by third-party scripts can also make developers deploy CSPs with an "'unsafe-inline'" directive value. We found that 11.80% of builder-generated websites and 9.57% of other websites contain scripts inserted into the first-party websites by direct third-party scripts.

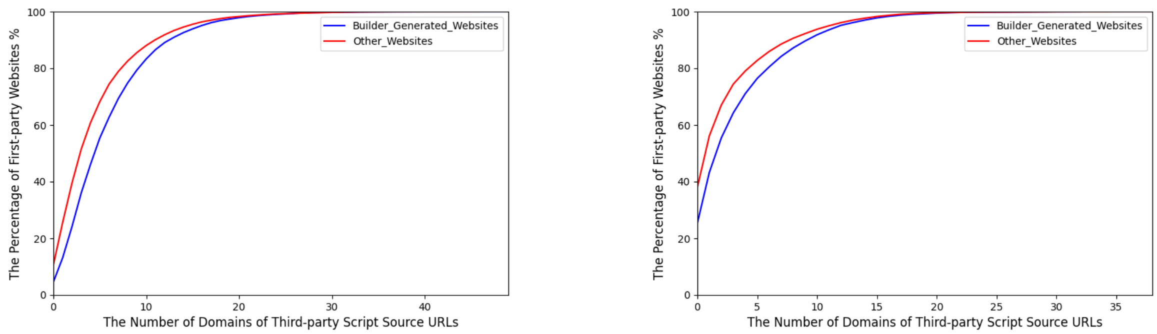
Meanwhile, 5.54% of builder-generated websites and 4.48% of other websites contain scripts inserted into the first-party websites by indirect third-party scripts. We can see that the insertion of scripts into first-party websites by direct third-party scripts occurs more often than that by indirect third-party scripts.

We can see clearly that six out of 10 types of third-party script resources are more popularly used on builder-generated websites than on other websites: (1) source URLs of direct third-party scripts, (2) source URLs of indirect third-party scripts, (3) scripts inserted into first-party websites by direct third-party scripts, (4) scripts inserted into first-party websites by indirect third-party scripts, (5) eval-like functions induced by direct third-party scripts, and (6) eval-like functions induced by indirect third-party scripts. The Chi-square test of independence found that the differences between builder-generated websites and other websites in the popularity of using each of these six types of third-party script resources are all statistically significant. These results indicate that builder-generated websites are more likely to use third-party scripts, thus would encounter more CSP deployment issues than other websites.

3.4.1.2 Domains of the Included Third-Party Scripts

To figure out whether builder-generated and other websites would encounter the CSP deployment issue of using a "*" directive value or a scheme directive value, we further analyzed the number of domains of third-party script source URLs. Figure 3.3 shows the cumulative distribution function (CDF) of first-party builder-generated websites and other websites regarding the number of domains of the included third-party scripts. As shown in Figure 3.3(a), most first-party websites include direct third-party scripts from no more than 10 third-party domains, indicating that they could easily avoid using a "*" directive value or a scheme directive value if they would whitelist the domains of direct third-party scripts. In more details, for builder-generated websites, 4.59% of them do not include direct third-party scripts; 83.30% of them include direct third-party scripts from no more than 10 third-party domains; 97.86% of them include direct third-party scripts from no more than 20 third-party domains. For other websites, 10.62% of them do not include direct third-party scripts; 88.04% of them include direct third-party scripts from no more than 10 third-party domains; 98.26% of them include direct third-party scripts from no more than 20 third-party domains. We can see that builder-generated websites are more likely to include direct third-party scripts from more domains than other websites. The Wilcoxon rank sum test found that this difference between these two types of websites is statistically significant.

Compared to the number of websites that include direct third-party scripts, fewer websites include indirect third-party scripts as shown in Figure 3.3(b). Specifically, for builder-generated websites, 25.37% of them do not include indirect third-party scripts; 91.83% of them include indirect third-party scripts from no more than 10 third-party domains; 99.48% of them include indirect third-party scripts from no more



(a) Direct Third-party Script Source URLs

(b) Indirect Third-party Script Source URLs

Figure 3.3 CDF Distributions of First-party Builder-generated Websites (n=22,105) and Other Websites (n=39,722) regarding the Number of Domains of the Included Third-party Scripts

than 20 third-party domains. For other websites, 38.04% of them do not include indirect third-party scripts; 93.79% of them include indirect third-party scripts from no more than 10 third-party domains; 99.61% of them include indirect third-party scripts from no more than 20 third-party domains. Overall, most first-party websites include indirect third-party scripts from no more than 10 third-party domains, indicating that they could also easily avoid using a "*" directive value or a scheme directive value if they would whitelist the domains of indirect third-party scripts. We can also see that builder-generated websites are more likely to include indirect third-party scripts from more domains than other websites. The Wilcoxon rank sum test found that this difference between these two types of websites is also statistically significant.

3.4.1.3 Websites Built on Different Web Builders

We further explored the tendencies of encountering different CSP deployment issues among websites built on different web builders. We intentionally define three types of tendencies of CSP deployment issues: (1) if at least 50% (i.e., majority) of websites built on a web builder trigger CSP violations of scripts and event handlers inserted into first-party websites by third-party scripts, we consider those websites having a tendency to deploy CSPs with an "'unsafe-inline'" directive value; (2) if at least 50% of websites built on a web builder trigger CSP violations of eval-like functions which include wasm eval functions induced by third-party scripts, we consider those websites having a tendency to deploy CSPs with an "'unsafe-eval'" directive value; (3) if at least 50% of websites built on a web builder load third-party scripts from more than 20 domains, we consider those websites having a tendency to deploy CSPs with a "*" directive value or a scheme directive value.

We analyzed these three types of tendencies of CSP deployment issues among websites built on 64 web builders (shown in Table B.1 in Appendix B.1) that have a minimum of 10 websites built on each of them. We found four web builders with websites built on them having a tendency to deploy CSPs with an "'unsafe-inline'" directive value, eight web builders with websites built on them having a tendency to deploy CSPs with an "'unsafe-eval'" directive value, and no web builders with websites built on them having a tendency to deploy CSPs with either a "*" or a scheme directive value. Some web builders host and provide third-party scripts for websites to use, which may impact the CSP deployment. Specifically, 2,552 out of 22,105 builder-generated websites include third-party scripts from the domains of web builders they used. Twelve web builders (shown in Table B.2 in Appendix B.2) have a minimum of 10 websites using them, and they supply third-party scripts to more than 50% of the websites built on them. Among these 12 web builders, three and seven provide scripts that can lead to the usage of an "'unsafe-inline'" and an "'unsafe-eval'" directive value, respectively.

3.4.1.4 Third-party Scripts' Domain Changes

We further analyzed the impacts of domain (i.e., eTLD+1) changes for third-party scripts (between the two crawlings) on CSP deployment among the 19,090 builder-generated websites and the 36,126 other websites that are commonly visited in both crawlings. Figure 3.4 shows the popularity of domain changes for the 10 types of third-party script resources. Overall, 13.72% of builder-generated websites and 11.78% of other websites have source domain changes for direct third-party scripts, while 13.35% of builder-generated websites and 11.82% of other websites have source domain changes for indirect third-party scripts. Most websites do not have source domain changes for third-party scripts, indicating that they would not encounter severe CSP deployment issues of using a "*" directive value or a scheme directive value simply due to source domains' dynamic changes.

Meanwhile, we found that source domain changes for most types of third-party script resources are more popular on builder-generated websites than on other websites. The Chi-square test of independence found that such differences for three types of third-party script resources are statistically significant: (1) source URLs of direct third-party scripts, (2) source URLs of indirect third-party scripts, and (3) scripts inserted into first-party websites by indirect third-party scripts. Furthermore, two web builders (i.e., TownNews and Adobe Experience Manager) with more than 50% of websites built on them have domain changes for third-party scripts.

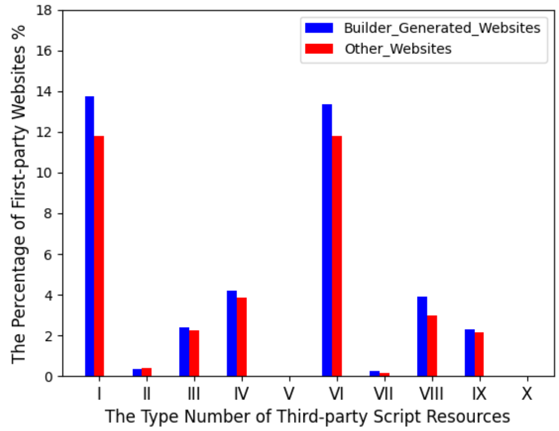


Figure 3.4 The Popularity of the 10 Types of Third-party Script Resources that Change Domains between the Two Crawdlings among Builder-generated Websites (n=19,090) and Other Websites (n=36,126)

3.4.1.5 CSP Support in Web Builders

Support for CSPs in web frameworks can also impact CSP deployment as discussed in [3]. Therefore, we further analyzed the CSP support of web builders. Web builders should support websites to deploy CSPs through HTTP(S) response headers because of the three limitations of deploying CSPs through <meta> tags. First, websites cannot deploy CSPs under the report-only mode for the testing purpose through <meta> tags. Second, resources fetched or prefetched through links or script elements that precede the <meta> tags for deploying CSPs will not be controlled. Last, "report-uri" directives are not supported to send CSP violations to web developers in CSPs deployed via <meta> tags.

Overall, 22,105 builder-generated websites are built on 126 CMSs and 41 web frameworks. To figure out the CSP support of these web builders, we analyzed their official online documents. We found that 77 out of the 126 CMSs support websites to deploy CSPs through HTTP(S) response headers, one CMS supports websites to deploy CSPs via <meta> tags, and 48 CMSs do not provide any information about the CSP deployment support. Among the 41 web frameworks, 31 of them support websites to deploy CSPs via HTTP(s) response headers, two of them only support websites to deploy CSPs through <meta> tags, and eight of them do not provide any information about the CSP deployment support.

Takeaways. Overall, most websites do not contain third-party scripts that can lead to CSP deployment issues. Notably, builder-generated websites are more likely to include third-party scripts and load them from more domains compared to other websites. Meanwhile, builder-generated websites are more likely to have source domain changes for third-party scripts. We also observed that some web builders host their own script resources that can lead to CSP deployment issues for websites, and some web builders do not (or do not adequately) provide CSP deployment support to websites; these results indicate that the choice of web builders can impact the CSP deployment on the corresponding websites.

3.4.2 Third-Party Script Resources in CDN-enabled Websites and Other Websites

To answer RQ2, we explored the usage patterns of third-party script resources and the related CSP deployment issues between CDN-enabled and other websites from multiple perspectives.

3.4.2.1 Popularity of the 10 Types of Third-Party Scripts

Figure 3.5 illustrates the popularity of the 10 types of third-party script resources related to CSP deployment issues (Section 3.3.3.2) on the two types of websites. Similar to that between builder-generated websites and other websites (Section 3.4.1.1), we can see that while the distributions of the 10 types of third-party script resources are similar between CDN-enabled websites and other websites, a greater percentage of CDN-enabled websites contain third-party scripts related to CSP deployment issues than other websites.

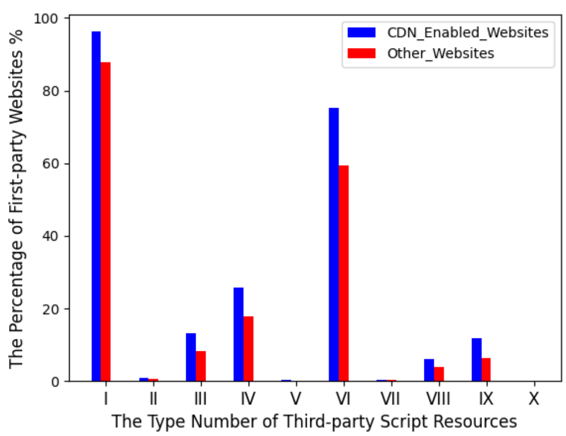


Figure 3.5 The Popularity of the 10 Types of Third-party Script Resources between CDN-enabled Websites (n=27,332) and Other Websites (n=34,495)

The three most popular types of third-party script resources used on CDN-enabled and other websites are: (1) source URLs of direct third-party scripts (96.23% and 87.81%, respectively), (2) source URLs of indirect third-party scripts (75.31% and 59.50%, respectively), and (3) eval-like functions induced by direct third-party scripts (25.85% and 17.77%, respectively). Additionally, 11.81% of CDN-enabled websites and 6.47% of other websites contain eval-like functions induced by indirect third-party scripts. Most CDN-enabled and other websites do not contain eval-like functions induced by third-party scripts, indicating that they would not encounter the related CSP deployment issue of using an "'unsafe-eval'" directive value.

The four least popular types of third-party script resources used on CDN-enabled and other websites are: (1) event handlers inserted into first-party websites by direct third-party scripts, (2) event handlers inserted into first-party websites by indirect third-party scripts, (3) wasm eval functions induced by direct third-party scripts, and (4) wasm eval functions induced by indirect third-party scripts. The popularity of each of these four types of third-party resources is less than 1%. We can see that similar to what we found in Section 3.4.1.1, most CDN-enabled and other websites at least do not contain event handlers inserted into them by third-party scripts, indicating that they would not encounter the related CSP deployment issue of using an "'unsafe-inline'" directive value.

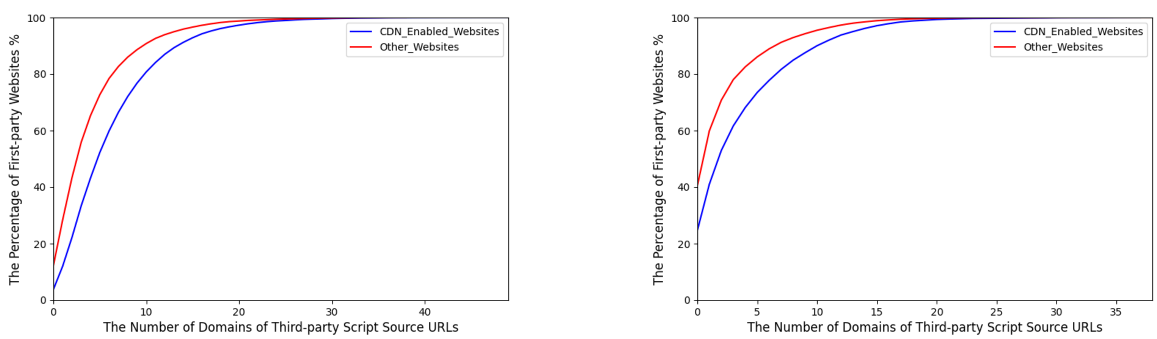
We found that 13.12% of CDN-enabled websites and 8.19% of other websites contain scripts inserted into first-party websites by direct third-party scripts. Meanwhile 6.13% of CDN-enabled websites and 3.86% of other websites contain scripts inserted into first-party websites by indirect third-party scripts. Most CDN-enabled and other websites do not contain scripts inserted into them by third-party scripts, indicating that they would not encounter the related CSP deployment issue of using an "'unsafe-inline'" directive value. In addition, similar to that in Section 3.4.1.1, the insertion of scripts into first-party websites by direct third-party scripts occurs more often than that by indirect third-party scripts.

We can see clearly that seven out of 10 types of third-party script resources are more popularly used on CDN-enabled websites than on other websites: (1) source URLs of direct third-party scripts, (2) scripts inserted into first-party websites by direct third-party scripts, (3) eval-like functions induced by direct third-party scripts, (4) wasm eval functions induced by direct third-party scripts, (5) source URLs of indirect third-party scripts, (6) scripts inserted into first-party websites by indirect third-party scripts, and (7) eval-like functions induced by indirect third-party scripts. The Chi-square test of independence found that the differences between CDN-enabled websites and other websites in the popularity of using each of these seven types of third-party script resources are all statistically significant. These results indicate that CDN-enabled websites are more likely to use third-party scripts, thus would encounter more CSP deployment issues than other websites.

3.4.2.2 Domains of the Included Third-Party Scripts

To figure out whether CDN-enabled and other websites would encounter the CSP deployment issues of using a "*" directive value or a scheme directive value, we further analyzed the number of domains of third-party script source URLs. Figure 3.6 shows the CDF distributions of first-party CDN-enabled websites and other websites regarding the number of domains of the included third-party scripts. Overall, similar to what we found in Section 3.4.1.2, most websites include direct or indirect third-party scripts from no more than 10 domains, indicating that they could easily avoid using a "*" directive value or a scheme directive value if they would whitelist the domains of third-party scripts.

In more details, as shown in Figure 3.6(a), 3.78% of CDN-enabled websites and 12.19% of other websites do not include direct third-party scripts; 80.74% of CDN-enabled websites and 90.77% of other websites include direct third-party scripts from no more than 10 domains; 97.31% of CDN-enabled websites and 98.76% of other websites include direct third-party scripts from no more than 20 domains. As shown in Figure 3.6(b), 24.69% of CDN-enabled websites and 40.50% of other websites do not include indirect third-party scripts; 90.03% of CDN-enabled websites and 95.51% of other websites include indirect third-party scripts from no more than 10 domains; 99.32% of CDN-enabled websites and 99.76% of other websites include indirect third-party scripts from no more than 20 domains. CDN-enabled websites are more likely to include both direct and indirect third-party scripts from more domains than other websites. The Wilcoxon rank sum tests found that these differences between CDN-enabled and other websites are all statistically significant.



