

UNIVERSITY OF CRETE  
DEPARTMENT OF COMPUTER SCIENCE  
FACULTY OF SCIENCES AND ENGINEERING

# **Securing the Modern Web Through Novel Black-Box and Context-Agnostic Techniques**

by

Konstantinos Drakonakis

PhD Dissertation

Presented

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

Heraklion, June 2024



UNIVERSITY OF CRETE  
DEPARTMENT OF COMPUTER SCIENCE

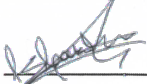
**Securing the Modern Web Through Novel Black-Box and Context-Agnostic Techniques**

PhD Dissertation Presented

by **Konstantinos Drakonakis**

in Partial Fulfillment of the Requirements  
for the Degree of Doctor of Philosophy

**APPROVED BY:**



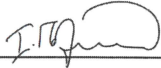
---

**Author:** Konstantinos Drakonakis

**SOTIRIOS IOANNIDIS** Digitally signed by SOTIRIOS IOANNIDIS  
Date: 2024.06.11 14:29:51 +03'00'

---

**Supervisor:** Sotiris Ioannidis, Associate Professor, Technical University of Crete



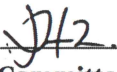
---

**Committee Member:** Jason Polakis, Associate Professor, University of Illinois at Chicago

PANAGIOTA FATOUROU  
11/06/2024 12:06

---

**Committee Member:** Panagiota Fatourou, Professor, University of Crete



---

**Committee Member:** Xenofontas Dimitropoulos, Professor, University of Crete

Polyvios Pratikakis Digitally signed by Polyvios  
Pratikakis  
Date: 2024.06.17 14:50:23 +03'00'

---

**Committee Member:** Polyvios Pratikakis, Associate Professor, University of Crete

KONSTANTINOS MAGOUTIS  
17/06/2024 18:15


---

**Committee Member:** Konstantinos Magoutis, Associate Professor, University of Crete

GEORGIOS VASILEIADIS Digitally signed by GEORGIOS  
VASILEIADIS  
Date: 2024.06.17 23:42:00 +03'00'

---

**Committee Member:** Giorgos Vassiliadis, Assistant Professor, Hellenic Mediterranean University



---

**Department Chairman:** Argyros Antonis, Professor, University of Crete

Heraklion, June 2024



To my beloved family.



# Acknowledgments

First of all, I would like to deeply thank my supervisor, Sotiris Ioannidis, for his guidance and, most importantly, for allowing me to be a member of this great lab and letting me follow my ideas and evolve through this process.

I would also like to express my sincerest gratitude to Jason Polakis, a great friend and mentor. Having worked with him all these years has taught me a great deal, but most importantly to not give up and pursue my goals. Your advice, guidance and support were always invaluable and appreciated; thank you.

I would also like to thank the members of my defense committee, Panagiota Fatourou, Xenofontas Dimitropoulos, Polyvios Pratikakis, Konstantinos Magoutis and Giorgos Vassiliadis, for their invaluable comments and questions.

Next, a huge thanks to all the friends that supported me in this journey, one way or another. A special thanks to Stefanos Fafalios and Michalis Diamantaris for all the good times and the trips; see you on the next one. Moreover, Victoria, thank you for *everything*, but mostly for being there through thick and thin and believing in me.

Finally, I would like to thank and dedicate this work to my family, whose support and unconditional love were always something I could count on. You helped me more than you can imagine. Thank you.





# Abstract

The World Wide Web has seen widespread adoption in the past decades, becoming an indispensable tool for our everyday activities. Undoubtedly, its prevalence can be largely attributed to its continuous evolution and the constant emergence of new functionalities and services. As a result, along with these new, convenient and often complex features, the Web ecosystem's complexity increases as well. At the same time, an increasing number of sensitive functionalities is being offered to users and a vast amount of private, sensitive information is circulated constantly. Unfortunately, this draws the attention of malicious actors who aim to gain access to such sensitive information and functionalities for their own nefarious purposes and profit, ultimately hurting users' privacy and safety. To make matters worse, the intricate nature of the Web provides attackers with a multitude of potential vantage points for launching attacks. It is therefore crucial to robustly secure Web applications and their assets, so as to mitigate such malicious acts and their implications.

Thankfully, the research community has proposed various defense mechanisms and countermeasures, as well as detection techniques for uncovering security issues and vulnerabilities. However, the ever-increasing complexity of the Web can often diminish existing approaches' effectiveness significantly or render them completely unsuitable. Moreover, the sheer scale of the Web, the vastly diverse and often closed-source Web applications and the complex interdependencies that drive them, hinder the efficacy of approaches that require *a priori* knowledge or specialized input to achieve their respective goal.

In this dissertation, we demonstrate that these intricacies of the Web and their inherent challenges, mandate automated, *black-box* solutions to be properly tackled. By proposing and extensively evaluating such novel approaches, covering various fronts of Web security and privacy, we showcase their significant benefits and improvements over prior systems. Most importantly, our systems adopt a *context-agnostic* modus operandi, i.e., they do not require any *a priori* knowledge or processing in their respective domain of operation.

Specifically, we initially propose a *scanner-agnostic* middleware framework which aims to transparently enhance existing black-box vulnerability scanners, by addressing their core limitations and improving their effectiveness both in terms of code coverage and vulnerability detection. Next, we design a fully automated, *website-agnostic*, black-box auditing framework for uncovering authentication and authorization flaws in Web applications and carry out the *first* large-scale study on such flaws to date. Finally, we address the problem of robust and real-time third-party script attribution, a crucial prerequisite for countless security and privacy countermeasures, by designing a novel, *script-agnostic* pipeline.

**Supervisor:** Professor Sotiris Ioannidis



# Περίληψη

Ο Παγκόσμιος Ιστός αποτελεί ένα απαραίτητο εργαλείο για ποικίλες καθημερινές μας δραστηριότητες τις τελευταίες δεκαετίες. Αναμφίβολα, η ευρεία επικράτηση και δημοφιλία του μπορεί, σε μεγάλο βαθμό, να αποδοθεί στη συνεχή του εξέλιξη και την αδιάκοπη εμφάνιση νέων λειτουργικότητων και υπηρεσιών. Σαν αποτέλεσμα, παράλληλα με αυτές τις νέες, χρήσιμες, αλλά και συχνά περίπλοκες λειτουργικότητες, αυξάνεται και η πολυπλοκότητα του ίδιου του οικοσυστήματος. Ταυτόχρονα, ένας αυξανόμενος αριθμός από ευαίσθητες λειτουργικότητες προσφέρεται στους χρήστες κι ένας τεράστιος όγκος από ιδιωτικές, ευαίσθητες πληροφορίες ανταλλάσσεται συνεχώς. Δυστυχώς, αυτό έλκει την προσοχή κακόβουλων χρηστών, οι οποίοι επιδιώκουν να αποκτήσουν πρόσβαση σε τέτοιες ευαίσθητες πληροφορίες και λειτουργικότητες για δικούς τους σκοπούς και κέρδος, πλήττοντας έτσι την ασφάλεια και την ιδιωτικότητα των χρηστών. Ακόμη, η ίδια η περίπλοκη φύση του Παγκόσμιου Ιστού προσφέρει στους επιτιθέμενους μια πληθώρα από πιθανά ευάλωτα σημεία που μπορούν να επιχειρήσουν να εκμεταλλευτούν, προκειμένου να εκτελέσουν τις επιθέσεις τους. Συνεπώς, η θωράκιση της ασφάλειας του Παγκόσμιου Ιστού και των εφαρμογών του κρίνεται υψίστης σημασίας, προκειμένου να περιοριστούν οι κακόβουλες ενέργειες και οι επιπτώσεις τους.

Αισίως, η ερευνητική κοινότητα έχει προτείνει ποικίλους μηχανισμούς άμυνας, αντίμετρα, καθώς και τεχνικές για τον εντοπισμό ευπαθειών και άλλων ζητημάτων ασφάλειας. Παρά ταύτα, η συνεχώς αυξανόμενη πολυπλοκότητα του Ιστού συχνά μειώνει σημαντικά την αποτελεσματικότητα των ήδη προταθέντων προσεγγίσεων και μηχανισμών, καθιστώντας τους ενίοτε εντελώς ακατάλληλους. Ακόμη, η υπερμεγέθης κλίμακα του οικοσυστήματος, η μεγάλη ποικιλία εφαρμογών, η έλλειψη πρόσβασης στον πηγαίο τους κώδικα και οι περίπλοκες αλληλεπιδράσεις που τις χαρακτηρίζουν, δυσχεραίνουν την χρήση τεχνικών οι οποίες απαιτούν την *a priori* απόκτηση γνώσης και πληροφοριών ή εξειδικευμένης εισόδου και παραμετροποίησης για να επιτύχουν τον στόχο τους.