(a) Direct Third-party Script Source URLs (b) Indirect Third-party Script Source URLs

Figure 3.6 CDF Distributions of First-party CDN-enabled Websites (n=27,332) and Other Websites (n=34,495) regarding the Number of Domains of the Included Third-party Scripts

3.4.2.3 Websites Enabled by Different CDNs

We further explored the three types of tendencies of CSP deployment issues (similar to what we intentionally defined in Section 3.4.1.3 but in terms of CDN-enabled websites) among websites using different CDNs. Among 26 CDNs (shown in Table B.3 in Appendix B.3) that have a minimum of 10 websites using them, we found two CDNs (i.e., Akamai and Edgio) with websites using them having a tendency to deploy CSPs with an "'unsafe-eval'" directive value, and no CDNs with websites using them having a tendency to deploy CSPs with either an "'unsafe-inline'", a "*", or a scheme directive value. Some CDNs host and provide third-party scripts for websites to use, which may impact the CSP deployment. Specifically, 12,341 out of 27,332 CDN-enabled websites include third-party scripts from the domains of CDNs they used. Eight out of the 26 CDNs provide third-party scripts to more than 50% of the websites enabled by them as indicated in Table B.3 in Appendix B.3. Among these eight CDNs, six and four provide scripts that can lead to the usage of an "'unsafe-eval'" and an "'unsafe-inline'" directive value, respectively.

3.4.2.4 Third-party Scripts' Domain Changes

We further analyzed the impacts of domain (i.e., eTLD+1) changes for third-party scripts (between the two crawlings) on CSP deployment among the 22,110 CDN-enabled websites and the 30,737 other websites that are commonly visited in both crawlings. Figure 3.7 shows the popularity of domain changes for the 10 types of third-party script resources. Overall, 14.94% of CDN-enabled websites and 9.60% of other websites have source domain changes for direct third-party scripts, while 14.88% of CDN-enabled websites and 9.21% of other websites have source domain changes for indirect third-party scripts. Most websites do not have source domain changes for third-party scripts, indicating that they would not encounter severe CSP deployment issues of using a "*" directive value or a scheme directive value simply due to source domains' dynamic changes.

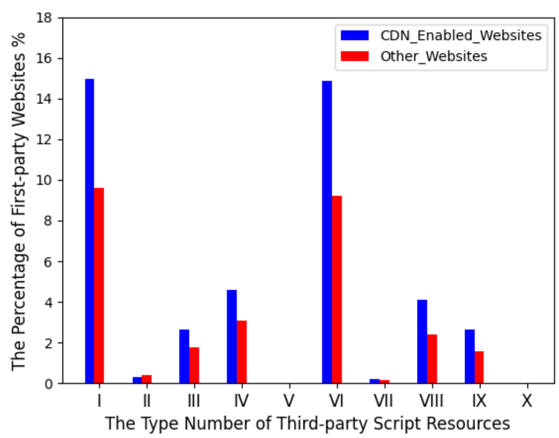


Figure 3.7 The Popularity of the 10 Types of Third-party Script Resources that Change Domains between the Two Crawlings among CDN-enabled Websites (n=22,110) and Other Websites (n=30,737)

Meanwhile, we found that source domain changes for most types of third-party script resources are more popular on CDN-enabled websites than on other websites. The Chi-square test of independence found that such differences for six types of third-party script resources are statistically significant: (1) source URLs of direct third- party scripts, (2) scripts inserted into first-party websites by direct third-party scripts, (3) eval-like functions induced by direct third-party scripts, (4) source URLs of indirect third-party scripts, (5) scripts inserted into first-party websites by indirect third-party scripts, and (6) eval-like functions induced by indirect third-party scripts. Additionally, no CDN with more than 50% of websites using them has domain changes for third-party scripts.

Takeaways. Overall, most websites do not contain third-party scripts that can lead to CSP deployment issues. Similar to builder-generated websites, CDN-enabled websites are more likely to include third-party scripts and load them from more domains compared to other websites. Meanwhile, CDN-enabled websites are more likely to have domain changes for third-party scripts. We also observed that some CDNs host their own script resources that can lead to CSP deployment issues for websites, indicating that the choice of CDNs can impact the CSP deployment on the corresponding websites.

3.4.3 Potential CSP Improvement or Deployment on Different Types of Websites

To answer RQ3, we analyzed the potential improvement for having safer CSPs on different types of CSP-deployed websites, and the potential deployment of safe CSPs on different types of websites without CSPs. If a CSP satisfies the following four requirements, we consider it as a ***safe CSP***. The first three requirements for a safe CSP are from [10]: (1) the CSP contains a directive that controls the scripts used on a webpage (e.g., containing a "script-src" or a "default-src" directive), (2) the CSP does not contain an "'unsafe-inline'" directive value unless a nonce or hash value also appears in it (with the exception that a nonce or hash value also appears in the directive because browsers will then ignore "'unsafe-inline'" in such cases as introduced in Section 3.2.1), and (3) the CSP does not contain a "*" or a scheme directive value unless a "'strict-dynamic'" directive value also appears in it. The fourth requirement we added for a safe CSP is that the CSP does not contain an "'unsafe-eval'" directive value.

3.4.3.1 CSP Improvement on CSP-deployed Websites

In the first crawling, 13,557 websites deployed CSPs. We filtered out websites deploying CSPs under the report-only mode, with empty policy content, or without any CSP violations on scripts, leaving 11,790 CSP-deployed websites for analysis. Among these 11,790 websites, 3,958 are builder-generated, and 5,562 are CDN-enabled.

Current Safe CSP Deployment and the Room for Potential Improvement. Overall, among the 11,790 CSP-deployed websites, only 359 (3.04%) of them deployed safe CSPs; 59.22% of them deployed unsafe CSPs and have the room to further improve their CSPs to be safe ones; 37.72% of them deployed unsafe CSPs but would be hard to further improve their CSPs to be safe ones due to the need for the cooperation from third parties. Additionally, we found that 57.90% of these 11,790 websites deployed CSPs not for the traditional XSS defense purpose but for frame control and TLS enforcement purposes; this result is similar to what Roth et al. observed in [22].

Among 3,958 CSP-deployed builder-generated websites, only 1.31% of them deployed safe CSPs, while 3.91% of 7,832 CSP-deployed other websites deployed safe CSPs. The Chi-square test of independence found that this difference in the safe CSP deployment is statistically significant. In addition, among the 3,958 CSP-deployed builder-generated websites, 59.82% of them deployed unsafe CSPs and have the room to further improve their CSPs to be safe ones, while 38.88% of them deployed unsafe CSPs but would be hard to further improve their CSPs to be safe ones due to the need for the cooperation from third parties; among the 7,832 CSP-deployed other websites, the two corresponding percentage numbers are 58.93% and 37.14%, respectively.

Among 5,562 CSP-deployed CDN-enabled websites, only 1.76% of them deployed safe CSPs, while 4.19% of 6,228 CSP-deployed other websites deployed safe CSPs. The Chi-square test of independence found that this difference in the safe CSP deployment is also statistically significant. In addition, among the 5,562 CSP-deployed CDN-enabled websites, 52.39% of them deployed unsafe CSPs and have the room to further improve their CSPs to be safe ones, while 45.84% of them deployed unsafe CSPs but would be hard to further improve their CSPs to be safe ones due to the need for the cooperation from third parties; among the 6,228 CSP-deployed other websites, the two corresponding percentage numbers are 65.35% and 30.45%, respectively.

Details on using an "'unsafe-inline'" or an "'unsafe-eval'" directive value. In total, 3,937 out of 11,790 CSP-deployed websites deployed CSPs that contain an "'unsafe-inline'" directive value without using a nonce or a "'strict-dynamic'" directive value. Among these 3,937 websites, 1,349 are builder-generated, and 1,601 are CDN-enabled. We found that only 642 out of these 3,937 websites contain scripts or event handlers inserted into them by third-party scripts. Among these 642 websites, 240 are builder-generated, and 287 are CDN-enabled. These results indicate that most websites (i.e., the remaining 3,295 out of the 3,937 websites) that deployed CSPs with an "'unsafe-inline'" value without using a nonce or a "'strict-dynamic'" directive value have the room to further improve their CSPs to be safer ones by without using an "'unsafe-inline'" value, as they have the control to change their own code without relying on third parties. Meanwhile, 3,716 out of 11,790 CSP-deployed websites deployed CSPs that contain an "'unsafe-eval'" directive value. Among these 3,716 websites, 1,256 are builder-generated, and 1,499 are CDN-enabled. We found that only 880 out of these 3,716 websites contain eval-like functions induced by third-party scripts. Among these 880 websites, 289 are builder-generated, and 385 are CDN-enabled. These results indicate that most websites that deployed CSPs with an "'unsafe-eval'" directive value have the room to further improve their CSPs to be safer ones by without using an "'unsafe-eval'" directive value.

Details on using a "*" or a Scheme Directive Value. In total, 1,908 out of 11,790 CSP-deployed websites deployed CSPs that contain a "*" or a scheme directive value without using a "'strict-dynamic'" directive value. Among these 1,908 websites, 662 are builder-generated, and 804 are CDN-enabled. We further analyzed the number of domains of third-party scripts used on these websites. We found that 1,584 out of these 1,908 websites include third-party scripts from no more than 20 domains. Among these 1,584 websites, 559 are builder-generated, and 686 are CDN-enabled. We also analyzed the domain changes (between the two crawlings) for third-party scripts on these 1,908 websites. In total, 1,875 of them were successfully visited in both crawlings. Among these 1,875 websites, 651 are builder-generated, and 794 are CDN-enabled. We found that overall, only 468 out of these 1875 websites have domain changes for third-party scripts. In more details, 24.27% of the 651 builder-generated websites and 25.32% of 1,224 other websites have domain changes for third-party scripts; 25.06% of the 794 CDN-enabled websites and 24.88% of 1,081 other websites have domain changes for third-party scripts. These results indicate that most websites that deployed CSPs with a "*" or a scheme directive value have the room to improve their CSPs if they would whitelist the domains of third-party scripts instead of using these two types of unsafe directive values.

3.4.3.2 Safe CSP Deployment on Websites without CSPs

In total, 48,330 out of 61,827 websites with CSP violations of script resources did not deploy a CSP in the first crawling. Among these 48,330 websites without CSPs, 17,578 are builder-generated, and 21,021 are CDN-enabled. In addition, among these 48,330 websites without CSPs, 47,092 were successfully visited in both crawlings; among these 47,092 websites, 17,127 are builder-generated, 20,476 are CDN-enabled, and 39,550 do not have domain changes (between the two crawlings) for third-party scripts.

Overall, based on our dataset and analysis, 68.97% of the 48,330 websites without CSPs can deploy safe CSPs. Specifically, 66.57% of 17,578 builder-generated websites without CSPs can deploy safe CSPs, while 71.62% of other websites without CSPs can deploy safe CSPs; the Chi-square test of independence found that this difference in the potential safe CSP deployment is statistically significant. Meanwhile, 62.65% of 21,021 CDN-enabled websites without CSPs can deploy safe CSPs, while 75.28% of other websites without CSPs can deploy safe CSPs; the Chi-square test of independence found that this difference in the potential safe CSP deployment is also statistically significant.

Among 17,578 builder-generated websites without CSPs, 85.44% of them can deploy CSPs without using an "'unsafe-inline'" directive value since they do not contain scripts or event handlers inserted into them by third-party scripts, 73.45% of them can deploy CSPs without using an "'unsafe-eval'" directive value since they do not contain eval-like functions induced by third-party scripts, 90.05% of them can deploy CSPs without using a "*" or a scheme directive value since they include third-party scripts from no more than 20 source domains, and 66.57% of them can deploy safe CSPs. Among 30,692 other websites without CSPs, 88.46% of them can deploy CSPs without using an "'unsafe-inline'" directive value, 76.81% of them can deploy CSPs without using an "'unsafe-eval'" directive value, 93.69% of them can deploy CSPs without using a "*" or a scheme directive value, and 71.62% of them can deploy safe CSPs. Note that 81.90% of 17,127 builder-generated websites without CSPs and 85.18% of 29,965 other websites without CSPs do not have domain changes for third-party scripts.

Among 21,021 CDN-enabled websites without CSPs, 83.79% of them can deploy CSPs without using an "'unsafe-inline'" directive value, 69.68% of them can deploy CSPs without using an "unsafe-eval" directive value, 88.45% of them can deploy CSPs without using a "*" or a scheme directive value, and 62.65% of them can deploy safe CSPs. Among 27,249 other websites without CSPs, 90.11% of them can deploy CSPs without using an "'unsafe-inline'" directive value, 80.13% of them can deploy CSPs without using an "'unsafe-eval'" directive value, 95.38% of them can deploy CSPs without using a "*" or a scheme directive value, and 75.28% of them can deploy safe CSPs. Note that 79.33% of 20,476 CDN-enabled websites without CSPs and 87.96% of 26,616 other websites without CSPs do not have domain changes for third-party scripts.

Takeaways. Great potential exists for CSP improvement or deployment on different types of websites. Most CSP-deployed websites did not deploy safe CSPs, but they have the room to further improve their CSPs to be safe ones; meanwhile, CSP-deployed builder-generated websites and CSP-deployed CDN-enabled websites are less likely to deploy safe CSPs than other websites. For websites without CSPs, most of them have the potential to deploy safe CSPs; meanwhile, builder-generated websites without CSPs and CDN-enabled websites without CSPs are less likely to deploy safe CSPs than other websites.

3.5 Discussion

In this section, we summarize the key results of our work, provide recommendations, and further discuss the limitations of this work and the potential future work.

3.5.1 Summary of Key Results

We have three key findings from our study. First, although most websites include third-party scripts, most of those scripts do not contain or include other third-party scripts which can lead to CSP deployment issues. However, greater percentages of builder-generated websites and CDN-enabled websites contain third-party scripts than other websites. Meanwhile, contrary to the results in the prior studies [4, 22] which considered event handlers as one major hindrance to deploying CSPs without using an "'unsafe-inline'" directive value, we found that websites rarely trigger CSP violations of event handlers inserted into them by third-party scripts.

Second, websites with different web technologies sometimes have different usage patterns of third-party scripts, which can make them encounter different CSP deployment issues. Some web builders and CDNs host and provide third-party scripts that can lead to CSP deployment issues for websites, indicating that the choice of web technologies can impact the deployment of CSPs on the corresponding websites.

Third, greater percentages of CSP-deployed builder-generated websites and CSP-deployed CDN-enabled websites tend to deploy unsafe CSPs than CSP-deployed other websites; nonetheless, most CSP-deployed websites have the room to improve their CSPs to be safe ones. Meanwhile, most websites without CSPs can deploy safe CSPs, although greater percentages of builder-generated websites without CSPs and CDN-enabled websites without CSPs are less likely to deploy safe CSPs than other websites.

3.5.2 Recommendations

Based on the key findings from our study, we provide four recommendations that may help developers deploy safe CSPs or improve their currently deployed CSPs. First, for developers who deploy CSPs with a "*" or a scheme directive value, we recommend them to whitelist domains of third-party script sources (if they would continue to take the simple whitelisting approach) instead of allowing scripts from any sources to be included; for developers who deploy CSPs with an "'unsafe-inline'" or an "'unsafe-eval'" directive value, we recommend them to inspect the necessity of using these unsafe directive values and avoid them if possible.

Second, for developers who would like to build their websites through certain web builders or configure their websites with certain CDNs, we recommend them to inspect whether the web builders or CDNs will introduce some third-party scripts that may incur CSP deployment issues.

Third, for web builder developers who provide templates for website developers to use, we recommend them to make their templates rely on fewer (if not zero) third-party services and avoid introducing untrustworthy third-party script source domains.

Finally, for developers of web builders and CDNs, we recommend them to avoid hosting and providing third-party scripts with eval-like functions. Meanwhile, we recommend them to explicitly describe the sources and behaviors of their provided scripts to facilitate the proper CSP deployment on websites. As a good example, the *Hubspot* web builder provides a list of third-party resource sources to help website developers properly deploy CSPs [52]. Web builder and CDN developers should also avoid making frequent changes to the sources or domains of their hosted third-party resources.

3.5.3 Limitations and Future Work

This study has three major limitations. The first limitation is that we only performed the measurement and analysis on the homepages of websites. A homepage is often the initial point of access for web users, and it should be well protected against attacks. Meanwhile, one prior study found that other webpages of 90% of all analyzed origins of websites that deployed CSPs shared the same CSPs with homepages [10]. However, it would still be valuable for future work to visit more webpages on websites to figure out whether there are some other issues or challenges for developers to properly deploy CSPs on their websites. The second limitation is that our constructed browser extension only emulated two simple user interactions, i.e., scrolling down a webpage to the bottom and clicking on a button (Section 3.3.1). Although that design and implementation choice was partially for the ethical considerations (Section 3.3.4), future work may emulate more user interactions to potentially trigger more dynamic resource accesses while avoiding too much interference with the normal operations of visited websites. The third limitation is that we performed the website crawling only twice. This may not accurately represent the full range of script resource changes on websites. Future work could conduct more crawlings to more comprehensively capture the script resource changes on websites.

3.6 Summary

In this chapter, we analyzed third-party scripts related CSP deployment issues among different types of websites. We constructed a Google Chrome extension to directly insert a specified CSP under the report-only mode into websites to collect CSP violations triggered by the accesses of resources used on webpages. Our analysis is based on the CSP violations triggered under the inserted CSP. Among 61,827 websites that have CSP violations triggered by the accesses of script resources, most websites would not encounter CSP deployment issues. However, compared to other websites, builder-generated websites and CDN-enabled websites are more likely to contain third-party scripts that can lead to CSP deployment issues. Additionally, some web builders and CDNs host and provide scripts that can lead to CSP deployment issues for websites, indicating that the choice of web builders and CDNs can impact the CSP deployment on the corresponding websites. Most CSP-deployed websites have the room to improve their CSPs to be safe ones. Meanwhile, most websites without CSPs can deploy safe CSPs. Based on our key findings, we further provided recommendations to help developers improve their deployed CSPs or deploy safe CSPs on more websites. We will share our constructed Google Chrome extension and dataset with other researchers for their potential future studies.