Στην κείμενη Διατριβή, αποδεικνύουμε ότι οι ιδιαιτερότητες αυτές του Παγκόσμιου Ιστού και οι εγγενείς προκλήσεις που υποκρύπτουν, απαιτούν την χρήση αυτοματοποιημένων, *black-box* τεχνικών και λύσεων, ώστε να αντιμετωπιστούν αποτελεσματικά. Προτείνοντας κι αξιολογώντας εκτενώς τέτοιες καινοτόμες προσεγγίσεις, οι οποίες καλύπτουν διαφορετικές πτυχές της ασφάλειας και ιδιωτικότητας στον Ιστό, αναδεικνύουμε τα σημαντικά οφέλη, καθώς και τις βελτιώσεις που αυτές προσφέρουν σε σχέση με προηγούμενες μελέτες και συστήματα. Εξίσου σημαντικά, τα συστήματά μας

υιοθετούν μία αγνωστικιστική προσέγγιση ως προς το πεδίο και την μέθοδο λειτουργίας τους. Με άλλα λόγια, δεν απαιτούν καμία εκ των προτέρων γνώση ή επεξεργασία στο εκάστοτε πεδίο λειτουργίας τους, ώστε να επιτύχουν τον σκοπό τους.

Ειδικότερα, προτείνουμε ένα ενδιάμεσο λογισμικό (*middleware*) για υπάρχοντα συστήματα εντοπισμού ευπαθειών, το οποίο, επιλύοντας τους κύριους περιορισμούς των συστημάτων αυτών, στοχεύει στο να βελτιώσει τις επιδόσεις τους, τόσο σε ό,τι αφορά τον εντοπισμό ευπαθειών όσο και την κάλυψη κώδικα της εκάστοτε εφαρμογής που ελέγχεται. Εν συνεχεία, σχεδιάζουμε και υλοποιούμε ένα πλήρως αυτοματοποιημένο σύστημα για τον εντοπισμό ευπαθειών σε μηχανισμούς αυθεντικοποίησης κι εξουσιοδότησης σε εφαρμογές Ιστού. Χρησιμοποιώντας το σύστημά μας εκπονούμε την πρώτη μελέτη μεγάλης κλίμακας στα εν λόγω ζητήματα ασφάλειας. Τέλος, επιλύουμε το πρόβλημα του καταλογισμού (*attribution*), σε πραγματικό χρόνο, προγραμμάτων τρίτων μερών που εμπεριέχονται σε εφαρμογές Ιστού, το οποίο αποτελεί ένα κρίσιμο προαπαιτούμενο για ένα μεγάλο πλήθος αντιμετρώων ασφάλειας και ιδιωτικότητας.

**Επόπτης:** Σωτήρης Ιωαννίδης

# Contents

Acknowledgments . . . . .	vii
Abstract . . . . .	ix
Περίληψη (Abstract in Greek) . . . . .	xi
Table of Contents . . . . .	xiii
List of Figures . . . . .	xv
List of Tables . . . . .	xvii
1 Introduction . . . . .	1
1.1 Thesis Statement and Contributions . . . . .	3
1.2 Organization of Dissertation . . . . .	5
1.3 Publications . . . . .	6
2 Black-Box Web Application Scanning . . . . .	7
2.1 Challenges and Design Requirements . . . . .	9
2.2 Design and Implementation . . . . .	10
2.2.1 Navigation model . . . . .	11
2.2.2 System Components . . . . .	13
2.2.3 URL Clustering . . . . .	21
2.2.4 API Abstraction for Future Scanners . . . . .	27
2.3 Experimental Evaluation . . . . .	27
2.3.1 Other Vulnerabilities . . . . .	32
2.3.2 URL Clustering . . . . .	34
2.3.3 State-of-the-Art Comparison . . . . .	35
2.4 Limitations and Future Work . . . . .	37
3 Black-box Auditing for Web Authentication and Authorization Flaws . . . . .	41
3.1 Background and Threat Model . . . . .	43
3.2 System Design and Implementation . . . . .	44
3.2.1 Automated Account Setup . . . . .	44
3.2.2 Cookie Auditor . . . . .	49
3.2.3 Privacy Leakage Auditor . . . . .	52
3.2.4 Browser Automation . . . . .	53
3.3 Experimental Evaluation . . . . .	56
3.4 Discussion . . . . .	65
4 Robust and Real-Time JavaScript Attribution . . . . .	67
4.1 Motivation . . . . .	69

4.2	Design and Implementation . . . . .	70
4.2.1	Capturing 3P Scripts . . . . .	71
4.2.2	AST rewriting . . . . .	73
4.2.3	StyxJS Concealment . . . . .	76
4.2.4	Maintaining Security Mechanisms . . . . .	78
4.2.5	Summary . . . . .	81
4.3	Experimental Evaluation . . . . .	81
4.3.1	Validation . . . . .	82
4.3.2	Performance Evaluation . . . . .	84
4.4	Use Cases . . . . .	85
4.4.1	SugarCoat . . . . .	86
4.4.2	LeakInspector . . . . .	90
4.4.3	ScriptProtect . . . . .	92
4.5	Discussion & Limitations . . . . .	93
5	Related Work . . . . .	95
5.1	Black-Box Web Application Scanning . . . . .	95
5.2	Web Authentication and Authorization . . . . .	96
5.3	Third-party Scripts and Attribution . . . . .	99
6	Conclusion . . . . .	101
6.1	Summary . . . . .	101
6.2	Future Work . . . . .	102
	Bibliography . . . . .	105
<b>Appendices</b>		
A	. . . . .	129
A.1	Static file extensions . . . . .	129
A.2	ReScan's API . . . . .	129
A.3	Scanners' Configuration . . . . .	130
B	. . . . .	131
B.1	JS inclusions . . . . .	131
B.2	PageGraph Validation . . . . .	131

# List of Figures

2.1	Architecture and workflow of the ReScan framework. . . . .	11
2.2	Crafted HTTP response. . . . .	17
2.3	Total scan time in seconds for each app/ scanner pair with and without ReScan. . . . .	30
2.4	Requests' CDF per application, in terms of total as well as individual components' processing time. . . . .	31
3.1	Major phases in our auditing workflow. . . . .	44
3.2	Success rate for different workflow phases. . . . .	56
3.3	Percentage ( <b>left</b> ) and absolute number ( <b>right</b> ) of vulnerable domains per ranking bin. . . . .	60
3.4	Time required by each module of our system. . . . .	62
4.1	Architecture and workflow of StyxJS. . . . .	70
4.2	Example stack snapshot for <code>async/await</code> functions. (a) An <code>eval</code> script (completed) has previously called <code>foo</code> , which is now suspended due to <code>await</code> . (b) Later, a script injected with <code>document.write</code> executes and calls the asynchronous function <code>bar</code> . (c) <code>bar</code> is also suspended and the dynamic script exits. (d) <code>foo</code> resumes execution and exits. (e) <code>bar</code> resumes execution, exits and all completed frames are removed. . . . .	75
4.3	Test page's JIG. We omit script node details for brevity. . . . .	83
4.4	Page loading time delta for 1st and 2nd visit. . . . .	84





# List of Tables

2.1	Scanners' features and capabilities. . . . .	9
2.2	Number and type of unique vulnerabilities discovered by each scanner without (left) and with ReScan (right) for each app. . . . .	29
2.3	Total lines of code (LoC) executed by ReScan (R), the standalone scanner (S), and common to both of them ( $R \cap S$ ). . . . .	30
2.4	Qualitative differences between ReScan and Black Widow. . . . .	35
2.5	Detection and coverage comparison between the best run of ReScan and Black Widow for each app. . . . .	36
3.1	Number of unique domains that do not adequately protect their cookies from specific attacks. . . . .	58
3.2	Number of domains for different values of authentication cookies and combinations of authentication cookies. . . . .	60
3.3	Most common categories of susceptible domains. . . . .	61
3.4	Personal user data that can be obtained by attackers. . . . .	62
3.5	Manually validated domains and hijacking capabilities. . . . .	64
4.1	Bypass techniques tested against StyxJS. . . . .	82
4.2	API calls captured by SugarCoat and StyxJS. . . . .	89
4.3	Comparison between ScriptProtect and StyxJS. . . . .	92
B.1	JavaScript inclusion methods covered by different systems. Exposing properties are marked with *. . . . .	132



# Chapter 1

## Introduction

*"I don't think we've even seen the tip of the iceberg. I think the potential of what the Internet is going to do to society, both good and bad, is unimaginable. [...] It's an alien life form." - David Bowie, 1999*

The World Wide Web has been an integral part of our everyday lives for over two decades, offering a plethora of services and functionalities pertinent to leisure, social, professional and financial activities, among others. According to recent reports, the number of users with Internet and Web access has skyrocketed to *5.3 billion*, accounting for *65.7%* of the global population [86]. This vast prevalence of the modern Web can be largely attributed to its ever-evolving nature. Specifically, the Web has gradually transformed from a set of static pages to a fully interactive ecosystem, with new features, functionalities and services emerging constantly [2, 27, 94]. At the same time, the Web's complexity increased as well; websites turned into massive applications, with complex interdependencies between them [158, 202, 266], and browsers evolved from simple navigators to powerful software components [2, 27, 77, 244].