CHAPTER 4

ANALYZING THE FEASIBILITY OF TAKING GOOGLE'S SECURE CSP APPROACH ON WEBSITES

Content Security Policy (CSP) is a leading security mechanism for mitigating content injection attacks such as Cross-Site Scripting (XSS). Nevertheless, despite efforts from academia and industry, CSP policies (in short, CSPs) are not widely deployed on websites, and deployed CSPs often have security issues or errors. Such low and insecure CSP deployment problems are mainly due to the complexity of the CSP mechanism. Google recently proposed a secure CSP approach that promotes four ***nonce-based CSP solutions***. Nonce-based CSP solution are simpler and more secure compared to traditional whitelisting-based CSP solutions. Google successfully deployed their nonce-based CSP solutions on over 160 services, covering 62% of all outgoing Google traffic. These nonce-based CSP solutions use simple CSPs but provide fine-grained control of web resources; therefore, if widely adopted on many other websites, they can be very helpful on addressing the low and insecure CSP deployment problems. In this chapter, we evaluate the feasibility of adopting Google's nonce-based CSP solutions on the Tranco top 10K websites. We construct a crawling tool to automatically visit websites, simulate user interactions, and insert four CSPs to collect the CSP violations triggered under them. We investigate the adoptability of nonce-based CSP solutions, adoption issues, and the stability of adopting them on websites by analyzing the CSP violations triggered under the inserted CSPs. We found that most websites can adopt the nonce-based CSP solutions on all their webpages visited in our study. For websites that cannot, usually the adoption is hard on around 40% of their webpages. There will be a noticeable increase in the number of websites that can adopt the nonce-based CSP solutions if websites can address the adoption issues caused by first-party resources. Overall, our results can help promote the proper deployment of CSPs.[3]

4.1 Introduction

The extensive functionality of the Web makes it vulnerable to content injection attacks, such as Cross-Site Scripting (XSS), that exploit vulnerabilities in web applications to insert malicious code. To protect web applications from these attacks, major browsers have adopted Content Security Policy (CSP) which is a leading mechanism for protecting webpages against content injection attacks. Web developers can deploy CSP policies (in short, CSPs) to allow permissible resources (e.g., scripts including JavaScript code or files) and behaviors (e.g., form submissions) to be loaded or executed on their websites. There are

[3]This is a joint work with Anhao Xiang and Chuan Yue [6]. Anhao is a graduate research assistant of computer science at the Colorado School of Mines. Chuan is an associate professor of computer science at the Colorado School of Mines.

two CSP deployment modes: **enforcement** and **report-only**. Resources violating CSPs deployed under the enforcement mode will be blocked, whereas resources violating CSPs deployed under the report-only mode will not be blocked but will be reported as CSP violations.

A CSP is composed of directives, and each directive is typically composed of a directive name with a set of directive values. A directive name specifies a certain type of resources to be controlled, while directive values specify the allowed resources or behaviors on a webpage. Different types of resources (e.g., frames, scripts, and fonts) are controlled by the corresponding directives. For example, the "script-src https://script.com " directive is composed of a *script-src* directive name with a script source URL (i.e., *https://script.com*) directive value. This directive only allows scripts from *https://script.com* to be loaded or executed on a webpage. CSPs can be deployed on webpages either via HTTP(S) response headers or via HTML <meta> tags.

While CSP offers a promising defense against content injection attacks, it is not widely deployed on websites yet (e.g., they appeared only on 12.06% websites as reported in a 2020 study [22], and this number increased to 17.74% websites as reported in a 2023 study [1]), and most deployed CSPs have security issues or errors due to the complexity of CSP [2, 8, 22]. Web developers often struggle with proper CSP deployment due to some technical difficulties (e.g., third-party services are blocked by proper CSPs, or it is hard to create or maintain source whitelists of whitelisting-based CSPs [3]). These issues have still not been solved, despite efforts from researchers and organizations.

Google has a significant history in both developing and deploying CSPs to protect their services against content injection attacks [2, 53, 54]. Recently, they proposed a secure CSP approach including four nonce-based CSP solutions with different security levels to balance security and adoption efforts [54]. These nonce-based CSP solutions can help address CSP deployment issues. They use unique nonces or hash values to control scripts without requiring maintaining extensive whitelists of resource sources, which offer a simpler deployment process and enhanced security defense compared to traditional whitelisting-based CSP solutions [2, 54]. Furthermore, they can allow resources that are further requested by already trusted resources to be loaded or executed without matched nonces or hash values, which provides a way to allow dynamic resources and third-party resources. According to the coverage ranges of XSS attack sinks, two of these CSP solutions are recognized as ***weaker nonce-based CSP solutions***, while the other two are recognized as ***stronger nonce-based CSP solutions***. Google successfully deployed their nonce-based CSP solutions on over 160 services, covering 62% of all outgoing Google traffic [54]. This resulted in blocking over 60% of XSS attacks in total, with a notable blockage of 80% of XSS attacks on highly sensitive domains [54].

Meanwhile, the nonce-based CSP solutions provide fine-grained control of web resources since they allow web developers to specify which scripts are allowed to be loaded or executed on a webpage by assigning unique nonces or hash values. Therefore, these nonce-based CSP solutions can be considered as page-level CSP deployment solutions that each webpage deploys a proper CSP which adheres to the principle of least privilege. However, most websites that already deployed CSPs commonly use the same policy on all webpages or the entire origin (e.g., 90% of all analyzed website origins that deployed CSPs shared the same CSP on their webpages [10]). Such a practice fails to guarantee the proper CSP deployment on each webpage.

In this chapter, we focus on analyzing the feasibility of adopting Google's nonce-based CSP solutions on websites. Specifically, we answer the following three research questions under the assumption that websites want to properly deploy CSPs on their webpages: ***RQ1***: To what extent could websites adopt Google's nonce-based CSP solutions?, ***RQ2***: What are the issues encountered by websites that would fail to adopt Google's nonce-based CSP solutions, and how could web developers address them?, and ***RQ3***: How stable could be the adoption of Google's nonce-based CSP solutions on websites? By answering these three research questions, we aim to help promote the proper deployment of CSPs on websites. We implemented a crawling tool to automatically visit webpages of the Tranco [45] top 10K websites, simulate user interactions, insert four CSPs (which can compose the two stronger nonce-based CSP solutions), and collect the corresponding CSP violations. User interactions are simulated as in realistic browsing to trigger more resource accesses for the comprehensive collection of CSP violations. We analyze the feasibility of adopting Google's nonce-based CSP solutions on websites based on CSP violations generated under the inserted CSPs. We use this method because CSP violations directly reflect browsers' policy enforcement and provide a more straightforward way for us to answer our research questions than using HTTP(S) requests. To analyze the stability of adopting the nonce-based CSP solutions on websites, we performed the website crawling three times. We finally collected 7,032, 6,784, and 7,226 unique websites in the first, second, and third crawlings, respectively.

Overall, we found that most (62.88%) websites can adopt Google's nonce-based CSP solutions on all their webpages visited in our study, and a noticeable percentage (21.76%) of them can even adopt the stronger ones on all their webpages. Meanwhile, for websites that cannot adopt the nonce-based CSP solutions on all their webpages visited in our study, usually the adoption is hard on around 40% of their webpages. We further pointed out the issues of adopting weaker and stronger nonce-based solutions on webpages across websites. Additionally, the percentages of websites that can adopt nonce-based CSP solutions (62.88%, 64.43%, and 63.81%, respectively) are similar across three crawlings.

We make three major contributions in this chapter:

- We design a crawling tool to directly insert four CSPs, which compose Google's two stronger nonce-based CSP solutions, under the report-only mode into webpages and collect the corresponding CSP violations. This provides a straightforward way for us to analyze the feasibility of adopting the nonce-based CSP solutions on websites.

- We perform a large-scale measurement study on 10K websites and analyze the feasibility of adopting the nonce-based CSP solutions on their webpages to help promote the proper deployment of CSPs.

- We found that most websites can adopt the nonce-based CSP solutions on all their webpages visited in our study. For websites that cannot, usually the adoption is hard on around 40% of their webpages. Meanwhile, we analyzed the issues of adopting different nonce-based solutions on webpages across websites. Additionally, the percentages of websites that can adopt the nonce-based CSP solutions are similar across three crawlings.

The rest of this chapter is organized as follows. Section 4.2 reviews the background of CSP and the related work. Section 4.3 describes the design of our study from the data collection, processing, and analysis perspectives. Section 4.4 presents the analysis of the feasibility of adopting the nonce-based CSP solutions on websites. Section 4.5 summarizes important findings of our work, makes recommendations, and further discusses the limitations of this work as well as potential future work. Section 4.6 concludes this chapter.

4.2 Background and Related Work

We first review the background of CSP and the related work.

4.2.1 Background of CSP

CSPs are composed of directives. Each directive controls a certain type of resources to be loaded or executed on webpages. CSPs that are deployed in different deployment modes and ways have different effects. We introduce CSP directives, CSP violations, deployment modes, and deployment ways in this section.

CSP Directives. A CSP directive is typically composed of a directive name with a set of directive values. A directive name specifies a certain type of resources to be controlled, while directive values specify the allowed resources or behaviors on a webpage. Developers can control web resources by whitelisting resource sources or using nonces or hash values as directive values in CSPs. Resources from whitelisted sources or with matched nonces or hash values are allowed to be loaded or executed. However, CSP also provides the following ***unsafe directive values*** which can cause security issues: (1) the " 'unsafe-inline' " directive value allows any scripts and event handlers included in the first-party websites to be loaded or executed, (2) the " 'unsafe-eval' " directive value allows any eval-like functions to be executed, (3) the " 'wasm-unsafe-eval' " directive value allows any wasm eval functions to be executed, (4) the "*" value allows resources from any domains to be loaded or executed, and (5) scheme values (e.g., an "https:" value) allow scripts from any sources with the corresponding scheme values to be executed.

CSP Violations. When a resource access (e.g., retrieving a JavaScript file) or behavior (e.g., executing a JavaScript function) violates the CSP deployed on a webpage, it will trigger a CSP violation on a browser. Developers can send CSP violation reports to the specified hosts or servers via reporting directives (i.e., a "report-uri" directive or a "report-to" directive). CSP violations contain information about violating resources such as their source files, their types, and the violated CSP directives.

CSP Deployment Modes and Ways. CSP has two deployment modes: *enforcement* and *report-only*. In the enforcement mode, browsers will block resources and disable behaviors that violate the deployed CSPs. In report-only mode, browsers will only report CSP violations (typically for web developers to test policies), without taking any enforcement actions. Meanwhile, developers can deploy CSPs on their webpages in two ways: via an HTTP(S) response header or an HTML <meta> tag. Developers can deploy a CSP in the enforcement mode via a "Content-Security-Policy" HTTP(S) response header or an HTML <meta> tag. However, developers can deploy a CSP in the report-only mode only via a "Content-Security-Policy-Report-Only" HTTP(S) response header. In addition, content that appears before a CSP in an HTML <meta> tag will not be controlled [19].

4.2.2 Related Work

CSP Deployment Problems. Although CSP is a leading mechanism for protecting webpages against content injection attacks, it is not widely deployed in practice [1, 22], and most deployed CSPs have security issues or deployment errors [1, 2, 4, 22]. Weichselbaum et al. found that 94.72% of deployed CSPs had policy misconfigurations and insecure whitelist entries [2], which led to ineffective protection against XSS attacks. Calzavara et al. found that above 90% of CSP-deployed websites still could not effectively defend against XSS attacks, due to the usage of the 'unsafe-eval' or 'unsafe-inline' directive value [23]. They further proposed Compositional CSP, which extends CSP by dynamically composing a webpage with needed policies at runtime [24]. Meanwhile, they found that around 90% of all analyzed website origins that deployed CSPs just shared the same CSPs on all their webpages, and only 8.8% of analyzed origins that deployed CSPs have at least one secure CSP [10]. Roth et al. conducted a longitudinal study to analyze CSP deployment evolution [22]. They found that CSP has been increasingly used for other purposes (such as frame control and TLS enforcement) in addition to the traditional content restriction purpose, and insecure practices (such as using an 'unsafe-eval' or 'unsafe-inline' directive value) were present on 90% of 421 websites that deployed CSPs for script restriction. By taking the unsupervised clustering approach, Ren et al. found that most deployed CSPs do not sufficiently protect webpages from the directive coverage and secure use perspectives [1].

Underlying Reasons for CSP Deployment Problems. Several studies investigated the underlying reasons for CSP deployment problems. Roth et al. conducted a qualitative study involving semi-structured interviews with 12 web developers and found that legacy code, web frameworks, third-party code, the complexity of the CSP mechanism, etc., could all block a proper CSP deployment [3]. Steffens et al. analyzed resource usage on websites by using in-browser hooks and found that proper CSP deployment on most first-party websites requires the cooperation from third parties [4]. Meanwhile, these studies opined that event handlers were one major reason for websites to deploy CSPs with an " 'unsafe-inline' " directive value instead of deploying nonce-based CSPs. However, analyzing website resource usage via in-browser hooks or interviewing a small number of web developers may not reveal the real scenarios of how nonce-based CSP solutions work on websites. In our work, we analyze the feasibility and issues of adopting the nonce-based CSP solutions on large-scale websites directly based on CSP violations.

Automated CSP Generation. To promote CSP adoption, some researchers focused on automated CSP generation. Fazzni et al. designed AutoCSP to identify scripts that should be whitelisted, and to automatically generate CSPs for webpages by modifying the server-side code [25]. Pan et al. designed CSPAutoGen to automatically generate CSPs for webpages by importing inline and dynamic scripts as external scripts [26]. However, these studies focused on automated whitelisting-based CSP generation. Compared to traditional whitelisting-based CSP solutions, Google's nonce-based CSP solutions are simpler and more secure, and they can help address CSP deployment issues (Section 4.1). Our work and the findings could benefit future studies of automated nonce-based CSP generation by shedding light on the feasibility of the nonce-based CSP solutions on websites thus the proper deployment of CSPs.

Google's Nonce-Based CSP Solutions. Recently, Google proposed four ***nonce-based CSP solutions*** as shown in Table 4.1. These four nonce-based CSP solutions provide protection at different security levels, and all of them do not allow plugins that are embedded in <object>, <embed>, or <applet> tags and URLs in <base> tags. The nonce-based CSP solution at level 4, providing the highest security, only allows scripts matched with nonces or hash values to be loaded or executed. In addition to allowing scripts with matched nonces or hash values, the nonce-based CSP solution at level 3 also allows non-parser-inserted scripts requested by already trusted scripts to be loaded or executed without matched nonces or hash values. The nonce-based CSP solutions at levels 3 and 4 are recognized as the ***stronger nonce-based CSP solutions*** since they can cover most XSS sinks.

Table 4.1 Google's Four Nonce-based CSP Solutions Categorized by Decreasing Security Levels

CSP Solution	***Security Level***	***Security Categorization***
script-src 'nonce-r4nd0m'; object-src 'none'; base-uri 'none';	4	Stronger
script-src 'nonce-r4nd0m' 'strict-dynamic'; object-src 'none'; base-uri 'none';	3	Stronger
script-src 'nonce-r4nd0m' 'strict-dynamic' 'unsafe-eval'; object-src 'none'; base-uri 'none';	2	Weaker
script-src-attr 'unsafe-hash' 'hash_values'; script-src-elem 'nonce-r4nd0m' 'strict-dynamic' 'unsafe-eval'; object-src 'none'; base-uri 'none';	1	Weaker

The nonce-based CSP solution at level 2, while similar to the nonce-based CSP solution at level 3, further allows eval-like functions to be executed on webpages. The nonce-based CSP solution at level 1, compared to the nonce-based CSP solution at level 2, additionally allows event handlers with matched hash values. However, it does not ensure event handlers execute as the developer intends. The nonce-based CSP solutions at levels 1 and 2 are recognized as the ***weaker nonce-based CSP solutions*** since they can cover fewer XSS sinks due to the usage of an " 'unsafe-eval' " or " an 'unsafe-hash' " directive value. Google successfully implemented their nonce-based CSP solutions on 160 services, which blocked over 60% XSS attacks in total [54]. The nonce-based CSP solutions are simpler and more secure compared to traditional whitelisting-based CSP solutions, and it provides fine-grained control of resources and can potentially address the CSP deployment problems (Section 4.1). Therefore, we explore the feasibility of adopting Google's nonce-based CSP solutions on websites to help promote the proper CSP deployment.

4.3 Design of Our Study

In this section, we describe the design of our study from the data collection, processing, and analysis perspectives. Our unique goal in this study is to analyze the feasibility of adopting Google's nonce-based CSP solutions on websites. There are two ways to conduct web measurements: utilizing vantage points of web archives (typically using Internet Archive (IA)) to perform an archive-based measurement, or constructing web crawlers to perform a traditional live measurement [55]. However, a large-scale analysis of dynamic web content is difficult when using the IA due to the rate-limiting policy, and IA aggregates data from multiple contributors which will cause a bias in results [55]. To achieve our goal, we use a traditional way to perform a live measurement by constructing a crawling tool to automatically visit webpages for each website, directly insert four CSPs under the report-only mode to each webpage, and collect the CSP violations triggered under the inserted CSPs. The feasibility of adopting the nonce-based CSP solutions on websites is analyzed based on the CSP violations triggered under the inserted CSPs. CSP violations directly reflect browsers' policy enforcement and provide a more straightforward way for us to answer our research questions than analyzing resource usage of webpages via HTTP(S) requests. Our crawling tool performs a full browser and stateless crawling to obtain web content for a real user visit and avoid the impact of the webpage access order on our results.

4.3.1 Website Crawling and Data Collection Framework

Figure 4.1 illustrates our website crawling and data collection framework. Our constructed crawling tool consists of four components: a *CSP inserter*, a *user interaction simulator*, a *request and response collector*, an *HTML document collector*. It instructs the browser, starting from the landing page of a website, to collect same-site links from webpages and sequentially visit webpages according to the collected links. We recursively repeat the process until we collect up to 101 webpages (including one landing page and 100 subpages) for each website or exhaust the webpages to visit. Our crawling tool waits up to 30 seconds to load a webpage. After each webpage is fully loaded, it performs user interactions and collects the original CSPs (if they exist) deployed on the webpage and CSP violations generated under inserted CSPs for 30 seconds.

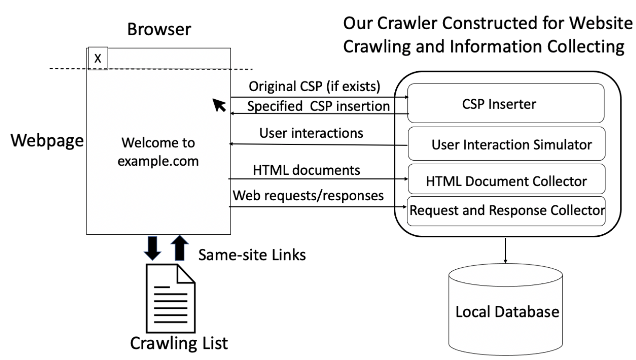


Figure 4.1 Website Crawling and Data Collection Framework

To collect CSP violations for evaluating the feasibility of adopting the nonce-based CSP solutions on websites, the CSP inserter component inserts four CSPs under the report-only mode via an HTTP(S) response header for each webpage. Figure 4.2 shows detailed directives of four inserted CSPs. The first CSP allows scripts with an 'f6393661395af2-5044660c187c2cd6c4' nonce value and non-parser-inserted scripts required by already trusted scripts to be loaded or executed. The second CSP only allows scripts with an 'f6393661395af25044660c187c2cd6c8' nonce value to be loaded or executed. The third CSP does not allow plugins embedded in <object>, <embed>, or <applet> tags to be loaded or executed. The fourth CSP does not allow URLs in <base> tags to be fetched on a webpage. The first, third, and fourth CSPs compose the nonce-based CSP solution at level 3, while the second, third, and fourth CSPs compose the nonce-based CSP solution at level 4. We evaluate the adoptability of nonce-based CSP solutions on webpages by analyzing CSP violations generated under these two nonce-based CSP solutions. Each triggered CSP violation contains information about the violating resource and the violated CSP, allowing us to determine under which CSP the violation is generated. Because these four CSPs are inserted under the report-only mode, they will not block any violating resources but will only generate CSP violations for resources, and they do not impact the CSP violation generation for each other. Due to the specification of the "report-uri" directive, CSP violations will be wrapped as HTTPS requests and sent to our server. Due to the specification of the "report-sample" directive value, CSP violations triggered by the accesses of scripts will contain a sample code.

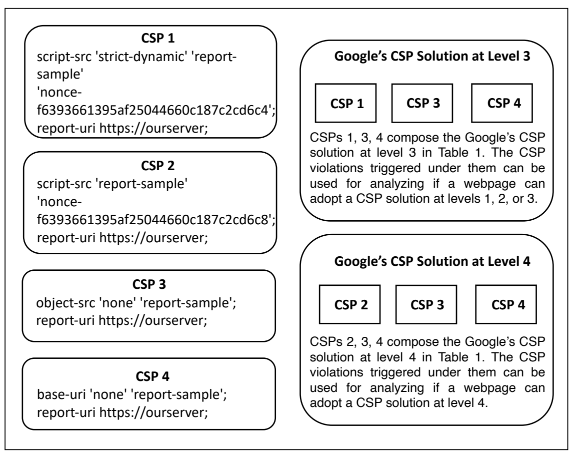


Figure 4.2 Four Inserted CSPs for Webpages

To achieve a more comprehensive collection of CSP violations by accessing more web resources, the user interaction simulator simulates two common user interactions after each webpage is fully loaded: scrolling down the webpage to the bottom and clicking on the first identified button (which can be a submit, reset, checkbox, or radio button) on the webpage. If the user interaction simulator cannot find an identified button to click on the webpage, it will find a link, an input box, or an image to click instead. The request and response collector intercepts all HTTP(S) requests and the corresponding responses generated for a webpage. The HTML document collector saves the loaded HTML document when a webpage is fully loaded. We determine whether a webpage is successfully visited according to its HTML document. All collected data will be saved in a local database. CSP violations will be extracted from HTTP(S) requests, and original CSPs (if they exist) will be extracted from HTTP(S) response headers. Note that our crawling tool does not remove the original CSPs (if they exist) on webpages when it inserts specified CSPs. CSP violations triggered under our inserted CSPs are based on currently allowed resources. Meanwhile, our crawling tool does not check whether there is a CSP deployed via an HTML <meta> tag in a webpage because in pratice only a very small number of websites deployed a CSP via an HTML <meta> tag [22].

4.3.2 Data Collection and Dataset

We visited webpages of the top 10K websites from the Tranco [45] list (dated November 8th, 2023), using ten computers with identical configurations, from November 2023 to January 2024. Each computer is installed with a Google Chrome browser (Version 104) on a Ubuntu 22.04 LTS system. To analyze the stability of adopting the nonce-based CSP solutions on websites, we performed the website crawling three times. Our three web crawlings were completed over the course of ten weeks. In total, 7,801 websites with 525,822 webpages, 7,378 websites with 481,049 webpages, and 7,998 websites with 540,179 webpages were successfully visited in the first, second, and third crawlings, respectively. As in studies such as [10], we denote a website as an eTLD+1 (effective top-level domain + 1) entity, like google.com or bbc.co.uk. Webpages sharing the same eTLD+1 are considered to be from the same website. We remove duplicate websites and webpages, which leaves us 7,032 unique websites with 484,752 webpages, 6,784 unique websites with 452,692 webpages, and 7,226 unique websites with 492,910 webpages in the first, second, and third crawlings, respectively. Our analysis of the potential adoption of Google's nonce-based CSP solution on websites (Section 4.4.1) and the issues encountered by websites that would fail to adopt them (Section 4.4.2) is based on the 7,032 websites visited in the first crawling. The analysis of the stability of the nonce-based CSP solution adoption on websites is based on websites visited in three crawlings (Section 4.4.3).

4.3.3 Evaluation Methods for the Feasibility of Adopting Nonce-Based CSP Solutions

The feasibility of adopting the nonce-based CSP solutions on a website is evaluated based on the highest security level of the nonce-based CSP solution that can be achieved on each visited webpage. To evaluate which security level of a nonce-based CSP solution can be achieved on a webpage, we assess whether all CSP violations under a certain nonce-based CSP solution can be resolved by developers. A CSP violation is considered ***resolvable*** under a nonce-based CSP solution if the CSP solution allows its reported violating resource to be loaded or executed by adding nonces or hash values without code refactoring (i.e., restructuring existing code while preserving its functionality) performed by web developers using code refactoring tools. We differentiate CSP violations generated by the accesses of third-party resources from those generated by the accesses of first-party resources to accurately determine the resolvability of CSP violations under a nonce-based CSP solution.

4.3.3.1 Feasibility Evaluation of Adopting Nonce-based CSP Solutions on Websites

Google proposed four nonce-based CSP solutions with different security levels as shown in Table 4.1 (Section 4.2.1). They encourage web developers to adopt the stronger nonce-based CSP solutions (which are at or above level 3), as these CSP solutions can cover most XSS sinks on webpages[54]. To determine the feasibility of adopting nonce-based CSP solutions on a website, we check which security level of nonce-based CSP solutions can be achieved on each webpage visited in our study. If all visited webpages of a website can adopt the nonce-based CSP solutions at or above level 1, we consider the website can adopt ***the nonce-based CSP solutions***. If all visited webpages of a website can adopt the nonce-based CSP solutions at or above level 3, we consider that the website can adopt ***the stronger nonce-based CSP solutions***.

4.3.3.2 Evaluation for the Highest Security Level of CSP Solutions on Webpages

For each webpage, we evaluate the nonce-based CSP solution at the highest security level it can adopt. Algorithm 3 is designed to determine the highest security level of nonce-based CSP solutions a webpage can adopt. Given CSP violations W_Vio generated under the four inserted CSPs of a webpage and four empty lists $CSP\_solution\_level\_i$ $(i = 1, 2, 3, 4)$ for recording the resolvability of CSP violations under each nonce-based CSP solution, our algorithm will automatically return the highest security level H_CSP_Sol of the nonce-based CSP solutions that the webpage can adopt. In more details, H_CSP_Sol is initialized as 0 in Line 1, which indicates that there are no nonce-based CSP solutions adoptable for the webpage. From Line 2 to Line 11, for each nonce-based CSP solution, the algorithm determines whether a CSP violation *csp_vio* is resolvable under the solution. If a CSP violation *csp_vio* is resolvable under the CSP solution, the corresponding list $CSP\_solution\_level\_i$ (i=1,2,3,4) that records the resolvability of CSP violations will add a "RES" value. Otherwise, the corresponding list will add an "UNRES" value to represent that the CSP violation *csp_vio* is unresolvable under the CSP solution. For the nonce-based CSP solution at level 4, the algorithm determines the resolvability of CSP violations generated under the inserted CSPs 2, 3, and 4 in Figure 4.2. For the nonce-based CSP solutions at levels 1, 2, and 3, the algorithm determines the resolvability of CSP violations generated under the inserted CSPs 1, 3, and 4 in Figure 4.2.

As shown in Table C.1 in Appendix C.1, under the nonce-based CSP solution at level 4, CSP violations triggered by the accesses of five types of resources are resolvable by adding nonces: (1) inline scripts, (2) first-party scripts requested by the current webpage, (3) first-party scripts requested by other scripts, (4) scripts inserted into the current webpage by first-party scripts, and (5) third-party scripts directly requested by the current webpage. Under the nonce-based CSP solution at level 3, CSP violations triggered by the accesses of eight types of resources are resolvable: (1) inline scripts, (2) first-party scripts directly requested by the current webpage, (3) third-party scripts directly requested by the current webpage, (4) non-parser-inserted scripts inserted into the current webpage by first-party scripts, (5) non-parser-inserted scripts inserted into the current webpage by third-party scripts, (6) first-party scripts requested by other scripts, (7) third-party scripts requested by other scripts, and (8) parser-inserted scripts inserted into the current webpage by first-party scripts. These resources can be allowed by a "strict-dynamic" directive value or by adding nonces under the nonce-based CSP solution at level 3. In addition to resolvable CSP violations under the nonce-based CSP solution at level 3, CSP violations triggered by the accesses of eval-like functions induced by first-party and third-party scripts are also resolvable under the nonce-based CSP solution at level 2 due to the usage of the " 'unsafe-eval' " directive value. In addition to resolvable CSP violations under the nonce-based CSP solution at level 2, CSP violations triggered by the accesses of event handlers inserted by first-party scripts are also resolvable under the nonce-based CSP solution at level 1 due to the usage of the " 'unsafe-hash' " directive value.

From Line 12 to Line 16, our algorithm determines the nonce-based CSP solutions that a webpage can adopt. For each nonce-based CSP solution, our algorithm checks the corresponding list $CSP\_solution\_level\_i$ (i=1,2,3,4). If there is no "UNRES" value in the list, we consider the webpage can adopt the nonce-based CSP solution. The algorithm records the highest security level H_CSP_Sol of nonce-based CSP solutions that a webpage can adopt and returns this value.

4.3.3.3 Third-party Resource Identification

As in studies such as [4, 10], we use the notion of an eTLD+1 to distinguish first and third parties. To differentiate third parties more accurately by considering that hosts with different eTLD+1 may actually belong to the same party or organization, we refined our results based on the domain relationships provided by the webXray domain owner list [48] and the domain owner list derived by Steffens et al. in [4]. Note that webXray was also used by Steffens et al. in [4]. The "blocked-uri", "violated-directive", and "source-file" fields of CSP violations generated by Google Chrome indicate the types (e.g., event handlers, inline scripts, or eval-like functions) of the reported resources and resource sources.

4.3.4 Ethical Considerations and Reproducibility

Our data collection process (Section 4.3.2) does not raise obvious ethical issues. It does not involve human subjects or potentially sensitive data (e.g., user behavior or social network information, etc.). Our

Algorithm 3 Determining the Highest Security Level of Nonce-Based CSP Solutions a Webpage Can Adopt

Input: W_Vio is a set of CSP violations generated under the four inserted CSPs of a webpage
Input: $CSP\_solution\_level\_i$ ($i = 1, 2, 3, 4$) is an empty list to record the resolvability of CSP violations for a certain CSP solution
Output: the highest security CSP solution H_CSP_Sol that the webpage can adopt

```
H_CSP_Sol ← 0
for each csp_vio in W_Vio do
    for i= 4 to 1 do
        if i==4 and csp_vio is generated by inserted CSP 1 (Figure 4.2) then
            continue;
        if i!=4 and csp_vio is generated by inserted CSP 2 (Figure 4.2) then
            continue;
        if csp_vio is resolvable under the CSP solution at level i then
            Append(CSP_solution_level_i, "RES")
        else
            Append(CSP_solution_level_i, "UNRES")
for i= 4 to 1 do
    if "UNRES" not in CSP_solution_level_i then
        H_CSP_Sol ← i
        break
return H_CSP_Sol
```

constructed crawling tool only visited the public webpages of websites. On each webpage, our tool inserts the specified CSPs, scrolls down to the bottom, only clicks on one button to collect HTTP(S) requests and responses (Section 4.3.1). Therefore, there is minimal or even no change to the click-through rate of ads, and the traffic burden incurred to each visited website is also minimal. In web measurement studies, reproducibility is a requirement that is crucial to sustainable research [49–51]. Our work is reproducible because we provide the details of our website crawling and data collection framework, our data collection process and dataset, and our data processing and analysis methods in this section as suggested in studies such as [49–51]; meanwhile, our constructed crawling tool and dataset will be open-sourced along with the publication of this chapter.

4.4 Results and Analysis

In this section, we answer the three research questions about the extent to which websites could adopt Google's nonce-based CSP solution (Section 4.4.1), the issues encountered by websites that would fail to adopt them (Section 4.4.2), and the stability of the nonce-based CSP solution adoption on websites across three crawlings (Section 4.4.3).

4.4.1 Adoptability of the Nonce-Based CSP solutions

To answer RQ1, we investigate the lowest security level of adoptable nonce-based CSP solutions on websites and the webpage distribution of adopting the nonce-based CSP solutions across websites (Section 4.4.1.1). Meanwhile, we analyze the improvement of deployed whitelisting-based CSPs on websites (Section 4.4.1.2).

4.4.1.1 Adoption Patterns of Nonce-Based CSP Solutions Across Websites

Website Distribution of the Lowest Security Level of Nonce-Based Adoptable CSP Solutions. If at least one webpage can adopt a CSP solution at a certain security level i (i=1,2,3,4), meanwhile, each of the remaining webpages is able to adopt nonce-based CSP solution at or above this level, we refer to this website as having ***the lowest security level of adoptable nonce-based CSP solutions*** at level i. Figure 4.3 illustrates the distribution of those 7,032 websites regarding the percentages of them having the lowest security levels of adoptable nonce-based CSP solutions at 1, 2, 3, and 4, as well as having at least one of their webpages visited in our study unable to adopt nonce-based CSP solutions. Overall, we found that 2,610 (37.12%) of 7,032 websites cannot adopt the nonce-based CSP solutions since they have at least one of their webpages visited in our study that cannot adopt the nonce-based CSP solutions at or above level 1, while 4,422 (62.88%=9.74%+12.02%+23.26%+17.86%) of 7,032 websites can adopt the nonce-based CSP solutions since all their webpages visited in our study can adopt nonce-based CSP solutions at or above level 1.

Specifically, among 7,032 websites, 9.74% of them can adopt the nonce-based CSP solutions at level 4 on all their webpages visited in our study; 12.02% of them have their lowest security level of adoptable nonce-based CSP solutions at level 3; 23.26% of them have their lowest security level of adoptable nonce-based CSP solutions at level 2; 17.86% of them have their lowest security level of adoptable nonce-based CSP solutions at level 1. It is worth noting that 1,530 (21.76%=9.74%+12.02%) of 7,032 websites can even adopt the stronger nonce-based CSP solutions since all their webpages visited in our study can adopt the nonce-based CSP solutions at or above level 3.

Webpage Distribution of Adopting Nonce-Based CSP Solutions across Websites Unable to Adopt Nonce-Based CSP Solutions. For each of those 2,610 websites that cannot adopt nonce-based CSP solutions, we further checked the distribution of the webpages visited in our study regarding what percentages of them can adopt nonce-based CSP solutions at levels 1, 2, 3, and 4, as well as cannot adopt any nonce-based CSP solutions. Figure 4.4 is the boxplot of such distributions across all those 2,610 websites. On average, the percentages of webpages that can adopt nonce-based CSP solutions at levels 1, 2, 3, and 4, and those that cannot adopt any nonce-based CSP solutions, are 10.48%, 25.36%, 10.87%,

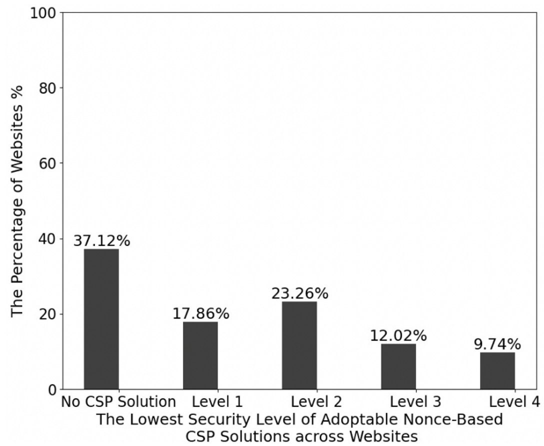


Figure 4.3 The Lowest Security Level of Adoptable Nonce-Based CSP Solutions across 7,032 Websites

13.31%, and 39.99%, respectively, across those 2,610 websites. Meanwhile, the boxes, the whiskers, and the median values are similar for webpages that can adopt nonce-based CSP solutions at levels 1, 3, and 4. The median value of the percentage of webpages that can adopt nonce-based CSP solutions at levels 1, 3, and 4 across websites are 0%, 0%, and 1.00%, respectively. The corresponding percentage number of webpages that cannot adopt any nonce-based CSP solutions and webpages that can adopt nonce-based CSP solutions at level 2 are 22.84% and 4.86%, respectively. We can see that for websites that cannot adopt the nonce-based CSP solutions, usually the adoption is hard on around 40% of their webpages with a noticeable percentage of webpages that can adopt nonce-based CPS solutions at level 2.