It is also apparent that a vast amount of information is constantly being circulated in the Web, which can often be extremely sensitive due to the nature of our online activities. As a consequence, such sensitive information *and* functionalities are high-value targets for malicious actors, known as *attackers*. As a matter of fact, numerous incidents through the years have demonstrated the prevalence and severity of Web-related attacks, which gained access to user accounts, credentials and even financial information, affecting *millions* of users [9–12, 51]. To make matters worse, due to the Web's complex architecture and the different entities involved therein, the attack surface is extremely vast, offering a multitude of different vantage points for launching attacks. For instance, an attacker could exploit vulnerabilities in the Web browser itself [187, 206, 236, 279], hijack the *Hypertext Transfer Protocol* (HTTP) [35] communication between browsers and websites at the network level [173, 192, 237], or even compromise entire websites or the third-party libraries they

might include [158, 176, 201, 202, 266]. Thankfully, the security and research communities have put great effort into fortifying the Web's overall security posture and users' privacy.

Specifically, several approaches have been proposed to detect various vulnerabilities and security issues in Web applications [118, 132, 141, 147, 155, 207, 234, 281], so as to be patched in a timely manner before they are uncovered and exploited by real attackers. One of the most prominent and useful categories of such systems, are the so called *black-box* vulnerability scanners [89, 132, 141, 167, 213, 222, 259]. As their name suggests, these systems treat the application under test as a black box, i.e., without requiring any a priori knowledge of the application, such as its source code or specialized input from its developers. Besides requiring minimal input, the major advantage of such systems is their ability to approach the application from a realistic attacker's perspective and shed light on possible attack vectors and vulnerabilities a real attacker could exploit. Another advantage of black-box techniques is their inherent resilience against code obfuscation, which has seen wide adoption in recent years [230, 241]. As such, as the Web continues to evolve rapidly, by incorporating more complex functionalities, APIs and client-side code, it is of crucial importance that vulnerability scanners follow along and adapt their capabilities to these changes, so as to remain effective. However, research studies have extensively demonstrated that existing black-box vulnerability scanners tend to suffer from several core limitations that hinder their effectiveness [105, 133, 135, 141, 208, 260, 270], both in terms of vulnerability detection, as well as code coverage. For instance, while existing scanners offer the option to initially log into the target application to test post-authentication functionalities, they typically assume that the established authenticated session remains intact for the remainder of the scanning process. This, however, is not necessarily the case and can lead to incomplete scans and missed vulnerabilities. Similarly, black-box scanners usually follow a naive approach when testing functionalities, by individually replaying their corresponding, specific HTTP requests. In reality, modern applications might require a series of initial, necessary steps to transition the application in the appropriate state *before* exercising a specific functionality.

Additionally, one of the most fundamental aspects of Web applications has been their ability to offer personalized content and services through user accounts. In more detail, users can register and log into their accounts to access a variety of different functionalities, such as social networking, content sharing, financial transactions and more. Thus, it becomes clear that the sensitive nature of such information and functionalities renders user accounts, and the authenticated *sessions* between users' browsers and Web applications, treasure troves for ambitious attackers. Unfortunately, several (semi-)manual studies have shown that even popular, high-profile websites with millions of users are susceptible to session-hijacking attacks [114, 115, 200, 238], while developers often struggle with the correct or complete deployment of relevant security mechanisms that could prevent them [118, 153, 173, 226, 227, 237, 246, 275]. Moreover, due to the significant challenges of *entirely* automating the auditing process and adopting a *website-agnostic, black-box* ap-

proach, these manual studies were inherently small-scale and only considered several popular websites, leaving the rest of the Web unexplored. As such, the true extent of such *authentication* and *authorization* flaws, along with the privacy implications users might suffer, remain unknown.

Finally, another aspect that can have severe implications on websites' security and users' privacy are the complex interdependencies that drive the Web. Specifically, one of the earliest and most prevalent [202] such dependencies, are the so called *third-party scripts*. In essence, a *first-party* website might include JavaScript (JS) files from other, unrelated domains for a multitude of additional functionalities, e.g., advertising [70], Single-Sign On [36, 48] or analytics [83]. Crucially, third-party scripts have certain characteristics that render their usage rather perilous. First, third-party scripts intentionally loaded by the first-party website, can *implicitly* load additional scripts dynamically at their discretion [158, 176, 266]. In addition, once loaded, third-party scripts operate with the exact *same privileges* as first-party code, opening up a plethora of attack vectors. In more detail, research studies have demonstrated that third-party scripts have been the source for a multitude of security and privacy issues. For instance, they can introduce client-side vulnerabilities [158, 201], prevent HTTPS deployment [176], carry out cookie-stealing attacks [134] or invasively track and fingerprint users [139, 159, 257]. Therefore, the ability to *robustly* attribute third-party script execution, i.e., disambiguate first- from third-party code, *at runtime*, is a crucial requirement for building effective protection mechanisms and countermeasures. Unfortunately, no standardized approach exists to achieve third-party script attribution and current solutions are either susceptible to bypasses [124, 160], cannot provide *real-time* security and privacy benefits for end users [56, 239] or require significant offline pre-processing to operate correctly [201, 242], diminishing both their practicality *and* effectiveness.

## 1.1 Thesis Statement and Contributions

Due to its ever-increasing complexity, the Web has often - and rightfully - been dubbed the *tangled Web* [176, 199, 254]. Despite the continuous and invaluable efforts by the research community to entrench several of the Web's security and privacy aspects, the constant addition of new features, APIs and client-side code, only tangles the ecosystem further, often hindering existing approaches' effectiveness. At the same time, new intricate challenges emerge that require novel and careful design choices to be properly addressed. Most importantly, the sheer scale of the modern Web, the vastly diverse Web applications and their assets (e.g., scripts) and their often closed-source, proprietary nature render the *a priori* collection of required information to effectively tackle said challenges a daunting and possibly infeasible task. As a consequence, severe vulnerabilities and security issues might remain undiscovered, while defense and privacy preserving mechanisms might fall short in achieving their respective goals, posing a significant threat to both websites' security and users'

privacy.

In summary, in this dissertation we aim to demonstrate that *the inherent complexity of the Web and the unique challenges it presents, stemming from the sheer scale of the ecosystem, the constant emergence of new features, the increasing addition of complex client-side code and the diverse nature of Web applications and their interdependencies, mandate novel, automated black-box and context-agnostic techniques to i) effectively uncover vulnerabilities and security issues at scale and ii) develop robust defense mechanisms and countermeasures.*

Specifically, in this dissertation we make the following contributions:

- We design and implement ReScan, a novel, *scanner-agnostic*, black-box middleware framework that *transparently* enhances existing (and future) Web application scanners by addressing their core limitations. In more detail, our system intercepts all HTTP requests initiated by the scanner, executes them in a modern, state-of-the-art browser to ensure realistic interaction with the Web application and employs a series of modules to address several identified limitations, e.g., persisting the authentication state and uncovering client-side events. By extensively evaluating ReScan using both popular open-source and academic black-box scanners, against a rich set of benchmark and modern applications, we find that it aids the underlying scanners in identifying more vulnerabilities, while increasing the achieved code coverage by 168% on average. ReScan, as well as our applications' Docker images, have been open-sourced to facilitate further research in the field of black-box vulnerability scanning [59, 60].
- We design a novel URL clustering algorithm that prevents black-box scanners from spending valuable time and resources in *redundantly* testing similar pages, which essentially expose the same functionality. Our evaluation demonstrates that our algorithm can reduce the total scanning time by  $\sim 6.7$  times in a representative, modern application.
- We develop *XDriver*, a custom browser automation tool that transparently offers robustness during prolonged interactions with Web applications. Our tool is tailored for security-oriented tasks and includes modules for assessing relevant security mechanisms, as well as a rich set of auxiliary functionalities (e.g., built-in crawler, network level HTTP request interception and tampering). As our system can streamline a wide range of research projects, our code has been made open-source [31].
- We develop a novel *website-agnostic* framework for the automated, black-box detection of authentication and authorization flaws in Web applications. Our framework incorporates a series of modules and oracles that employ differential analysis for automatically evaluating the feasibility of cookie hijacking attacks under different threat models, while taking into consideration and assessing the deployment of relevant security mechanisms. Moreover, our system explores and detects the exposure of sen-

sitive information and user data, in case of a successful attack. To facilitate further research, we share our auditing pipeline with vetted researchers.