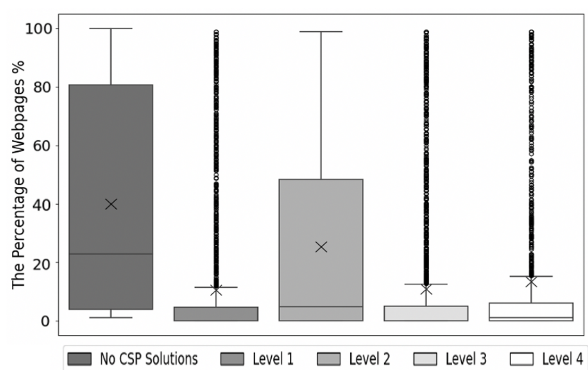


Figure 4.4 Distribution of the webpages visited in our study regarding what percentages of them can adopt nonce-based CSP solutions at levels 1, 2, 3, and 4, as well as cannot adopt any nonce-based CSP solutions, respectively, across 2,610 websites that cannot adopt nonce-based CSP solutions. The x marks represent the mean values.

Webpage Distribution of Adopting Nonce-Based CSP Solutions across Websites Able to Adopt Nonce-Based CSP Solutions. Similarly, for each of those 4,422 websites that can adopt nonce-based CSP solutions, we further checked the distribution of the webpages visited in our study regarding what percentages of them can adopt nonce-based CSP solutions at levels 1, 2, 3, and 4. Figure 4.5 shows the boxplot of such distributions across all those 4,422 websites. On average, the percentages of webpages that can adopt nonce-based CSP solutions at levels 1, 2, 3, and 4 are 13.46%, 34.93%, 26.25%, and 25.36%, respectively, across those 4,422 websites. The median values of the percentages of webpages that can adopt nonce-based CSP solutions at levels 1, 2, 3, and 4 across websites are 0%, 1.11%, 0% and 1.07%, respectively. We can see that for those 4,422 websites, their webpages can often adopt the nonce-based CSP solutions at level 2, with a noticeable percentage of them that can even achieve nonce-based CSP solutions at levels 3 and 4.

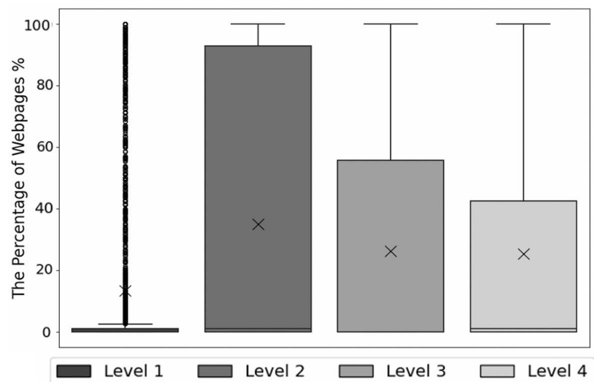


Figure 4.5 Distribution of the webpages visited in our study regarding what percentages of them can adopt nonce-based CSP solutions at levels 1, 2, 3, and 4, respectively, across 4,422 websites that can adopt nonce-based CSP solutions. The x marks represent the mean values.

Takeaways. Overall, most websites can adopt the nonce-based CSP solutions, and a noticeable percentage of them can even adopt the stronger nonce-based CSP solutions. For websites that cannot adopt the nonce-based CSP solutions, usually the adoption is hard on around 40% of their webpages, while a noticeable percentage of their webpages can adopt nonce-based CSP solutions at level 2.

4.4.1.2 Potential CSP Improvement

In total, 113,863 webpages from 3,041 websites deployed CSPs under the enforcement mode. We found that 109,878 (96.50%) out of 113,863 webpages deployed whitelisting-based CSPs. Among 109,878 webpages that deployed whitelisting-based CSPs, 45,669 (41.45%) of them deployed whitelisting-based CSPs for content control. Among these 45,669 webpages, 83.46% of them deployed whitelisting-based CSPs using an " 'unsafe-inline' " directive value which allows any inline or embedded scripts to be loaded or executed, and 54.79% of them deployed whitelisting-based CSPs using a "*" directive value or scheme directive values (i.e., an "http:" directive value, an "https:" directive value, or an "data:" directive value) which allow resources from any source URLs to be loaded or executed. We explore the feasibility of adopting the nonce-based CSP solutions on those 109,878 webpages that deployed whitelisting-based CSPs. We found that only 16.95% of them cannot adopt any nonce-based CSP solutions, while 17.33%, 27.11%, 17.36%, and 21.23% of them can adopt the nonce-based CSP solutions at levels 1, 2, 3, and 4, respectively. We can see that most webpages that deployed whitelisting-based CSPs can adopt the nonce-based CSP solutions.

Furthermore, we found 440 websites that deployed whitelisting-based CSPs on all their webpages visited in our study, and 373 of them deployed the same whitelisting-based CSP on all their webpages. Among those 440 websites, 55.22% of them can adopt the nonce-based CSP solutions, and 21.36% of them can even adopt the stronger nonce-based CSP solutions. Among 373 websites that deployed the same whitelisting-based CSP on all their webpages visited in our study, 152 deployed whitelisting-based CSPs for content control, among which only 19 have no security issues. It is worth noting that 57.64% of those 373 websites can adopt the nonce-based CSP solutions; 24.12% of them can even adopt the stronger nonce-based CSP solutions.

Takeaways. Most deployed whitelisting-based CSPs are not used for content control, and those used for content control often have security issues. Fortunately, most webpages that deployed them can adopt the nonce-based CSP solutions which are simpler and more secure. Meanwhile, for websites that deployed the same whitelisting-based CSP on all their webpages, most of them can adopt the nonce-based CSP solutions, and a noticeable percentage of them can even adopt the stronger ones.

4.4.2 Adoption Issues of Nonce-Based CSP Solutions

To answer RQ2, we investigate the adoption issues of the weaker nonce-based CSP solutions for websites that cannot adopt the nonce-based CSP solutions (Section 4.4.2.1), and the adoption issues of the stronger nonce-based CSP solutions for websites that cannot adopt the stronger nonce-based CSP solutions (Section 4.4.2.2). Meanwhile, we explore the promotion of adopting the nonce-based CSP solutions by addressing adoption issues caused by first-party resources (Section 4.4.2.3).

4.4.2.1 Upgrade to the Weaker Nonce-Based CSP Solutions

Overall, 2,610 websites contain webpages that cannot adopt any nonce-based CSP solutions. To adopt the nonce-based CSP solution at level 1 on webpages that cannot adopt any nonce-based CSP solutions across these 2,610 websites, 45.82% of them need to address the adoption issues of URLs of first-party resources in <base> tags, while 12.87% of them need to address the adoption issues of URLs of third-party resources in <base> tags; 7.43% of them need to address the adoption issues of plugins embedded into

<object>, <embed>, or <applet> tags by first-party scripts, while 0.69% of them need to address the adoption issues of plugins embedded into <object>, <embed>, or <applet> tags by third-party scripts; 54.25% of them need to address the adoption issues of parser-inserted scripts inserted by third-party scripts. To adopt the nonce-based CSP solutions at level 2 on webpages that cannot adopt any nonce-based CSP solutions, besides addressing the adoption issues of nonce-based CSP solutions at level 1, 42.72% of those 2,610 websites need to further address the adoption issues of event handlers inserted by first-party scripts, while 21.03% of them need to address the adoption issues of event handlers inserted by third-party scripts on webpages.

Takeaways. To adopt the weaker nonce-based CSP solutions on webpages unable to adopt any nonce-based CSP solutions, most websites need to address the adoption issues of parser-inserted scripts inserted by third-party scripts on webpages.

4.4.2.2 Upgrade to the Stronger Nonce-Based CSP Solutions

Overall, 2,381 websites contain webpages that can adopt the nonce-based CSP solutions at level 1. To adopt the nonce-based CSP solutions at level 3 on these webpages, all the 2,381 websites need to address the adoption issues of event handlers inserted by first-party scripts, while none of them need to address the adoption issues of event handlers inserted by third-party scripts; 43.05% of them need to address the adoption issues of eval-like functions induced by first-party scripts, while 54.01% of them need to address the adoption of eval-like functions induced by third-party scripts. To adopt the nonce-based CSP solutions at level 4 on webpages that can adopt the nonce-based CSP solutions at level 1, besides addressing the adoption issues of eval-like functions and inserted event handlers as adopting the nonce-based CSP solutions at level 3, 84.54% of those 2,381 websites also need to address the adoption issues of third-party scripts further requested by other scripts; 48.55% of them need to address the adoption issues of scripts inserted by third-party scripts on webpages.

Overall, 4,122 websites contain webpages that can adopt the nonce-based CSP solutions at level 2. To adopt the nonce-based CSP solutions at level 3 on webpages that can adopt the nonce-based CSP solution at level 2 across 4,122 websites, 68.34% of them need to address the adoption issues of eval-like functions induced by first-party scripts, and 74.76% of them need to address the adoption issues of eval-like functions induced by third-party scripts. To adopt the CSP solution at level 4 on those webpages, besides addressing the adoption issues of eval-like functions as adopting the nonce-based CSP solutions at level 3, 88.11% of those 4,122 websites also need to address the adoption issues of third-party scripts that are further requested by other scripts, and 56.04% of them also need to address the adoption issues of scripts that are inserted by third-party scripts.

Adoption Issues Caused by Third-party Resources. Prior studies opined that event handlers were one major reason for websites to deploy CSPs with an " 'unsafe-inline' " directive value instead of using nonces [3, 4]. However, based on our analysis results, we found that among 2,610 websites that cannot adopt the nonce-based CSP solutions, only 21.03% of them are due to event handlers inserted by third-party scripts. To adopt the weaker nonce-based CSP solutions on webpages unable to adopt any nonce-based CSP solutions, most websites need to address the adoption issues of parser-inserted scripts inserted by third-party scripts. To adopt the stronger nonce-based CSP solutions on webpages that cannot adopt stronger nonce-based CSP solutions, most websites need to address the following three third-party scripts related adoption issues: (1) eval-like functions induced by third-party scripts, (2) third-party scripts requested by other resources, and (3) scripts inserted by third-party scripts on webpages.

Takeaways. Adoption of the nonce-based CSP solutions on websites is primarily hindered by the adoption issues of parser-inserted scripts inserted by third-party scripts. Meanwhile, adoption of the stronger nonce-based CSP solutions on websites is primarily hindered by the adoption issues of event handlers inserted by first-party scripts, eval-like functions induced by first-party resources, eval-like functions induced by third-party resources, third-party scripts requested by other resources, and scripts inserted by third-party scripts on webpages.

4.4.2.3 Promotion of Adopting Nonce-Based CSP Solutions by Addressing Adoption Issues Caused by First-Party Resources

The adoption issues of the nonce-based CSP solutions caused by third-party resources may be hard to address, while web developers can possibly address the adoption issues caused by first-party resources. We further explore the promotion of adopting the nonce-based CSP solutions on websites if websites can successfully address the adoption issues caused by first-party resources. Figure 4.6 illustrates the changes of the lowest security level of adoptable nonce-based CSP solutions across 7,032 websites before and after addressing the adoption issues caused by first-party resources. We found that if websites can address the adoption issues caused by first-party resources, the number of websites that can adopt the nonce-based CSP solutions will increase by 13.31%. Meanwhile, websites for which the lowest security level of adoptable nonce-based CSP solutions is level 1 can adopt the nonce-based CSP solutions at or above level 2 on all their visited webpages since event handlers used on these websites are all inserted by first-party resources. Additionally, the number of websites that have their lowest security level of adoptable nonce-based CSP solutions at levels 2, 3, and 4 will increase by 8.59%, 15.70%, and 6.88%, respectively. In total, the number of websites that can adopt the stronger nonce-based CSP solutions will increase by 22.58%.

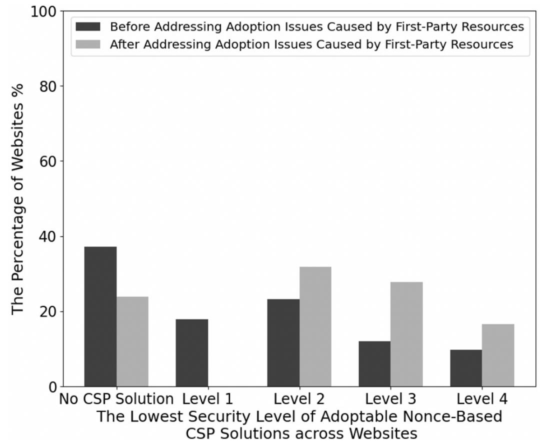


Figure 4.6 The Lowest Security Level Changes of Adoptable Nonce-Based CSP Solutions across 7,032 Websites after Addressing First-Party Resources that Can Cause Adoption Issues

In more details, among 2,610 websites that contain webpages unable to adopt any nonce-based CSP solutions, none of them would have the lowest security level of adoptable nonce-based CSP solutions at level 1, while 17.01%, 9.35%, and 9.50% of them would have the lowest security level of adoptable nonce-based CSP solutions at levels 2, 3, and 4, respectively. Among 1,256 websites that have the lowest security level of adoptable nonce-based CSP solutions at level 1, 55.33%, 36.54%, and 8.12% of them would have the lowest security level of adoptable nonce-based CSP solutions at levels 2, 3 and 4, respectively. Among 1,636 websites that have the lowest security level of adoptable nonce-based CSP solutions at level 2, 24.51% and 8.19% of them would have the lowest security level of adoptable nonce-based CSP solutions at levels 3 and 4, respectively. There are no changes for websites that can already adopt the stronger nonce-based CSP solutions since webpages that can adopt the stronger nonce-based solutions do not contain first-party resources that can cause adoption issues.

Takeaways. It is worth noting that if websites can address the adoption issues of nonce-based CSP solutions caused by first-party resources, the number of websites that can adopt the nonce-based CSP solutions will increase by 13.31%, and the number of websites that can adopt the stronger CSP solutions will increase by 22.58%.

4.4.3 Stability of the Nonce-Based CSP Solution Adoption

The analysis results of the nonce-based CSP solution adoption on websites across three crawlings are similar, while they varied slightly. The percentages of websites that can adopt the nonce-based CSP solutions in the first, second, and third crawlings are 62.88%, 64.43%, and 63.81%, respectively. The percentages of websites that can adopt the stronger nonce-based CSP solutions in the first, second, and third crawlings are 21.75%, 24.88%, and 23.07%, respectively. To answer RQ3, we investigate the impact of one site-level and two page-level changes (i.e., accessibility changes of websites and webpages as well as resource type changes of webpages) on the result variations of the nonce-based CSP solution adoption across three crawlings.

Variations Caused by Newly Appeared and Disappeared Websites. Newly appeared websites and disappeared websites can cause variations in the results of the nonce-based CSP solution adoption. We found that in the second crawling, 1,026 websites that were successfully visited in the first crawling disappeared, and 778 websites newly appeared. Among 1,026 websites that disappeared in the second crawling, 656 of them can adopt the nonce-based CSP solutions, and 242 of them can adopt the stronger nonce-based CSP solutions. Among 778 websites that newly appeared in the second crawling, 555 of them can adopt the nonce-based CSP solutions, and 252 of them can adopt the stronger nonce-based CSP solutions. Overall, in the second crawling, newly appeared websites and disappeared websites make the number of websites that can adopt the nonce-based CSP solutions increase by 0.81%, and make the number of websites that can adopt the stronger nonce-based CSP solutions increase by 0.95%.

In the third crawling, 639 websites that were successfully visited in the second crawling disappeared, and 1,081 websites newly appeared. Among 639 websites that disappeared in the third crawling, 446 of them can adopt the nonce-based CSP solutions, and 192 of them can adopt the stronger nonce-based CSP solutions. Among 1,081 websites that newly appeared in the third crawling, 690 of them can adopt the nonce-based CSP solutions, and 249 of them can adopt the stronger nonce-based CSP solutions. Overall, in the third crawling, newly appeared websites and disappeared websites make the number of websites that can adopt the nonce-based CSP solutions decrease by 0.56%, and make the number of websites that can adopt the stronger nonce-based CSP solutions decrease by 0.73%.

Variations Caused by Newly Appeared and Disappeared Webpages. Websites commonly visited but with different webpages in two crawlings can cause variations in the results of the nonce-based CSP solution adoption on websites. Specifically, websites may no longer be able to adopt the nonce-based CSP solutions if newly appeared webpages cannot adopt any nonce-based solutions, while websites may be able to adopt the nonce-based CSP solutions if webpages that cannot adopt any nonce-based solutions disappear. Similarly, websites may no longer be able to adopt the stronger nonce-based CSP solutions if newly appeared webpages cannot adopt the stronger nonce-based solutions, while websites may be able to adopt the stronger nonce-based CSP solutions if webpages that cannot adopt the stronger nonce-based solutions disappear.

We found that 4,436 websites that were commonly visited in the first and second crawlings have different webpages. Among these 4,436 websites, 4,306 of them have webpages that were successfully visited in the first crawling but disappeared in the second crawling (among which 47.02% have less than 20% webpages that disappeared), and 4,162 of them have new webpages in the second crawling (among which 49.08% have less than 20% webpages that newly appeared). Among 4,306 websites that have webpages that disappeared in the second crawling, 131 of them can adopt the nonce-based CSP solutions, and 212 of them can adopt the stronger nonce-based CSP solutions, due to their disappeared webpages. Among 4,162 websites that have newly appeared webpages in the second crawling, 78 of them can no longer adopt the nonce-based CSP solutions, and 55 of them can no longer adopt the stronger nonce-based CSP solutions, due to their newly appeared webpages. Overall, in the second crawling, newly appeared webpages and disappeared webpages make the number of websites that can adopt the nonce-based CSP solutions increase by 0.79%, and make the number of websites that can adopt the stronger nonce-based CSP solutions increase by 2.31%.

In the second and third crawlings, 4,455 websites were commonly visited but have different webpages. Among these 4,455 websites, 4,210 of them have webpages that disappeared in the third crawling (among which 47.26% have less than 20% webpages that disappeared), and 4,304 of them have newly appeared webpages in the third crawling (among which 46.12% have less than 20% webpages that newly appeared). Among 4,210 websites that have webpages which disappeared in the third crawling, 134 of them can adopt the nonce-based CSP solutions, and 129 websites can adopt the stronger nonce-based CSP solutions, due to their disappeared webpages. Among 4,304 websites that have newly appeared webpages in the third crawling, 146 of them can no longer adopt the nonce-based CSP solutions, and 216 websites can no longer adopt the stronger nonce-based CSP solutions, due to their newly appeared webpages. Overall, in the third crawling, newly appeared webpages and disappeared webpages make the number of websites that can adopt the nonce-based CSP solutions decrease by 0.17%, and make the number of websites that can adopt the stronger nonce-based CSP solutions decrease by 1.21%.

Variations Caused by Webpages that Have Changes on the Types of Used Resources. The changes in the types of used resources on webpages are another cause of variations in the results of the nonce-based CSP solution adoption. Specifically, websites may no longer be able to adopt the nonce-based CSP solutions if some webpages cannot adopt any nonce-based solutions due to their resource type changes (e.g., using event handlers inserted by third-party scripts in the next crawling), while websites may be able to adopt the nonce-based CSP solutions if some webpages can adopt the nonce-based solutions due to their resource type changes (e.g., not containing URLs in <base> tags in the next crawling). Similarly, websites may no longer be able to adopt the stronger nonce-based CSP solutions if some webpages cannot adopt the stronger nonce-based solutions due to their resource type changes, while websites may be able to adopt the stronger nonce-based CSP solutions if some webpages can adopt the stronger nonce-based solutions due to their resource type changes.

We found that 2,031 websites that were commonly visited in the first and second crawlings contain webpages that have different types of used resources, 73.81% of them have less than 20% webpages that have resource type changes. Among these 2,031 websites, 44 of them can no longer adopt the nonce-based CSP solutions in the second crawling due to webpage resource type changes, while 40 of them can adopt the nonce-based CSP solutions due to webpage resource type changes; 50 of them can no longer adopt the stronger nonce-based CSP solutions in the second crawling, while 31 of them can adopt the stronger nonce-based CSP solutions, due to webpage resource type changes. Overall, in the second crawling, webpage resource type changes make the number of websites that can adopt nonce-based CSP solutions decrease by 0.063%, and make the number of websites that can adopt the stronger nonce-based CSP solutions decrease by 0.28%.

In the second and third crawlings, 1,992 websites that were commonly visited contain webpages that have different types of used resources, and 75.25% of them have less than 20% webpages that have changes in types of used resources. Among these 1,992 websites, 38 of them can no longer adopt the nonce-based CSP solutions in the third crawling due to webpage resource type changes, while 50 of them can adopt the nonce-based CSP solutions due to webpage resource type changes; 29 of them can no longer adopt the stronger nonce-based CSP solutions in the third crawling due to webpage resource type changes, while 42 of them can adopt the stronger nonce-based CSP solutions due to webpage resource type changes. Overall, in the third crawling, webpage resource type changes make the number of websites that can adopt the nonce-based CSP solutions increase by 0.16%, and make the number of websites that can adopt the stronger nonce-based CSP solutions increase by 0.18%.

Meanwhile, we found that in the second crawling, 24 websites that can no longer adopt the nonce-based CSP solutions and 25 websites that can no longer adopt the stronger CSP solutions are due to newly appeared webpages and resource type changes on webpages; 23 websites that can adopt the nonce-based CSP solutions and 15 websites that can the stronger nonce-based CSP solutions are due to disappeared webpages and resource type changes on webpages. In the third crawling, 24 websites that can no longer adopt the nonce-based CSP solutions and 17 websites that can no longer adopt the stronger nonce-based CSP solutions are due to newly appeared webpages and resource type changes on webpages; 28 websites that can adopt the nonce-based CSP solutions and 21 websites that can adopt the stronger nonce-based CSP solutions are due to disappeared webpages and resource type changes on webpages.

Takeaways. The percentages of websites that can adopt nonce-based CSP solutions are similar across three crawlings. The appearance and disappearance of websites and webpages, as well as resource type changes on webpages, have a minor impact on the stability of adopting the nonce-based CSP solutions on websites.

4.5 Discussion

In this section, we summarize the key findings of our work, provide recommendations for better CSP deployment, and further discuss the limitations of this work and the potential future work.

4.5.1 Summary of Key Findings

We have four key findings from our study. First, most websites can adopt the nonce-based CSP solutions, and a noticeable percentage of them can even adopt the stronger nonce-based CSP solutions (Section 4.4.1.1). For websites that cannot adopt the nonce-based CSP solutions, usually the adoption is hard on around 40% of their webpages. Second, whitelisting-based CSPs deployed on webpages are usually not used for content control, and those used for content control often have security issues (Section 4.4.1.2). Most webpages that deployed whitelisting-based CSPs can adopt nonce-based CSP solutions which are simpler and more secure. Meanwhile, most websites that deployed the same whitelisting-based CSP on all their webpages can adopt the nonce-based CSP solutions. Third, both first-party resources and third-party resources can cause issues of adopting the nonce-based CSP solutions on websites (Sections 4.4.2.1 and 4.4.2.2). If websites can address the adoption issues caused by first-party resources, there will be noticeable increases in the number of websites that can adopt the nonce-based CSP solutions and those that can adopt the stronger solutions (Section 4.4.2.3). Fourth, the percentages of websites that can adopt nonce-based CSP solutions are similar across three crawlings (Section 4.4.3). The appearance and disappearance of websites and webpages, as well as resource type changes on webpages, have a minor impact on the stability of adopting the nonce-based CSP solutions on websites

4.5.2 Recommendations

Based on the key findings from our study, we provide three recommendations that may help developers adopt Google's nonce-based CSP solutions. First, for developers who deployed whitelisting-based CSPs using a "*" or an " 'unsafe-inline' " directive value, we recommend them to check the feasibility of adopting the nonce-based CSP solutions on their webpages. Second, for developers who are constructing their websites and would like to adopt nonce-based CSP solutions on their webpages, we recommend them to avoid or refactor their first-party resources (e.g., eval-like functions, parser-inserted scripts, and event handlers) that can cause the adoption issues of the nonce-based CSP solutions. Meanwhile, we recommend them to use third-party resources only if these resources will not cause an issue of adopting nonce-based CSP solutions, or if absolutely necessary. Finally, for developers of third-party services, we recommend them to avoid providing scripts (e.g., scripts inserting parser-inserted scripts into webpages, embedding event handlers into webpages, or using eval-like functions) that can cause adoption issues of the nonce-based CSP solutions on webpages. We recommend them to explicitly describe the behaviors of their provided scripts to facilitate the proper CSP deployment on webpages.

4.5.3 Limitations and Future Work

This study has two major limitations. First, our analysis only focuses on 101 webpages of each website. This may not be accurate to represent the entire website. It would be valuable for future work to visit more webpages on websites to figure out whether there are some other adoption issues for websites to adopt Google's nonce-based CSP solutions. Second, our crawling tool only emulated two simple user interactions, i.e., scrolling down a webpage to the bottom and clicking on a button (Section 4.3.1). Although that design and implementation choice was partially for the ethical considerations (Section 4.3.4), future work may emulate more user interactions to potentially trigger more dynamic resource accesses while avoiding too much interference with the normal operations of visited websites.

4.6 Summary

In this chapter, we analyzed the feasibility of adopting Google's nonce-based CSP solutions on websites. We constructed a crawling tool based on Puppeteer to directly insert four specified CSPs, which can compose Google's two stronger nonce-based CSP solutions, under the report-only mode into websites to collect corresponding CSP violations. Our analysis of the feasibility of adopting the nonce-based CSP solutions on websites is based on the CSP violations triggered under the inserted CSPs. Among 7,032 websites, 62.88% of them can adopt the nonce-based CSP solutions, and 21.76% of them can even adopt the stronger ones. For websites that cannot adopt the nonce-based CSP solutions, usually the adoption is hard on around 40% of their webpages. Meanwhile, if developers can address adoption issues caused by first-party resources, the number of websites that can adopt nonce-based CSP solutions and those that can adopt stronger ones will increase noticeably. Additionally, the percentages of websites that can adopt nonce-based CSP solutions are similar across three crawlings. Based on our key findings, we further provided recommendations to help websites adopt Google's nonce-based CSP solutions and improve their deployed CSPs. We will share our constructed crawling tool and dataset with other researchers for their potential future studies.

CHAPTER 5

WEBMEA: A GOOGLE CHROME EXTENSION FOR WEB SECURITY AND PRIVACY MEASUREMENT STUDIES

Web measurement is an integral part of web security and privacy analysis. A large-scale web measurement usually relies on a crawler. Browser-based crawlers are often preferred and they are typically implemented through one of two approaches: the DevTools approach or the browser extension approach. It is often not clear to researchers which approach should be taken to implement a crawler since the analysis of these two approaches is limited. In this chapter, we analyze and compare the web measurement capability of the Google Chrome DevTools approach and the Google Chrome extension approach to help researchers make decisions. We find that both approaches have the capability to satisfy primary web measurement requirements, although their APIs and the complexity of using their APIs are different. We point out such differences to help researchers more easily choose or take the appropriate approach. Taking the former approach, Puppeteer is a crawler that can be readily used by researchers. However, it is difficult for researchers to find a good baseline Google Chrome extension to start with their extension-based crawler construction. Therefore, we further build WebMea as a baseline Google Chrome extension that can measure multiple types of web data. Researchers can use or customize WebMea according to their web measurement requirements. WebMea is applicable to different web measurement studies, and it can help make web measurement studies more reproducible and replicable. [4]

5.1 Introduction

Many large-scale web security and privacy measurement studies rely on crawlers for data collection. Some researchers use or customize existing crawlers, while others implement their own crawlers. Properly building, choosing, and configuring crawlers can help improve the reproducibility and replicability of web measurement studies; meanwhile, browser-based crawlers are often preferred due to their better integration with browsers and correspondingly their advanced measurement capabilities [49–51].

Browser-based crawlers are typically implemented through one of two approaches: the DevTools approach or the browser extension approach. DevTools is a set of developer tools provided by a given browser for web testing and debugging, and it is often wrapped as a set of DevTools protocol APIs [56, 57]. In the DevTools approach, researchers use DevTools protocol APIs to implement a crawler. A browser extension is a software program for extending the browser's functionalities [58, 59]. In the browser

[4] This is a joint work with Joshua Josey, and Chuan Yue [7]. Joshua is an undergraduate research assistant whose contribution is on the analysis of DevTools. Chuan is an associate professor of computer science at the Colorado School of Mines.

extension approach, researchers implement a crawler (which is a browser extension) by using extension APIs and providing other required files such as the manifest and resource files. Because of their different features, these two approaches are applicable to different scenarios. However, it is often not clear to researchers which is the more appropriate approach to constructing a crawler since the analysis of these two approaches is limited.

In this chapter, we study these two approaches in the context of the Google Chrome browser because of its popularity. We first outline the primary requirements of the crawlers for web security and privacy measurement studies. We then analyze and compare the web measurement capability of these two Google Chrome browser-based approaches. A good crawler implementation approach should have the capability to satisfy all primary web measurement requirements. We find that the Google Chrome DevTools approach and the Google Chrome extension approach rely on Chrome DevTools Protocol (CDP) APIs [57] and Chrome extension APIs [59] respectively to implement a crawler, and the complexities of using their APIs are different. However, both approaches have the capability to satisfy all primary web measurement requirements. To help researchers take the appropriate approach, we further point out the advantages, limitations, and applicable scenarios of both approaches.

Meanwhile, taking the Google Chrome DevTools approach, researchers can easily find a baseline Puppeteer crawler which is one popular browser automation tool controlling Google Chrome through CDP APIs [60]. However, there is no good baseline Google Chrome extension for researchers to start with their extension-based crawler construction because of the complexity of using the Google Chrome extension APIs. Therefore, we further build WebMea, a Google Chrome extension that can measure both web content and HTTP(S) traffic, as a baseline Google Chrome extension for researchers to construct their extension-based crawlers. Researchers can use or customize WebMea according to their web measurement requirements. Our contributions in this chapter are as follows:

- We compare the web measurement capability of two Google Chrome browser-based crawler implementation approaches to help researchers more easily choose or take the appropriate approach.
- We build WebMea as a baseline Google Chrome extension for researchers to construct their extension-based crawlers. WebMea can be used for different web measurement studies because it can measure multiple types of web data, which can help make web security and privacy measurement studies more reproducible and replicable.

The rest of this chapter is organized as follows. Section 5.2 reviews the related work. Section 5.3 presents the primary web measurement requirements, the overview of the Google Chrome DevTools approach and the Google Chrome extension approach, and the comparison of the web measurement capability of these two approaches. Section 5.4 describes the design, implementation, and customization of WebMea. Section 5.5 concludes this chapter.

5.2 Related Work

Browser-based crawlers are popularly used in web security and privacy measurement studies because of their better integration with browsers and correspondingly their advanced measurement capabilities [49–51]. Some browser-based crawlers are built through the DevTools approach. Taking this approach, Puppeteer is a crawler that can be readily used by researchers [61, 62]. Din et al. implemented PERCIVAL, a browser-embedded ad blocker for the Chromium-based browsers, in which Puppeteer was used for collecting ad and non-ad images of webpages [61]. Smith et al. constructed SugarCoat, a system for automatically generating privacy-preserving JavaScript (JS) replacements, in which Puppeteer was used for extracting JS scripts [62]. Some browser-based crawlers are implemented through the browser extension approach. Englehardt et al. constructed OpenWPM, a web privacy measurement framework for collecting cookies, response metadata, and behavior of scripts, in which data collection is performed by a Firefox extension [63]. Roesner et al. implemented TrackingObserver, a Google Chrome extension that collects trackers' in-browser behaviors to perform third-party tracking analysis [64]. However, both browser extensions are more applicable for web privacy measurement studies, and they also have limited capabilities on extracting web content and simulating user interactions.

The reproducibility and the replicability of web security and privacy measurement studies are often impacted by experimental setups including crawlers and crawler configurations. Ahmad et al. conducted comparative empirical crawler evaluations to understand how the choice of crawlers impacts the web measurement studies [49]. They found that the characteristics of network traffic and application data measured by different crawlers are not the same, which impacts web measurement results and conclusions. Demir et al. researched the reproducibility and replicability of current web measurement studies, and performed a large-scale web measurement study to show that different experimental settings (e.g., browsers, regions, and user interactions) impact the overall results of web measurement studies [50]. They also found that experimental setups were insufficiently documented in many analyzed web measurement studies. Jueckstock et al. explored how configurations of crawlers affect the web security and privacy measurement studies [51]. They observed that under different vantage points and browser configurations of crawlers, measurement bias exists in the HTTP traffic, known ad/tracking domains, and JS context. Therefore, researchers should ensure that their crawlers match the real-world user experience and can be reproduced or customized.

Overall, existing studies did not analyze or compare the two major approaches for browser-based crawler construction; therefore, we provide such an analysis to help researchers (especially beginning researchers) more easily choose or take the appropriate approach. Meanwhile, there is no good baseline Google Chrome extension for researchers to start with their extension-based crawler construction; therefore, we build WebMea which can measure multiple types of data for different studies and can simulate user interactions for matching the real-world user experience. Researchers can use or customize WebMea according to their web measurement requirements. We also expect that WebMea can help make web security and privacy measurement studies more reproducible and replicable.

5.3 Comparing the Two Crawler Implementation Approaches

A good crawler implementation approach should have the capability to satisfy all primary web measurement requirements. In this section, we first outline these primary requirements. We then review, analyze, and compare the Google Chrome DevTools approach and the Google Chrome extension approach. To further illustrate the differences between these two approaches, we also provide case studies for each approach in this section.

5.3.1 Primary Requirements for Crawlers

We outline the primary requirements of crawlers for web security and privacy measurement studies as follows.

Automated Crawling. Crawlers should be able to automatically visit webpages in a particular sequence according to a supplied URL list, and can further identify and visit a specific number of subpages (with the same domain or subdomain) of the visited webpages. Automated crawling is an essential requirement for large-scale web security and privacy analysis.

HTTP(S) Traffic Interception and Modification. Based on the need of the studies, crawlers should be able to control HTTP(S) requests and responses. Specifically, crawlers should be able to intercept requests, add requests, redirect requests, remove requests, and modify request body content and request headers. Crawlers should also be able to intercept responses and control them such as modifying response body content and response headers. For example, content security policy analysis and web tracking analysis usually need crawlers to measure the HTTP(S) traffic of crawled webpages.

Webpage Content Extraction and Modification. Crawlers should have the capabilities of extracting web content, adding web elements to a webpage, removing web elements from a webpage, and modifying attributes of web elements. For example, web attack analyses (e.g., phishing detection) often require crawlers to extract or modify web content.