- We leverage our framework to conduct the *first fully automated*, large-scale study of cookie-based authentication and authorization flaws by auditing ~25K domains, several orders of magnitude larger than prior studies with a similar focus. Our comprehensive evaluation reveals a plethora of security malpractices and misconfigurations, as 50.3% of the domains are vulnerable to at least one attack. To make matters worse, relevant security mechanisms that could prevent the attacks are seldom deployed (11.8% of vulnerable domains do so) and are often misconfigured, effectively nullifying their benefits. We responsibly disclosed our findings to ~43% of affected domains and setup a passive notification service for developers to acquire our results after proving ownership of their domain.
- We develop StyxJS, a system that enables *robust* and *real-time* third-party script attribution in a *script-agnostic* manner, i.e., without requiring any pre-processing or knowledge of the encountered scripts. Our extensive evaluation shows that our system effectively captures a multitude of script inclusion methods, while preventing common and custom bypass attempts. It also does not incur page breakage and induces a negligible performance overhead in most cases. Importantly, StyxJS integrates seamlessly with page-deployed security mechanisms (e.g., CSP [24]), and takes precautions to ensure their security guarantees are never affected by its operation.
- We analyze three existing security and privacy countermeasures and provide multiple novel bypass techniques for each one, highlighting the practical gap StyxJS aims to fill. Subsequently, we retrofit their approaches as custom plugins on top of StyxJS, demonstrating the significant improvements it offers in achieving their respective goal, as well as its flexibility to accommodate diverse pipelines that may have vastly different goals. StyxJS, along with its plugins, have been made open-source [92] to streamline the creation of more robust security and privacy countermeasures.

## 1.2 Organization of Dissertation

The rest of this dissertation is organized in the following way. Chapter 2 highlights the core limitations of existing Web black-box vulnerability scanners that hinder their effectiveness in terms of code coverage and vulnerability detection. We then introduce our novel scanner-agnostic middleware approach and demonstrate how it can transparently address these limitations and enhance the overall scanning process, as well as the significant improvements it offers.

Chapter 3 presents our work in detecting cookie-based authentication and authoriza-

tion flaws in Web applications, using novel, black-box and website-agnostic techniques. We also introduce our custom browser automation tool, offering robustness during prolonged interactions with Web applications and several auxiliary functionalities. Finally, we carry out the first fully automated large scale study on cookie-based authentication and authorization flaws in the wild and report on our alarming findings.

Chapter 4 highlights the necessity (and lack) of robust and real-time script attribution for developing effective countermeasures and details how our proposed approach can achieve it in a script-agnostic manner. We also present novel bypass techniques against existing countermeasures and then retrofit them as custom plugins on top of our system, demonstrating its ability to effectively solve their limitations and achieve their respective goal.

Chapter 5 presents the related work to the approaches, techniques and systems proposed in this dissertation. Finally, Chapter 6 summarizes our contributions and key results and proposes possible directions for future work.

### 1.3 Publications

Parts of the work for this dissertation have been published in international refereed conferences:

- Kostas Drakonakis, Sotiris Ioannidis, Jason Polakis. ReScan: A Middleware Framework for Realistic and Robust Black-box Web Application Scanning. In *Proceedings of the 30th Annual Network and Distributed System Security Symposium (NDSS)*. February 2023.
- Kostas Drakonakis, Sotiris Ioannidis, Jason Polakis. The Cookie Hunter: Automated Black-Box Auditing for Web Authentication and Authorization Flaws. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. November 2020.

At the time of writing, our work on robust and real-time JavaScript attribution, detailed in Chapter 4, is currently under submission:

- Kostas Drakonakis, Sotiris Ioannidis, Jason Polakis. Dredging the River Styx: Fortifying the Web through Robust and Real-Time Script Attribution



## Chapter 2

# Black-Box Web Application Scanning

Web application scanners play a crucial role for security engineers and developers for uncovering vulnerabilities in applications and patching them in a timely manner. Black-box scanners can be extremely useful since they do not require any *a priori* knowledge of the target application. However, as the web ecosystem continues to evolve at a breakneck pace, modern applications incorporate more complex functionalities, features [168], APIs [191], and client-side code, and therefore need a fully-fledged, modern browser environment for their functionality to be fully exercised and the applications to be accurately tested.

State-of-the-art academic [132, 197, 213] and community-developed scanners [89, 222, 258, 259], which have seen wide recognition, suffer from core limitations that hinder their effectiveness and lead to incomplete scanning, lower coverage and missed vulnerabilities. First, many existing scanners use raw HTTP requests to interact with the application instead of a real browser, thus missing out on dynamically generated DOM content (e.g., new URLs or forms) and asynchronous requests. Second, many scanners are limited to a specific method of navigating the application (e.g., extracting static links and HTML forms) or rely solely on client-side code and events [213]. Applications, however, often make use of both. Moreover, existing tools typically simply replay requests when crawling or fuzzing the application, and do not adhere to the intended and correct execution of steps for moving the application from one state to another. Another major limitation is that while scanners can be configured to log into the application, they typically assume that the authenticated session remains intact for the duration of the scan, which quite often is not the case. In addition, scanners can be prone to false positives and negatives for certain types of vulnerabilities (e.g., XSS) due to their naive approach to verifying successful injections. Finally, since black-box scanners are not context-aware of applications' content and functionalities, they can spend a significant amount of time redundantly testing similar pages.

Recently, Eriksson et al. [141] highlighted some of these problems and the importance of taking them all into account when implementing a web application scanner. These limitations are further evidenced by the fact that certain scanners attempt to tackle *some* of them by offering the ability to be used as a proxy [3, 8, 90] between a user's browser and the

application, so as to collect useful information (e.g., event originating requests). However, this is not a robust or effective strategy, as it requires significant manual effort and therefore does not scale, and is inherently unable to address all the limitations. Overall, while *certain* scanners attempt to address these limitations, they either only *partially* address them or only tackle a *subset* of the limitations.

Nonetheless, despite their limitations, the aforementioned tools offer a plethora of different scanning techniques and capabilities which are undoubtedly of great value. Ideally, overcoming these limitations would require redesigning these tools or collecting their individual techniques and re-implementing them from scratch. Unfortunately, this is an unlikely and impractical scenario, as it would require an exorbitant amount of time and engineering effort. Instead, we propose an alternative strategy for leveraging the capabilities of existing (and future) scanners while addressing their limitations.

Specifically, we design and implement ReScan, a scanner-agnostic black-box middleware framework that *transparently* enhances web application scanners and addresses the aforementioned limitations. In more detail, our framework intercepts scanner requests and provides a realistic state-of-the-art orchestrated browser environment with a rich set of additional capabilities (e.g., event triggering, HTTP request tampering). Our system detects new endpoints that reveal further endpoints or trigger asynchronous requests, to construct a navigation model of the target web application, and mirrors the scanner's requests through the browser and the model. Additional enhancement modules operate concurrently to verify the validity of the authenticated session and re-authenticate if needed, detect *inter-state dependencies* (i.e., submitted values that appear on and affect other URLs) and cluster similar pages that would be redundant to audit. In general, ReScan does not require *any* information about the scanner's or app's internals and does not make any assumptions; it receives HTTP requests and attempts to accurately mirror them based on the learned model so as to respect the navigation workflow. This is done inside the browser, to ensure realistic interaction and response rendering.

Our extensive evaluation with state-of-the-art scanners shows that ReScan effectively facilitates the detection of more vulnerabilities, both for benchmark and modern applications, while offering a code coverage improvement between 3% and 935% (168% on average). Moreover, we outline several prominent vulnerability examples that demonstrate the practicality of our different enhancement techniques and also show that ReScan can handle more than a single class of vulnerabilities. While our system induces a considerable performance overhead, due to the numerous techniques it employs and the unavoidable cost of leveraging a fully-fledged modern browser, we show that our URL clustering algorithm can dramatically reduce the total scan time for a representative modern application, resulting in a 6.7x speedup.

Table 2.1: Scanners' features and capabilities.

Feature / System	w3af	wapiti	Enemy of the State	ZAP
Browser support	○	○	○	●
Navigation model	○	○	●	○
Inter-state dependencies	○	○	○	○
Client-side events	○	○	○	●
Authentication	◐	◐	●	●
FP / FN elimination	○	○	○	○
URL clustering	○	○	◐	○

●: feature supported, ◐: partially supported, ○: not supported.

## 2.1 Challenges and Design Requirements

Implementing a scanner-agnostic middleware framework requires solving numerous technical challenges and overcoming the aforementioned limitations in a way that is *transparent* to the scanner, while also providing functionality enhancements without understanding or tampering with scanners' internals. Essentially, the black-box interaction should be *bidirectional*; the scanner knows nothing about ReScan and vice-versa. Here we outline some of the main challenges that our system tackles.

**Inter-state dependencies.** Certain types of vulnerabilities, such as stored XSS, are not necessarily triggered directly on the landing page after the payload is delivered. On the contrary, successful exploitation (and detection) might require the scanner to visit a different URL in the application. For instance, consider editing a vulnerable field on a user's account page, which is then triggered when visiting that user's profile page. ReScan needs to uncover and keep track of these *inter-state dependencies* so as to enable detection of said vulnerabilities, and also account for the *order* in which a scanner fuzzes potential injection points and visits URLs they might affect.

**Authentication.** Modern web apps typically include functionality and resources that are only available post authentication, and also include account and session management features that terminate active sessions. When performing an authenticated scan, at some point the session might break, e.g., due to following the logout URL or sending a malformed request that the web application cannot handle and all session info is invalidated. This directly affects the coverage and vulnerability detection scanners can achieve, as they might not be able to detect such state changes on time, or even at all, and proceed to perform an incomplete scan. ReScan needs to account for this major limitation as well, by ensuring the session remains valid for the entire duration of the scan and for all tested functionalities.

**False positives & negatives.** Scanners typically assert successful exploitation of certain vulnerabilities by checking whether the injected payload appears as-is in the application's response. This is not foolproof and is prone to false positives, as the mere existence of the payload inside the DOM does not necessarily imply successful exploitation. Additionally, a

scanner might successfully exploit a vulnerability but not be able to detect it as the payload's structure might have been slightly altered or even completely stripped *after* being executed, leading to false negatives. Since we cannot tamper with each scanner's internal detection mechanisms, we need to devise a mechanism so as to eliminate both false positives and false negatives, or provide additional information to the user about such possible cases to facilitate *triage*.

**URL clustering.** Web app scanners can typically spend a significant amount of time testing redundant application endpoints, i.e., pages that are conceptually similar and offer the exact same functionality. It is apparent that such behavior directly affects their performance and overall scanning times. Thus, ideally, we need to prevent scanners from ever learning the existence of such redundant endpoints, by efficiently and effectively comparing and clustering them under a single, representative URL.

**Response enhancement.** Even if we successfully overcome the core scanner limitations, we still need to communicate ReScan's findings back to the scanner, such as request-triggering or DOM-changing client-side events. Due to our black-box approach, ReScan cannot directly interact with the scanner and is restricted to the existing communication channel (i.e., the HTTP connection) for transmitting artifacts.

**Limitations of prior work.** In Table 2.1 we outline the capabilities of the different scanners that we evaluate our system on (§ 2.3), so as to paint a clear picture of how each one can benefit from ReScan's enhancement techniques. As can be observed, only a single scanner leverages a modern, full-fledged browser environment; the rest are oblivious to dynamic content, functionalities and client-side events. Similarly, *only one* scanner leverages a navigation model to properly traverse the application, and *none* of them consider dependencies between different endpoints of the app. All of them offer some mechanism to handle authentication, however, some do so partially, i.e., assume that the authenticated session remains valid throughout the scan. When it comes to false positives and negatives, none of them take steps to eliminate them. Similarly, for all scanners we find that they spend significant time testing redundant URLs or can incorrectly exclude pages from scanning due to deeming them to be similar to other tested pages. Overall, we find that all scanners do not take into account *at least four* aspects that can directly affect their vulnerability detection efficacy and achieved code coverage. This motivates our novel design approach of offering a framework that operates as a middleware component for enhancing the capabilities of any vulnerability scanner.

## 2.2 Design and Implementation

Figure 2.1 presents an overview of ReScan's architecture. Our system consists of a series of modules that operate concurrently and communicate with each other. The entry point to the system is an *intercepting proxy* which captures scanner requests and feeds them into

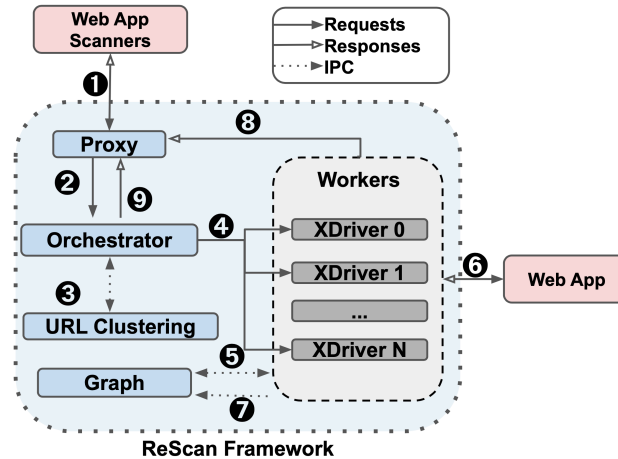


Figure 2.1: Architecture and workflow of the ReScan framework.

the system. Next, the intercepted requests are loaded by the *orchestrator* and passed on to the *browser workers*, each of which has its own browser instance and attempts to accurately mirror the request in the application. They are also responsible for detecting new endpoints, e.g., links, forms and event originating requests, which are all wrapped in a single, enhanced HTTP response and sent back to the scanner. All discovered endpoints are stored in the application’s navigation model by the *graph worker*, which is leveraged by the browser workers to properly retrace and execute all necessary steps when executing a request. Each worker utilizes the *authentication helper* module to ensure the session is valid. Meanwhile, the *background worker* is responsible for inspecting all data submitted to the application and uncovering *inter-state dependencies* (i.e., if a value submitted in a page appears on another one) which is necessary for detecting certain vulnerabilities (e.g., stored XSS). Finally, the *URL clustering* module employs a novel algorithm to detect similar pages, and notifies the orchestrator to filter out such requests and prevent the scanner from spending valuable time on redundantly testing them. Before describing each component in detail, we first describe our navigation model (which is inspired by the approach of Eriksson et al. [141]). It is important to stress that while *some* of our techniques are inspired by prior work, incorporating them into our middleware-style architecture is a demanding process that requires a methodical design strategy and addressing numerous implementation challenges, as we outline in the following sections.

### 2.2.1 Navigation model

Our model is realized as a directed graph, where each node represents the state of the application in terms of unique URLs, and edges describe the transitions between states – the specific actions required to move from one state to another, like client-side events or forms.

**Edges.** Specifically, the edges are one of five types: GET, FORM, EVENT, IFRAME or REDIRECT.

Thus, when visiting a page we collect all such edges and add them to the model. Each edge is assigned a unique ID, which consists of the edge type, the destination URL, and the normalized HTTP payload (i.e., the set of parameter names present in the payload). We ignore the payload values as they are volatile and can be altered, e.g., when the scanner fuzzes a form. The payload information is required since two forms may point to the same URL, but perform different operations in the back-end, e.g., a login and a signup form that both POST data to the `/auth.php` endpoint. We need a way to distinguish the two different transitions and the edge type and destination URL alone would not suffice. Additionally, edges are annotated with any information required to replay them, e.g., for form edges we store the specific element's unique CSS selector in the page, while for events we also store the exact event type. Moreover, each edge is also connected to its parent edge, so we can construct workflows.

**Mapping requests to edges.** Since our system's input are raw HTTP requests, we need a way to map them to existing edges in the navigation model, so we can properly replay the correct workflow. An incoming request can initially be assigned only two of the five edge types, i.e., GET or FORM, depending on the HTTP method and whether it carries a payload. However, in practice, the request might consider a different edge type, e.g., an asynchronous POST request triggered by an event. To handle such cases, and only when we cannot locate an edge with the initial edge type, we simply change the type and check again. In more detail, a simple GET request can be mapped to a GET, IFRAME, EVENT or REDIRECT edge, since it does not include a payload. In contrast, a payload-bearing request can either be mapped to a FORM or an EVENT edge, since these are the only edge types that can transmit a payload. Subsequently, to properly execute the request, we recursively construct its *workflow* by following the parent edges until we find a *safe* GET edge, which based on the HTTP specification [145] is not considered state-changing and can be safely executed as a starting point.

**Arbitrary requests.** The model is constructed based on the edges the system observes in each page, i.e., links, forms, events, iframes and redirections. However, scanners are not restricted to sending these exact same requests, which we can map and replay based on the model. Instead, they can send arbitrary requests to *virtually* any endpoint of the application. For instance, they can extract the URL `example.com/user/auth/` from a form action (which is included in the navigation model) and send a request to a part of that URL (e.g., `example.com/user/`) that has not been observed by the system so far and is *not* included in the model. Depending on the request's type, we tackle this issue in two ways. For simple GET requests, we simply execute the request, as GET requests are not considered state-changing. In contrast, for arbitrary requests that include a payload (i.e., corresponding to FORM or EVENT edges) we don't have any information on the necessary workflow needed to properly replay them, which might not even exist in the first place. In this case, we employ a best-effort approach where we generate a form matching the request (same input fields, action and

HTTP method) on the fly and submit it. This way we can at least get a properly rendered response through the browser's environment.