Simulation of User Interactions. Crawlers should be able to simulate user interactions such as clicking on links or scrolling down the webpages. The simulation of user interactions is an important part of setting up an experimental environment that can match the real-world user experience.

5.3.2 Google Chrome DevTools and Extension Approaches

The DevTools approach and the browser extension approach are based on different designs. DevTools is a set of developer tools provided by a given browser for creating, testing, or debugging websites or web applications [46, 65]. DevTools can either be built in a browser or used through the DevTools protocol APIs. Google Chrome provides both the built-in Chrome DevTools and the Chrome DevTools Protocol (CDP) APIs [57] for developers. However, the built-in Chrome DevTools is not often applied to large-scale web measurement studies since it is built for in-browser webpage debugging. In the Google Chrome DevTools approach, researchers use the CDP APIs to construct their crawlers. Google Chrome divides the CDP APIs into 46 domains (e.g., DOM domain, Page domain, and Network domain). Each domain includes a number of APIs and events it supports and generates. Puppeteer is a good baseline crawler that takes this approach for controlling Google Chrome in web security and privacy measurement studies [61, 62].

Extensions are software programs for extending browsers' functionalities. Google Chrome extensions are event-based programs that rely on three major web techniques: HTML, CSS, and JS [59]. Google Chrome provides 76 extension APIs to help extensions control the browser and interact with crawled webpages. In the Google Chrome extension approach, researchers construct an extension as a crawler by using the Google Chrome extension APIs and providing other required files. A Google Chrome extension can be composed of the manifest file, background scripts, content scripts, an options page, and UI elements. The manifest file is used for configuring extension settings and permissions. The background scripts can monitor events triggered by the browser, and Google Chrome extensions interact with the browser through the background scripts. The content scripts can run on crawled webpages, and Google Chrome extensions interact with webpages through the content scripts. The options page allows users to adjust extension options. The UI elements help users interact with Google Chrome extensions.

The capability of the Google Chrome DevTools approach and the Google Chrome extension approach depends on the capability of the CDP APIs and the extension APIs, respectively. We draw Figure 5.1 to summarize the major categories of capabilities of these two types of APIs. These two types of APIs have some common capabilities but also have some different capabilities. They both have the capabilities of configuring browser settings, interacting with webpages, controlling browser actions, controlling web content, controlling HTTP(S) traffic, auditing webpages, and emulating different page environments. Other major capabilities of the CDP APIs are controlling media, controlling background services, and measuring web performance. Other major capabilities of the Google Chrome extension APIs are implementing text-to-speech (TTS) engines, extending DevTools panels, managing Chrome OS, and managing extensions and web applications installed or running in the browser. Although the CDP APIs and the Google Chrome extension APIs focus on web debugging and browser function extension respectively, both of them have the capabilities of interacting with webpages and browsers. Therefore, both approaches can be used to construct crawlers. Note that Google Chrome extensions can also access 15 CDP API domains (e.g., those for webpage interaction, web content control, and HTTP(S) traffic control) through the *chrome.debugger* extension API. They cannot access other CDP API domains for security reasons.

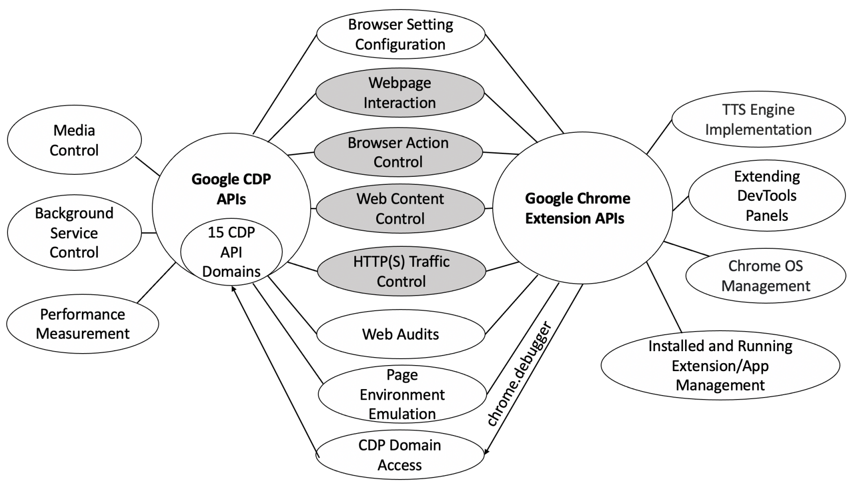


Figure 5.1 Major Categories of Capabilities of the Chrome DevTools Protocol (CDP) APIs and the Google Chrome extension APIs. Google Chrome extensions can also access 15 CDP API domains through the *chrome.debugger* extension API.

A good crawler implementation approach should have the capability to satisfy all primary web measurement requirements outlined in Section 5.3.1. In other words, it should be capable of controlling browser actions to achieve automated crawling, controlling web content to achieve webpage content extraction and modification, controlling HTTP(S) traffic to achieve HTTP(S) traffic interception and modification, and interacting with webpages to achieve simulations of user interactions. As Figure 5.1 shows, the CDP APIs and the Google Chrome extension APIs both have these four categories of web measurement capabilities to satisfy the primary web measurement requirements.

Although the CDP APIs and the Google Chrome extension APIs commonly have the major web measurement capabilities, their granularities of controlling webpages and browsers are different. Therefore, these two approaches have different advantages and are sometimes applicable to different scenarios. With the CDP APIs, the Google Chrome DevTools approach can simulate page environments at a more fine-grained level than the Google Chrome extension approach. For example, the CDP LayerTree domain provides APIs for analyzing and operating on layers, and the CDP DeviceOrientation domain provides APIs for clearing and overriding device orientations. However, these two CDP domains are not accessible to Google Chrome extensions. Researchers can give priority to constructing a crawler through the Google Chrome DevTools approach when page environments need to be very close to the real one. On the other hand, the Google Chrome extension approach can control HTTP(S) requests at a more fine-grained level through the webRequest extension API. This API can monitor a set of events that follow the entire life cycle of an HTTP(S) request. Researchers can give priority to implementing a crawler through the Google Chrome extension approach when their crawlers are required to work closely on requests and responses following the life cycle of requests.

To help researchers meet the primary web measurement requirements in their crawlers, we further summarize the corresponding major APIs of the two approaches in Table 5.1. The Google Chrome DevTools approach mainly relies on Page domain CDP API and Network domain CDP API to achieve the primary web measurement capabilities, while the Google Chrome extension approach mainly relies on webRequest extension API, tabs extension API, runtime extension API, and content scripts. To further illustrate the differences between these two approaches, we provide a case study for each of them in Sections 5.3.3 and 5.3.4, respectively. Sometimes crawler implementation by taking the Google Chrome extension approach is more complex than taking the Google Chrome DevTools approach since the former needs multiple required files to work together to achieve a web measurement function. Therefore, it will be helpful to have a good baseline Google Chrome extension for researchers to start with their extension-based crawler construction (Section 5.4).

Table 5.1 The major CDP APIs and Google Chrome extension APIs of the two approaches for satisfying the primary web measurement requirements.

Requirement	***Operation***	***DevTools Approach***	***Extension Approach***
Automated Crawling	Accessing webpages	Page domain CDP API	tabs extension API
	Webpage link extraction	Page domain CDP API	Content scripts and runtime extension API
HTTP(S) Traffic Interception and Modification	Request interception and modification	Network domain CDP API	webRequest extension API
	Response interception and response header modification	Network domain CDP API	webRequest extension API
	Response body modification	Network domain CDP API	Network domain CDP API and debugger extension API
Webpage Content Extraction and Modification	Webpage content extraction	Page domain CDP API	Content scripts and runtime extension API
	Webpage content modification	Page domain CDP API	Content scripts and runtime extension API
Simulation of User Interactions	Simulating user experience in the real-word	Page domain, Runtime domain, and Emulation domain CDP APIs	Content scripts and runtime extension API

5.3.3 DevTools Case Study: Puppeteer

Puppeteer is a Node.js library that provides a high-level API to control headless or real Chrome and Chromium browsers over the CDP APIs [60]. The high-level API contains multiple classes and corresponding class methods to help users easily implement the functions of a crawler. Users can create an instance of a browser and then manipulate it with Puppeteer's class methods which can create CDP sessions for using the CDP APIs. Puppeteer is composed of JS files, declaration files, and source map files. The JS files and declaration files are used for class declarations. The methods of a class are in the corresponding JS file, and the method parameters are in the corresponding declaration file. Source map files are provided for each class to make JS debugging and testing efficient. Because Puppeteer is CDP-based, its capability depends on the capability of the CDP APIs. Therefore, Puppeteer has web testing and debugging capabilities that include navigating webpages, collecting screenshots, automating form submissions, and extracting or modifying data from the DOM.

Puppeteer's high-level interaction with the browser as well as low-level control over data collection make it a good baseline crawler, upon which many other Google Chrome DevTools-based crawlers are built. The crawlers used in the studies of Din et al. [61] and Smith et al. [62] were built upon Puppeteer as we reviewed in Section 5.2. They only used the basic features of Puppeteer without simulating user interactions; meanwhile, they focused on web content extraction and had a limited capability of HTTP(S) traffic measurement. In addition, there is a Google Chrome headless-chrome-crawler [66] built on Puppeteer for crawling dynamic websites, and researchers can customize this crawler through the Puppeteer high-level API and the CDP APIs.

5.3.4 Chrome Extension Case Study: TrackingObserver

The Tracking-Observer Google Chrome extension [64] we reviewed in Section 5.2 has the following primary web measurement functions: (1) crawling a list of websites in an arbitrary link depth, (2) tracking and resetting cookies, and (3) tracking history. All these functions are implemented in a background script and a content script. For crawling a website in an arbitrary link depth, the background script sends a request to the content script to extract a specific number of links from crawled webpages. The content script listens to requests sent from the background script and sends the extracted links back to the background script through the *chrome.extension* API. TrackingObserver tracks cookies by filtering cookie items from HTTP(S) request headers through the webRequest extension API. It can also cancel and remove original HTTP(S) requests through the webRequest extension API. TrackingObserver resets cookies through the *set()* method of the cookies extension API. For tacking history, the *chrome.history.search()* extension method is used for obtaining all history of the last visit time.

TrackingObserver takes the advantages of the webRequest extension API to process HTTP(S) requests.

TrackingObserver is more applicable for web privacy measurement studies, and it also has limited capabilities on extracting web content and simulating user interactions. Therefore, using it as a baseline Google Chrome extension for other web security and privacy measurement studies will still need lots of development effort from researchers.

5.4 Proposed Chrome Extension WebMea

In this section, we present the design, implementation, and customization of our proposed Google Chrome extension WebMea.

5.4.1 Design of WebMea

WebMea can collect HTTP(S) requests and corresponding responses, HTML documents, and http-equiv attributes of meta elements from crawled webpages. It can be used in different web security and privacy measurement studies. Figure 5.2 shows the overview of WebMea. It mainly comprises a background script, four content scripts (i.e., subpage_extract.js, dom_extract.js, meta_extract.js, and page_scroll.js files), and a popup.js file. The background script implements functions that need to interact with the browser. Content scripts implement functions that need to interact with crawled webpages. The popup.js file provides UI for users to interact with WebMea. Researchers can start crawling by clicking the "Start Crawling" button on the popup page. Researchers can save the collected web data either in an in-browser database maintained by Google Chrome or in a remote database supported by Node.js. By default, WebMea saves all collected data in the in-browser database. WebMea provides multiple functions listed as follows to satisfy the primary web measurement requirements.

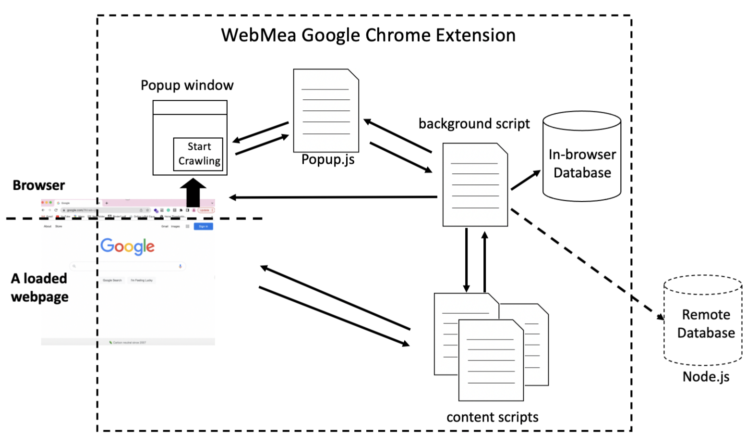


Figure 5.2 The High-level Overview of WebMea

Automated Crawling. Given a crawling website list, WebMea will crawl the homepage of each website and a specific number of randomly selected subpages (with the same domain or subdomain) of each homepage. WebMea will stay for a certain period of time on each crawled webpage for the page rendering and event handling.

HTTP(S) Traffic Interception and Modification. When the browser receives HTTP(S) response headers of webpages, WebMea will intercept them. Researchers can extract content security policies [19] deployed in a webpage from its HTTP(S) response headers. WebMea will also intercept all successfully processed HTTP(S) requests and corresponding responses in the process of loading and rendering a webpage. The types of intercepted HTTP(S) requests include "main_frame", "sub_frame", "stylesheet", "script", "image", "font", "object", "XMLHttpRequest", "ping", "csp_report", "media", "websocket", and "other". The intercepted HTTP(S) requests and corresponding responses can be used for various security and privacy analyses such as malicious script analysis and insecure communication analysis.

Webpage Content Extraction and Modification. WebMea will extract the HTML document of a webpage when the webpage is completely loaded. Researchers can further process the extracted web content for various security and privacy analyses such as phishing attack analysis and malicious web element analysis. Some content security policies are also deployed through the meta http-equiv attributes thus WebMea also extracts such attributes of webpages.

Simulation of User Interactions. WebMea will scroll down each crawled webpage to the bottom for simulating user interactions on viewing more content of a webpage. WebMea can simulate other user interactions such as link clicking and form filling. It can also clear the browsing history, cached data, and cookies after crawling a webpage to simulate stateless browsing scenarios.

5.4.2 Implementation of WebMea

WeaMea is configured through a Chrome Manifest V2 file, and it saves web data into an in-browser SQLite database. WebMea relies on the background script and content scripts to implement web measurement functions. Specifically, the background script contains the functions of crawling a webpage for a certain period of time, intercepting successfully processed HTTP(S) requests and corresponding responses, saving collected data in the in-browser database, and clearing browsing data. Content scripts contain functions of extracting subpage links from a homepage, scrolling down a webpage to the bottom, extracting HTML documents, extracting http-equiv attributes of meta elements, and so on.

To implement web measurement functions, sometimes the content scripts need to communicate with the background script, and the background script needs to insert and execute the content scripts in a crawled webpage. The content scripts communicate with the background script by sending requests to it through the *sendMessage()* method of the runtime extension API. The background script receives requests by monitoring the *onMessage()* event of the runtime extension API. The background script inserts and executes content scripts in a webpage by using the *executeScript()* method of the tabs extension API. We detail the implementation of these web measurement functions as follows.

Automated Crawling. By default, WebMea automatically crawls the homepage and a specific number of subpages for each website. The automated crawling is implemented in the background script through the *onUpdated.addListener()*, *update()*, and *query()* methods of the tabs extension API. The *update()* method updates the URL of the current tab. The *onUpdated.addListener()* method monitors the webpage updates in the current tab. The *query()* method can get the information of the webpage in the current tab. When the *onUpdated.addListener()* method observes the webpage update in a tab, the corresponding callback function will be executed to further process the current webpage. The callback function of the *onUpdated.addListener()* method checks whether the current webpage is a homepage. If the webpage is a homepage, the callback function will execute the content script subpage_extract.js which will extract and send back the subpage links of the homepage. Then the callback function will receive and add subpage links to the crawling list. WebMea uses the JS setInterval() method to control the time staying on a webpage.

HTTP(S) Traffic Interception and Modification. By default, WebMea can intercept HTTP(S) response headers before responses are processed, and it can intercept successfully processed HTTP(S) requests and corresponding responses after webpages are completely loaded. These functions are implemented in the background script. The interception of HTTP(S) response headers is implemented by monitoring the *onHeadersReceived()* event of the webRequest extension API. The interception of processed HTTP(S) requests and corresponding responses is implemented through monitoring the *onCompleted()* event of the webRequest extension API. For *onHeadersReceived()* and *onCompleted* events, returned data (i.e., HTTP(S) response headers and HTTP(S) requests) are saved in the corresponding "details" parameter of each event. The "filter" parameter is provided in both events for filtering intercepted HTTP(S) response headers and requests.

Webpage Content Extraction and Modification. By default, WebMea can extract HTML documents and http-equiv attributes of meta elements from crawled webpages. The web content is extracted by inserting and executing content scripts in the crawled webpage and further processed by the background script. The HTML documents are extracted through the JS *XMLSerializer()* method in the content script dom_extract.js. The meta http-equiv attributes are extracted through the *getAttribute()* method of the HTML Element class in the content script meta_extract.js. The dom_extract.js and meta_extract.js scripts wrap the extracted HTML documents and meta http-equiv attributes in requests and send the requests to the background script. The background script receives the requests and will further process the received web content.

Simulation of User Interactions. By default, WebMea scrolls down a webpage to the bottom after the webpage is completely loaded. The *scrollingElement* property of the HTML document object implements the scrolling down function in the content script page_scroll.js. Then the background script inserts and executes the page_scroll.js script in the crawled pages. After crawling a webpage, WebMea can clear the browsing history, cached data, and cookies through the remove() method of the browsingData extension API.

5.4.3 Customization of WebMea

Researchers can customize WebMea based on their web measurement requirements. Some customization examples are as follows.

Customizing Automated Crawling. Researchers can specify their desired number of crawled subpages in the subpage_extract.js script. They can change the crawling sequence of webpages by modifying the callback function of the *onUpdated.addListener()* method of the tabs extension API in the background script.