**Model reuse.** It is worth noting that the constructed navigation model is not scanner-specific, but rather a generic, high-level representation of the web application. In practice, this allows the reuse of existing models either when reconfiguring a scanner or performing a completely new scan with another tool, effectively limiting the costly parts of ReScan's processing to the first run on a new application.

### 2.2.2 System Components

**Intercepting proxy.** This component accepts incoming requests from a scanner and constitutes *one of the two* requirements for our system's operation: the scanner *must* be configured to send its requests through this proxy. The other is setting the scanner's timeout per HTTP request to a large value, so ReScan has enough time to employ its numerous enhancement techniques. Both capabilities are supported by all scanners we have encountered so far. Our component is built as an add-on script on top of *mitmproxy* [123]. When a request is intercepted a new thread is spawned to handle it and our custom add-on code is executed ❶. Initially, the request is appended to all intercepted requests, along with all other relevant information (HTTP headers, method, payload and URL). The request thread then waits to receive its response from the system ❸, which will be sent back to the scanner. The only exception are requests towards static resources that do not have state-changing effects and are directly proxied to the target application. We achieve this by filtering requests based on known file extensions, which we list in Appendix A.1.

**Orchestrator.** This component periodically loads the intercepted requests populated by the proxy ❷, and enqueues any newly appended requests in a FIFO queue to ensure requests are served by the browser workers ❹ in the order they appear. It is also responsible for initializing and configuring all other components of the system.

**Browser workers.** The browser workers' goal is to accurately mirror each request through a fully-fledged, automated browser by executing the necessary workflow from the navigation model. We build the workers by leveraging XDriver [134], a robust Selenium-based browser automation tool with a rich set of security-oriented features pertinent to our goal (e.g., extracting HTTP redirection flows, spoofing request headers). The number of workers is a configurable parameter.

*Serving requests.* Initially, each worker reserves a request, constructs the corresponding edge ID based on the request information, queries the *graph worker* (described later on) so as to fetch the edge's workflow ❺ and proceeds to execute it ❻. Before executing the workflow the worker sets its cookie jar according to the request's *Cookie* header, so as to acquire the necessary state for authentication. Executing individual edges is rather straightforward; the worker simply fetches the page, fills and submits a form, and switches the browser's fo-

cus inside an iframe or triggers an event. We have also extended XDriver so as to capture events' asynchronous requests or redirects with the browser's internal proxy and make all necessary modifications before sending it to the application. However, in certain cases, some edges might not be required or cannot even be replayed. For instance, an intermediary edge corresponding to a login form that we previously used to log into the application will most likely *not* be present when executing the workflow of an admin dashboard's functionality. In such cases, we simply ignore non-existing, intermediary edges and proceed to the rest of the workflow.

Another special case involves EVENT edges when executing workflows. The main idea is that a client-side event that reveals further edges when triggered (e.g., other events or a form), might not be required to be triggered more than once to reveal these elements; on the contrary, triggering the event again might make these elements disappear or become inaccessible (e.g., removing a dynamically generated form). To address such cases, when encountering an EVENT edge, we perform a look-ahead operation on the workflow. In more detail, we check if the last edge (i.e., the one corresponding to the initial request) is already present and can be executed, in which case we simply ignore all intermediary edges. If it does not exist, we check for the second to last edge and repeat the same process, until we either find an edge to jump to and continue the execution from there, or end up on the event edge at hand and continue as normal.

Moreover, when a scanner sends a request containing a CSRF token, it will include a stale token value, as it was captured in a previous request. During the execution of its workflow ReScan will acquire a fresh token value, but in order to correctly submit the request, it needs to replace the stale scanner value with the new one. To achieve this after executing the workflow, when we are ready to send the request (i.e., submit the form or trigger the event) we check each payload parameter against common token and nonce keywords and simply ignore the scanner-provided value. For forms we iteratively inspect the form's input elements statically in the DOM, while for event-originating requests we perform the inspection on-the-wire using XDriver's internal proxy. Since this is a best-effort approach based on common naming conventions for CSRF tokens, it is important to note that even if ReScan misses an actual token (i.e., incorrectly assumes it is a regular parameter) it does not negatively affect the underlying scanner, which would miss it even without our system. Moreover, scanners can typically be configured to ignore specific parameters, such as CSRF tokens; in such cases, this mechanism would essentially be idle as the scanner would not fuzz the tokens.

Another aspect we need to consider are the HTTP request headers sent by the scanner in each request. Generally, we want to avoid sending the scanner's headers and need the browser to send its own to ensure realistic interaction (e.g., user-agent, accept-encoding etc.). However, there are a couple of exceptions to this. Firstly, any cookies sent by the scanner *must* be passed through unchanged, as detailed previously. Another case in which we need to preserve the scanner's original request headers is when it tries to inject a payload



through them. To do this, we observe the first incoming requests to learn the scanner's default header values and, for subsequent requests, pass through any headers that had their default values changed. While at this point the scanner's request has been served ⑥, we still need to enrich the response before sending it back to the scanner by leveraging any client-side events and the browser's JavaScript execution engine for triggering the events and recording any meaningful changes to the DOM or any asynchronous requests or redirects.

*Event discovery.* In order to trigger client-side events, we first need to identify which HTML elements have registered event listeners by hooking into them; we achieve this by using jÄk's JS library [213]. We then iterate over all captured events and attempt to trigger them; we expanded XDriver to accurately trigger each event. To detect DOM changes we utilize the MutationObserver API [52] by registering an observer on each document, which returns all changes caused by the last fired event. This is much more efficient compared to prior approaches of constantly scanning the DOM for relevant changes [141]. The DOM changes we consider are new links, forms and iframes, since they can reveal new endpoints of the application to the scanner, and at the same time constitute potential injection points. Similarly, for asynchronous requests, we modified jÄk's library to capture all the information our system needs. We also note that right before triggering each event, we block all asynchronous requests; after capturing the request information we do not send the actual request to the application so as to avoid possible side-effects due to changing the application's state (e.g., logging out or deleting a user). Users can disable our event discovery module with a simple configuration option, when testing web applications that do not make heavy use of JavaScript.

Another aspect we have to take into account during this phase is that elements with events can produce *additional* elements, which can also produce others, make DOM changes and so on. To capture such *nested* events we follow a BFS approach and start by triggering all displayed events when loading the page (i.e., *level zero* events). After triggering each event we inspect whether any new events appeared and store this dependency link between them, thus constructing *event dependency chains*. In addition, if an event hides any level zero events (e.g., due to opening an overlapping menu), we immediately trigger it once again in an effort to make the base event reappear. When we are finished with all of a level's events, we proceed to the next level and repeat the process. It is important to note that for nested events we first check if they actually exist and are displayed in the page in order to trigger them immediately. This is needed since a previous event might have made permanent changes on the page (e.g., a sidebar with further events that remains open throughout the interaction with the page). If, however, the nested event cannot be triggered immediately, we proceed to recursively execute its dependency chain. The recursion is necessary since an intermediary event of the dependency chain might also not exist, thus its own chain should be executed in order to arrive to the final event. All dependency chains are also depicted in the navigation model as individual edges.

Finally, an exhaustive approach would require performing the event discovery for each unique edge we execute. In practice, however, we empirically observed that many similar edges (i.e., same base URL but different query) land on the same or similar pages and include the same or many common events; as such, triggering all events again would be redundant. To address this, for each base URL we store each event along with its unique CSS selector. For ensuing requests to the same base URL with differing parameters we skip any previously encountered events and trigger only the new ones that may exist, thus significantly reducing the event discovery time.

*Inter-state dependencies.* It is common for certain parts of a web application to interact with and affect other parts. For instance, data submitted in a registration form can be reflected in other pages (e.g., the username being shown in the user’s profile). The extraction of such *inter-state dependency* links (ISD) is crucial for discovering certain types of vulnerabilities that are not directly visible on the landing page after the payload injection but appear on other URLs, such as stored XSS. The core idea is to identify if a parameter value for a given POST request (the *ISD source*) appears on another page (the *ISD sink*). Thus, whenever a browser worker executes a POST request, mapped to either a FORM or EVENT edge, it will feed the request’s edge ID, as well as all parameters and their values to the *Background Worker*, which is responsible for detecting such dependencies.

Assuming we have detected such ISD links, when executing a POST request the browser worker collects parameters whose values’ entropy exceeds a threshold to capture the actual scanner payload being tested, which typically includes several different characters and has a greater length. Then, for every parameter it internally fetches ISD sinks associated with it and inspects whether the value truly appears in their source. If so, the value and its encapsulating HTML elements are stored, to be used in the final HTTP response and “carry” the necessary context for the scanner to properly evaluate its injection. We do not store the entire sink’s DOM, as it can significantly increase the response size and corresponding processing time, especially given that a single POST request might be associated with more than one sinks.

*Response enhancement.* At this point, the system needs a way to inform the scanner of newly discovered endpoints or inter-state dependencies. Since our approach is fully scanner-agnostic, we treat each scanner as a black-box that produces HTTP requests and consumes responses. Thus, we need to transcribe each of these newly discovered artifacts into a final, *static* HTTP response, in a way that they are “detected” and leveraged by scanners. To that end, we initially append the ISD sinks’ relevant HTML elements and the DOM changes “as is” to the response (e.g., a dynamically generated form is appended to the document’s body). In contrast, simple asynchronous GET requests and redirects are transcribed as new links, while more composite requests are converted into an equivalent HTML form, with input fields matching those of the HTTP payload, as well as the HTTP method of the asynchronous request. During our analysis, we also observed that modern browsers may

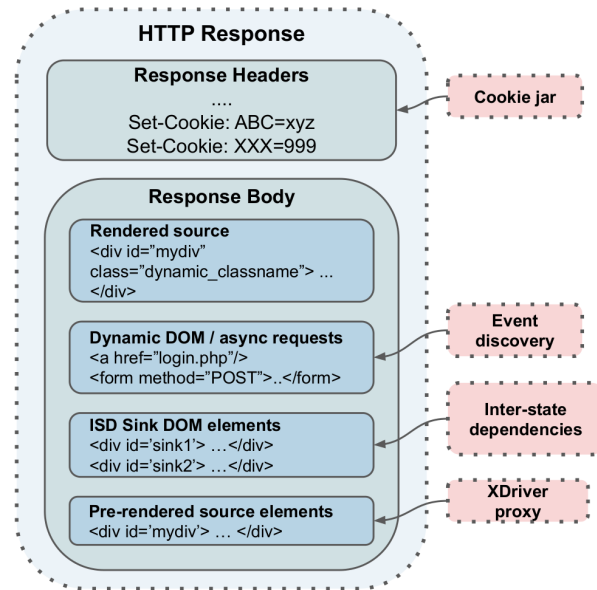


Figure 2.2: Crafted HTTP response.

alter the structure of the original page source in regards to syntax semantics. For instance, if the browser encounters *stray* element attributes, i.e., only an attribute name without a value (which can be part of a scanner’s payload), it will modify their structure, e.g., by appending the = sign and quotes. For example, consider the following scenario, where a form field vulnerable to XSS originally has the form `<input name="username" value="user">`. The scanner payload, trying to escape the value attribute, might look like `abc('xyz`, which will leave the `'xyz` portion as a new, stray attribute inside the element and will be converted by the browser to `'xyz=""` with a leading space, thus preventing the scanner from detecting its otherwise successful injection. In addition, we use BeautifulSoup [71] with the `html5lib` standards-compliant parser [21], to parse and modify the final HTTP response (e.g., to append sink DOMs for ISD), which might further alter the payload structure by rearranging the order of such stray attributes. To overcome this challenge, much like the ISD approach, when submitting values, by appending the *pre-rendered* source’s HTML elements that include the value in the final HTTP response, we allow the scanner to detect and evaluate its injections that may have been altered due to this behavior.

Moreover, the application might have set or unset cookies during the execution of the workflow either with *Set-Cookie* response headers or via JavaScript. Since we capture the response headers for the last executed edge (i.e., the initial scanner request), and several scanners do not include a JavaScript execution environment, we also need to pass this information to the scanner as well. To that end, we iterate over the browser’s cookie jar, compose equivalent *Set-Cookie* headers, and append them in the response so as to inform the scanner of the current state. The worker then proceeds to craft the final HTTP response in-

cluding all relevant information, i.e., the enriched response body, status code and response headers and sends it back to the proxy's request thread waiting for it ⑧. A detailed depiction of how the response is crafted can be seen in Figure 2.2. Subsequently, the worker submits all discovered events, along with all links, forms, iframes and redirects of the page to the *graph worker* so they can be included as new edges in the navigation model ⑦.

**Graph worker.** This component is responsible for interacting with the navigation model, and consists of two separate threads. The first one operates in a read-only manner on the model: it awaits for browser workers' requests for specific edges, constructs and returns the necessary workflow ⑤. The second thread, operating in a write-only fashion, receives newly submitted edges by the workers and adds them to the model ⑦.

**Background worker.** Detecting inter-state dependencies is crucial for detecting certain types of vulnerabilities. The background worker's (BG worker) goal is to detect such dependency links in a timely manner. During its lifetime, the BG worker constantly observes the POST requests executed by the browser workers and identifies ISD sources that might appear in other parts of the application. While in practice other HTTP verbs might be used for state-changing requests as well (e.g. PUT), similarly to prior work [141] we only consider POST as others are not intended to be state-changing at all (e.g., GET [145]). Specifically, it initially keeps track of all submitted values that have an entropy higher than a given threshold. This is a necessary performance/detection trade-off, so as to eliminate commonly seen values (e.g., "1", "true") which would lead to a prohibitive number of candidate ISD sinks being fetched by the browser workers; we empirically found that a threshold of 1.4 effectively eliminates such values. It then queries the graph worker and collects *all* the GET, IFRAME and REDIRECT edges found so far, as they constitute potential ISD sinks. It then proceeds to iteratively fetch each of these edges and inspect whether any of the submitted, higher-entropy values appear in its source. If so, it has detected an ISD sink and immediately notifies the browser workers of the uncovered dependency, so they can fetch it whenever submitting the ISD source.

Before moving on to the next edge, the BG worker submits *all* edges it observes to the graph worker so they can be inspected as well. This is required as certain sinks might appear only *after* submitting the corresponding ISD source (e.g., the URL for editing a comment appears after posting the comment). The BG worker will repeatedly visit all potential sinks as long as there are POST requests executed by the browser workers that have not been fully inspected (indicating that not all potential sinks have been explored). While this approach involves a non-negligible number of additional requests it does not affect the performance, as the BG worker operates concurrently with the browser workers.

We note that a more straightforward approach would be to detect ISD links by leveraging the underlying scanner's requests. For instance, whenever we execute a workflow, we could inspect the landing page for previously submitted values. This, however, creates a strong dependency between our system's effectiveness and the underlying scanner; if the scanner

decides to fuzz a form and does not later visit the corresponding ISD sink, either due to not having discovered it or because it had crawled it earlier, then we would have no chance of detecting the ISD link on time and the underlying vulnerability would be missed. With our approach we can effectively tackle such cases and detect inter-state dependencies on time.

*Input pre-filling.* Another technical challenge is scanners' default behavior when submitting a form. In more detail, when scanners start fuzzing a form, they will typically use any values already filled out by the application itself and will submit their own default values for empty fields. For instance, wapiti sends "default", while ZAP uses "ZAP" as their default values. This behavior, however, can lead to the detection of numerous *candidate* sinks during our ISD detection and can significantly affect its performance, as such values may appear in multiple areas of the application and could stem from multiple ISD sources. To tackle this issue, right before sending the final HTTP response back to the scanner, we iterate over all *empty* input fields in the page and assign a unique value we generate on the fly, with an entropy high enough so it can be detected in actual ISD sinks. This way, scanners will use our unique tokens instead of their defaults and we can precisely map ISD sources to sinks.

**Authentication helper.** A common problem with scanners that execute authenticated scans is that they will submit the valid, user-provided credentials once but will forget them in subsequent authentication requests and instead submit their default credentials. This can occur if the scanner decides to fuzz the login or an account settings form (i.e., change username or password); this is problematic since these default credentials do not correspond to a valid user, thus permanently losing the authentication state for the remainder of the scan. Similarly, scanners do not infer when they are logged out (e.g., due to a malicious HTTP request that the application cannot handle) and continue their (incomplete) scanning irregardless of the authentication state. To overcome these problems, we develop an authentication helper module, leveraged by the browser workers.

*Credential detection.* Initially, the module captures the *first* authentication request (i.e., includes a password in its POST data) and detects the valid user credentials based on common parameters' naming conventions. This is based on the assumption that the first authentication request will include the user-defined, valid credentials, which holds true for all scanners we evaluated. ReScan will then detect these fields in subsequent requests and will overwrite them with the valid credentials, thus ensuring the integrity of the authenticated session.