Customizing HTTP(S) Traffic Interception and Modification. Researchers can cancel HTTP(S) requests by monitoring the *onBeforeRequest* event, the *onAuthRequired* event, or the *onHeadersReceived* event of the webRequest API. They can redirect an HTTP(S) request by monitoring the *onBeforeRequest()* event and the *onHeadersReceived()* event of the webRequest API. They can intercept, add, modify, and delete HTTP(S) request headers by monitoring the *onBeforeSendHeaders()* event of the webRequest API. They can modify HTTP(S) response headers by monitoring the *onHeadersReceived()* event of the webRequest API. They can intercept failed HTTP(S) requests by monitoring the *onErrorOccurred()* event of the webRequest API. They can modify HTTP(S) response body content through the Network domain CDP API which is accessible by using the debugger extension API.

Customizing Webpage Content Extraction and Modification. There are two ways to modify the content of a webpage. Researchers can modify webpage content through the Page domain CDP API which is accessible by using the debugger extension API. They can also modify webpage content by using content scripts, in which the methods of the HTML document object such as *createElement()* can be used to modify web content; meanwhile, they need to let the background script insert and execute the content scripts in the webpage.

Customizing Simulation of User Interactions. WebMea implements scrolling down webpages by using content scripts. Researchers can implement other user interactions in the same way. For example, to simulate clicking on a webpage element such as a link or a button, researchers can implement a function to trigger a click on the identified webpage element in a content script, and then let the background script insert and execute the content script in the webpage.

Customizing Dataset Connection. Researchers can save web data to remote databases. One way is to save web data to a remote database supported by a Node.js web server. At the server side, researchers need to create a database (e.g., a MongoDB database in our WebMea implementation), use the *connect()* method of Node.js to connect the web server to the database, and process the web data contained in the HTTP(S) requests that are sent from WebMea at the browser side. Based on their preferences, researchers can use other types of database systems instead of MongoDB, and can use other types of web servers instead of relying on Node.js.

5.5 Summary

In this chapter, we studied two browser-based crawler implementation approaches in the context of the Google Chrome browser: the DevTools approach and the extension approach. We found that both approaches have the capability to satisfy primary web measurement requirements, although their APIs and the complexity of using their APIs are different. The DevTools approach is more appropriate in scenarios where page environments need to be very close to the real one. The extension approach is more appropriate in scenarios where crawlers are required to work closely on HTTP(S) requests and responses following the life cycle of requests. We expect the analysis and comparison of these two approaches can help researchers more easily choose or take the appropriate approach to construct their browser-based crawlers. Meanwhile, we built and presented WebMea as a baseline Google Chrome extension for researchers to easily start with their extension-based crawler construction. WebMea can be used or customized in different web security and privacy measurement studies, and can help make those studies more reproducible and replicable. We have used WebMea in [1] and another project for HTTP(S) traffic, webpage content, and JavaScript measurement. We provide the GitHub link to WebMea at [67].

CHAPTER 6
CONCLUSION

In previous chapters, we have introduced this dissertation and detailed its main contributions in investigating and promoting proper CSP deployment. In this final chapter, we make a conclusion and outline future work.

6.1 Conclusion

CSP is a standardized leading technique for protecting webpages against attacks such as XSS, and all major web browsers have successfully adopted it. However, it is often hard to properly deploy CSPs on webpages, and the deployed CSPs often contain security issues or errors. This dissertation aims to investigate and promote the proper deployment of CSPs. Moreover, our work answers three important research questions: Can deployed CSPs protect websites effectively? What CSP deployment issues would websites encounter? How can web developers properly deploy CSPs on their websites? Specifically, for the first question, we investigated the security levels of the deployed CSPs from the directive coverage and secure use perspectives via clustering (Chapter 2). For the second question, we analyzed potential CSP deployment issues related to third-party scripts among different websites by conducting a measurement study (Chapter 3). For the third question, we explored the feasibility of taking Google's secure CSP approach on websites by conducting a measurement study (Chapter 4). Meanwhile, we studied two browser-based crawler implementation approaches in the context of the Google Chrome browser and further built a Google Chrome extension named WebMea as a baseline Google Chrome extension that can measure multiple types of web data (Chapter 5).

Overall, our contributions come from both an analysis methodology perspective and a results perspective. From the analysis methodology perspective, we proposed to take the clustering approach for analyzing the security levels of the deployed CSPs from the directive coverage and secure use perspectives, and we proposed to analyze the CSP deployment issues and the feasibility of adopting CSP solutions directly based on CSP violations triggered under specific inserted CSPs in the report-only mode. Additionally, we built a Google Chrome extension named WebMea as a baseline Google Chrome extension that can measure multiple types of web data. From the results perspective, we found that most deployed CSPs do not sufficiently protect webpages, and clustering helps identify the corresponding common or different reasons from the directive coverage and secure use perspectives. Meanwhile, most websites would not encounter CSP deployment issues. However, compared to other websites, builder-generated websites and CDN-enabled websites are more likely to contain third-party scripts that can lead to CSP deployment issues, and the choice of web builders and CDNs can impact the CSP deployment on the corresponding websites. Furthermore, we found that a significant proportion of websites can take Google's secure approach, and most whitelisting-based CSPs deployed on websites can be converted to nonce-based ones.

6.2 Future Work

Based on our work, we have identified the following opportunities for future work. First, we focused on analyzing potential CSP deployment issues related to third-party scripts on homepages of different websites (Chapter 3), it would still be valuable for future work to visit more webpages on websites to figure out whether there are some other issues or challenges for developers to properly deploy CSPs on their websites. Second, we found that most analyzed websites can adopt Google's nonce-based CSP solutions (Chapter 5). It is valuable to explore the automated nonce-based CSP generation to promote the nonce-based CSP deployment.

REFERENCES

[1] Mengxia Ren and Chuan Yue. Coverage and secure use analysis of content security policies via clustering. In *Proceedings of IEEE European Symposium on Security and Privacy (EuroS&P)*, 2023.

[2] 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.

[3] 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.

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

[5] Mengxia Ren and Chuan Yue. Analysis of third-party scripts related content security policy deployment issues. In *Submission to a Conference*, 2024.

[6] Mengxia Ren, Anhao Xiang, and Chuan Yue. Analyzing the feasibility of taking google's secure csp approach on websites. In *Submission to a Conference*, 2024.

[7] Mengxia Ren, Joshua Josey, and Chuan Yue. Webmea: A google chrome extension for web security and privacy measurement studies. In *Proceedings of International Conference on Science of Cyber Security (SciSec)*, 2023.

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

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

[10] 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.

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

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[31] 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.

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

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

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

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

[36] 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.

[37] 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.

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

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

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

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

[42] Webflow-Site. Webflow Website Builder, 2022. https://webflow.com.

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

[44] Data-Code-Github. Datasets and Code of The Chapter 2, 2023. https://github.com/mengxiaren2024/CSP_via_Clustering.git.

[45] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. 2019.

[46] Google-Chrome-Devtools. Google Chrome Devtools, 2023. https://chromedevtools.github.io/devtools-protocol/tot/Network/.

[47] Wappalyzer. Wappalyzer, 2023. https://www.wappalyzer.com/.

[48] webXray-Domain-Owner-List. webXray Domain Owner List, 2023. https://web.archive.org/web/20200604213008/https://github.com/timlib/webXray_Domain_Owner_List/blob/master/domain_owners.json.

[49] Syed Suleman Ahmad, Muhammad Daniyal Dar, Muhammad Fareed Zaffar, Narseo Vallina-Rodriguez, and Rishab Nithyanand. Apophanies or epiphanies? how crawlers impact our understanding of the web. In *Proceedings of the ACM Web Conference (WWW)*, 2020.

[50] Nurullah Demir, Matteo Große-Kampmann, Tobias Urban, Christian Wressnegger, Thorsten Holz, and Norbert Pohlmann. Reproducibility and replicability of web measurement studies. In *Proceedings of the ACM Web Conference (WWW)*, 2022.

[51] Jordan Jueckstock, Shaown Sarker, Peter Snyder, Aidan Beggs, Panagiotis Papadopoulos, Matteo Varvello, Benjamin Livshits, and Alexandros Kapravelos. Towards realistic and reproducible web crawl measurements. In *Proceedings of the ACM Web Conference (WWW)*, 2021.

[52] HubSpot-CSP. SSL and Domain Security in HubSpot, 2023. https://knowledge.hubspot.com/domains-and-urls/ssl-and-domain-security-in-hubspot.

[53] Lukas Weichselbaum. Mitigate cross-site scripting (XSS) with a strict Content Security Policy. 2024. https://web.dev/articles/strict-csp.

[54] Lukas Weichselbaum and Michele Spagnuolo. Content Security Policy- A Successful Mess between Hardening and Mitigation. 2024. https://tinyurl.com/yyohn6o6.

[55] Florian Hantke, Stefano Calzavara, Moritz Wilhelm, Alvise Rabitti, and Ben Stock. You call this archaeology? evaluating web archives for reproducible web security measurements. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.

[56] Firefox-Remote-Protocol. Firefox Remote Protocol, 2023. https://wiki.mozilla.org/WebDriver/RemoteProtocol.

[57] Google-Chrome-DevTools-Protocol. Google Chrome DevTools Protocol, 2023. https://chromedevtools.github.io/devtools-protocol/.

[58] Firefox-Extension. Firefox Extension, 2023. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions.

[59] Google-Chrome-Extensions. Google Chrome Extension, 2023. https://developer.chrome.com/docs/extensions/.

[60] Google-Chrome-Puppeteer. Google Chrome Puppeteer, 2023. https://developer.chrome.com/docs/puppeteer/.

[61] Zainul Abi Din, Panagiotis Tigas, Samuel T King, and Benjamin Livshits. Percival: Making in-browser perceptual ad blocking practical with deep learning. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2020.

[62] Michael Smith, Pete Snyder, Benjamin Livshits, and Deian Stefan. Sugarcoat: Programmatically generating privacy-preserving, web-compatible resource replacements for content blocking. In *Proceedings of the ACM CCS*, 2021.

[63] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[64] TrackingObserver. TrackingObserver: A Browser-Based Web Tracking Detection Platform, 2023. https://trackingobserver.cs.washington.edu/.

[65] Firefox-DevTools. Firefox DevTools, 2023. https://firefox-source-docs.mozilla.org/devtools-user/.

[66] Headless-Chrome-Crawler. Headless Chrome Crawler, 2023. https://github.com/yujiosaka/headless-chrome-crawler.

[67] WebMea. WebMea, 2023. https://github.com/mengxiaren2024/WebMea.git.

[68] 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.

APPENDIX A

APPENDICES OF CHAPTER 2

A.1 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 [68] while the learning rate is 1×10^{-4}, which are used to optimize the encoder network f and the projection head g. Table A.1 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 A.1 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

A.2 The 358 Ranked Policy Features

Table A.2 lists all the 358 ranked policy features which are used for clustering the CSPs from the 13,317 homepages.

Table A.2 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)				

A.3 Some Website and CSP Examples

A.3.1 Example Websites Built through Web Development Platforms with Identical CSPs

We analyzed in Section 2.4.3.5 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;"*.

A.3.2 Example Websites with Severe Problems in Their CSPs

We analyzed in Section 2.4.3.6 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.

A.3.3 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 A.1) 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 A.1 The CSP Deployed on *https://www.xrptoolkit.com/*

APPENDIX B

APPENDICES OF CHAPTER 3

B.1 The 64 Web Builders That Have a Minimum of 10 Websites Built on Them

Table B.1 lists all the 64 web builders that have a minimum of 10 websites built on them in the first crawling.

Table B.1 Details of the 64 web builders that have a minimum of 10 websites built on them. The "*" indicates web builders with websites built on them having a tendency to deploy CSPs with an "'unsafe-inline'" directive value. The "Δ" indicates web builders with websites built on them having a tendency to deploy CSPs with an "'unsafe-eval'" directive value.

Index	**Web Builder Name**	**Index**	**Web Builder Name**
1	1C-Bitrix	2	*Adobe Experience Manager
3	ΔArc XP	4	Optimizely Content Management
5	*ΔBrightspot	6	HCL Digital Experience
7	DNN	8	Microsoft SharePoint
9	Craft CMS	10	Concrete CMS
11	Drupal	12	DataLife Engine
13	Gnuboard	14	ExpressionEngine
15	*ΔChorus	16	HubSpot CMS Hub
17	Joomla	18	Kentico CMS
19	Liferay	20	MODX
21	Contentful	22	October CMS
23	Omni CMS	24	OpenCms
25	Bloomreach	26	Pimcore
27	Plone	28	Prismic
29	ΔRebelMouse	30	SDL Tridion
31	SPIP	32	Sanity
33	Sitecore	34	Sitefinity
35	ΔSquarespace	36	Squiz Matrix
37	TYPO3 CMS	38	Tilda
39	*ΔTownNews	40	Umbraco
41	WHMCS	42	Wagtail
43	ΔWix	44	WordPress
45	eZ Publish	46	ΔContentstack
47	CakePHP	48	CodeIgniter
49	Django	50	Express
51	Java Servlet	52	JavaServer Faces
53	Laravel	54	JavaServer Pages
55	Livewire	56	Microsoft ASP.NET
57	Marko	58	Nette Framework
59	Next.js	60	Nuxt.js
61	Spring	62	Ruby on Rails
63	ThinkPHP	64	Yii

B.2 The 12 Web Builders That Host and Supply Third-party Scripts

Table B.2 lists all the 12 web builders that have a minimum of 10 websites built on them in the first crawling, and they supply third-party scripts to more than 50% of the websites built on them.

Table B.2 Details of the 12 web builders that have a minimum of 10 websites built on them in the first crawling, and they supply third-party scripts to more than 50% of the websites built on them. The "*" indicates web builders that provide third-party scripts which can lead to the usage of an "'unsafe-inline'" directive value in CSPs. The "Δ" indicates web builders that provide third-party scripts which can lead to the usage of an "'unsafe-eval'" directive value in CSPs.

Index	Web Builder Name	Index	Web Builder Name
1	1C-Bitrix	2	ΔWix
3	ΔBrightspot	4	ΔChorus
5	Prismic	6	HubSpot CMS Hub
7	ΔSquarespace	8	RebelMouse
9	*ΔTilda	10	*TownNews
11	*ΔAdobe Experience Manager	12	ΔOptimizely Content Management

B.3 The 26 CDNs That Have a Minimum of 10 Websites Using Them

Table B.3 lists all the 26 CDNs that have a minimum of 10 websites using them in the first crawling.

Table B.3 Details of the 26 CDNs that have a minimum of 10 websites using them. The "▲" indicates CDNs that provide third-party scripts to more than 50% of the websites enabled by them. The "*" indicates CDNs that provide third-party scripts which can lead to the usage of an "'unsafe-inline'" directive value in CSPs. The "Δ" indicates CDNs that provide third-party scripts which can lead to the usage of an "'unsafe-eval'" directive value in CSPs.

Index	Name	Index	Name
1	cdnjs	2	Cloudflare
3	Edgio	4	ArvanCloud
5	Imgix	6	Bunny
7	EdgeCast	8	Netlify
9	Fireblade	10	Fastly
11	▲ΔAkamai	12	Section.io
13	▲*ΔUnpkg	14	GoCache
15	▲*ΔjsDelivr	16	Cloudinary
17	Amazon S3	18	Tencent Cloud
19	Acquia Cloud Platform CDN	20	▲Microsoft Ajax Content Delivery Network
21	▲ΔGoogle Cloud CDN	22	Amazon Cloudfront
23	Azure CDN	24	▲*ΔGoogle Hosted Libraries
25	▲Yandex Cloud CDN	26	▲*ΔjQuery CDN

APPENDIX C

APPENDICES OF CHAPTER 4

C.1 The Types of CSP violations that can be Resolved under Google's Nonce-Based CSP Solutions

Table C.1 lists the types of CSP violations that can be resolved under Google's nonce-based CSP solutions.

Table C.1 Details of the types of CSP violations that can be resolved under Google's nonce-based CSP solutions.

CSP Solution	***Types of CSP Violations that Can be Resolved under a Nonce-Based CSP Solution***
CSP Solution at Level 4	inline scripts
	first-party scripts directly requested by the current webpage
	third-party scripts directly requested by the current webpage
	first-party scripts further requested by other scripts
	scripts inserted into the current webpage by first-party scripts
CSP Solution at Level 3	inline scripts
	first-party scripts directly requested by the current webpage
	third-party scripts directly requested by the current webpage
	first-party resources requested by other scripts
	third-party resources requested by other scripts
	non-parser-inserted scripts inserted into the current webpage by first-party scripts
	non-parser-inserted scripts inserted into the current webpage by third-party scripts
	parser-inserted scripts inserted into the current webpage by first-party scripts
CSP Solution at Level 2	inline scripts
	first-party scripts directly requested by the current webpage
	third-party scripts directly requested by the current webpage
	first-party resources requested by other scripts
	third-party resources requested by other scripts
	non-parser-inserted scripts inserted into the current webpage by first-party scripts
	non-parser-inserted scripts inserted into the current webpage by third-party scripts
	parser-inserted scripts inserted into the current webpage by first-party scripts
	eval-like functions induced by first-party scripts
	eval-like functions induced by third-party scripts
CSP Solution at Level 1	inline scripts
	first-party scripts directly requested by the current webpage
	third-party scripts directly requested by the current webpage
	first-party resources requested by other scripts
	third-party resources requested by other scripts
	non-parser-inserted scripts inserted into the current webpage by first-party scripts
	non-parser-inserted scripts inserted into the current webpage by third-party scripts
	parser-inserted scripts inserted into the current webpage by first-party scripts
	eval-like functions induced by first-party scripts
	eval-like functions induced by third-party scripts
	event handlers inserted into the current webpage by first-party scripts