*Oracle.* Moreover, again on the first authentication request, the module will perform a series of steps to dynamically infer an authentication oracle, conceptually similar to the one proposed in [134]. In more detail, the worker that performed the login will send another request to the landing page without containing *any* cookies, to obtain an unauthenticated response. The module will then check if the detected username (or email) or any logout-related string appears in the authenticated response but not in the unauthenticated one. If so, it has deduced a robust authentication oracle. If they appear in both responses, the

worker will then fetch the page containing the login form and check whether the form appears only in the unauthenticated response. Similar to the first step, if this yields a positive result we have again found the oracle. For each subsequent request, right after the execution of its workflow, we deploy the oracle in a new browser tab so as to check the validity of the established session. The new tab is required so as to maintain the main request's state (e.g., landing page) and execute the remaining components. During our experimental evaluation we observed that the overhead induced by executing the oracle in *every* request is negligible yet significantly increases robustness.

*Relogin.* If after the execution of a request the oracle detects that we have been logged out of the application, the module must attempt to transparently re-establish an authenticated session. This can happen if another concurrent worker triggered a logout and invalidated the session for everyone, or the session broke due to this specific request. However, at this point we cannot be certain which is the cause. In any case, the module will revisit the login URL and try to login with the valid credentials and consult the oracle to verify that the relogin succeeded. If it succeeds, the worker will retry the request at hand to ensure that its workflow was executed properly. If the session breaks again, indicating that this is indeed the faulty request, we relogin and skip the remaining components for this request; otherwise we proceed as normal. Finally, if the relogin fails, we inform all workers to shutdown since we cannot continue the authenticated scan, indicating that either the credentials are no longer valid (e.g., the user was deleted or blocked), or that the application is no longer responsive (e.g., it returns an error message). However, this behavior can be disabled with a simple configuration option, so as to allow the scan to continue regardless of a successful relogin. It is important to note that while the oracle is conceptually inspired by [134], the entirety of the authentication helper module and its capabilities is a new contribution, rooted in the challenges presented by the novel middleware architecture of ReScan.

**False positive & negative elimination.** Another common scanner limitation is their susceptibility to false positives (FPs) and negatives (FNs), especially when testing for XSS vulnerabilities. This is due to the fact that scanners often verify the success of their injections based solely on the payload's existence in the HTTP response, without verifying if the payload was actually executed. Therefore, it is clear that a payload that appears "as is" but was never executed (e.g., due to it being part of an input element's value attribute) will lead to a FP. Similarly, even if the payload is executed, but its structure is altered or completely removed by client-side code, the scanner will not be able to detect the successful injection, leading to a FN. Due to our black-box approach, we cannot tamper with each scanners' internals and tackle this; we can, however, eliminate or reduce such cases by providing the user with additional results as a separate report, which can be intersected with the scanner's reported vulnerabilities. To achieve this, we employ the following approach.

First off, for each incoming request, we try to identify if it is an injection attempt by collecting all query parameters for GET requests and payload parameters for POST requests, and

check their values against common keywords used among scanners (e.g., `alert`, `prompt`, `javascript:`). The intuition is that regardless of the payload structure, most scanners will attempt to trigger an alert box with their injected code. Then, whenever an alert box opens throughout the rest of the scan, we extract its text and try to match it against all detected injections in the previous step. If the alert text does appear inside any of the injection values, we can be certain that the detected XSS is a true positive, since it leads to code execution. If it does not match any of the injection attempts, it is discarded, as it is a legitimate alert by the application. It is important to note that the effectiveness of this technique relies on the underlying vulnerability scanner.

For scanners that do *not* reuse payloads, i.e., do not alert the same value for different injection points (e.g., Wapiti), then all FPs and FNs can be eliminated as each alert box can be exactly mapped to one injected value. If the scanner does reuse payloads (e.g., ZAP always tries to execute `alert(1)`) then any alert box that occurs will be mapped to all attempted injections so far. Thus, we know there is *at least* one XSS vulnerability up to that point, but we cannot be certain which parameter is vulnerable, nor if there are more than one. For such cases, we simply inform the user that the confidence level for the legitimacy of the reported vulnerabilities is lower and they should account for possible FPs reported by the scanner. Moreover, if there are no alerts throughout the scan, indicating that no XSS vulnerabilities were triggered, then any reported vulnerabilities by the scanner can be safely considered as FPs.

### 2.2.3 URL Clustering

All the enhancement techniques we have devised so far aim to improve scanners' code coverage and vulnerability detection capabilities. However, during our empirical analysis we identified an orthogonal limitation which affects scanners' performance. We observed that scanners are not able to identify *similar URLs*, which may include the same or related content but essentially offer the exact same functionality. For instance, if a web application includes a series of product URLs of the form `/products.php?pid=X`, the scanner will crawl *and* audit each URL separately, which can eventually lead to the detection of other similar URLs, specific to each product (e.g., `/products.php?pid=X&action=edit`). Clearly this behavior impacts scanners' performance, as they spend resources and time repeatedly testing the same functionality.

Additionally, while some scanners offer some sort of control mechanism to limit this behavior, they are coarse-grained approaches that require careful configuration by the end user and can also wrongfully skip URLs. For instance, Wapiti offers an option to limit the crawl's depth, which can result in skipping important URLs that appear in deeper levels of the application while similar URLs that are at the same depth will be redundantly crawled. w3af provides an option to limit the number of URLs with the same path and set of query

parameters that will be considered during the scan. While this can decrease the number of discovered similar URLs, it requires careful examination and proper configuration. More importantly, some URLs with the same query parameters might need a different threshold than others; for example setting the threshold to 1 would work for `/products.php?pid=X` URLs, as more than a single product page would be redundant, but for URLs of the form `/products.php?pid=X&action=Y`, where `action` can be one of `edit`, `update`, `delete`, a value of 1 would cause the scanner to consider only one of these functionalities, thus directly affecting its coverage and, potentially, the detected vulnerabilities.

It is worth mentioning that prior work on web application scanning has proposed page clustering algorithms [132, 213]; however, their approach relied solely on a page's link structure or did not consider URLs' query parameter values. Regarding the link structure, while such a strategy may have been adequate at the time, modern web applications utilize new methods for navigation and different functionalities (e.g., stray input or button elements that do not belong to a form) which are driven by client-side events. Ignoring these can lead to the incorrect clustering of different pages. On the other hand, ignoring URL parameter values can lead to subtle but important inconsistencies. For example, consider two product pages, with `pid=1` and `pid=2` and their corresponding sub-URLs for editing each one, `/products.php?pid=X&action=edit`. When the scanner encounters the product URLs, it deems them to be similar and clusters them under `pid=1`. When it visits the editing pages, if it does not take into account the `pid` value it used previously, it can end up clustering them under `/products.php?pid=2&action=edit`. Thus, if a vulnerability stems from editing a product's field and is reflected in that product's page, the scanner will miss it as it will edit the second product but inspect the page of the first one.

To overcome these limitations, we design an advanced URL clustering algorithm that aims to cluster similar URLs in real time when requested by the scanner and prevent it from ever learning the existence of redundant URLs, *without* requiring any specific configuration by the user.

**Page similarity.** The first requirement for our algorithm is to be able to *accurately* and *efficiently* identify whether two URLs and their respective DOMs are in fact similar and should be clustered together. We consider two URLs to be similar if they share the same path, regardless of the URL query parameters. If they are not similar they are not clustered together; if they are, we still have to investigate whether their DOMs are similar to decide if they should be clustered or not. To achieve this, we build upon the *normalized DOM-edit distance* metric (NDD), proposed by Vissers et al. [271], which essentially takes as input two DOM trees and computes the number of edit operations required to move from one tree to the other. Initially, we observed that the tree edit distance algorithm they used at the time (ZSS [277]) was rather slow even for relatively simple DOMs. Since we need to compute DOM similarity immediately when a page is requested by the scanner, we cannot afford such a performance penalty. Thus, we decided to replace the tree edit distance algorithm



with current state-of-the-art, namely *APTED*, proposed by Pawlik et al. [210, 211], which offers a significant speed up. However, in certain cases of more complex DOMs, even this algorithm had a prohibitive processing time for our use case.

Since designing a more efficient tree edit distance algorithm is out of scope of this work, the only way to reduce the processing time is to reduce the size of the algorithm's input, i.e., the DOMs' sizes. While [271] adopted a horizontal pruning approach, discarding all tree nodes below a level of five, such an approach would not be suitable in our case. This could lead to subtle but important differences that indicate different functionality between the two pages (e.g., a form residing in deeper levels of the DOM) not being considered, leading to pages being incorrectly clustered together.

Thus, we devise our own fine-grained pruning methodology. When constructing the corresponding trees from the pages' DOMs we recursively discard leaf nodes that do not offer any sort of functionality (e.g., line breaks, paragraph, span, div or font formatting tags), while maintaining nodes that denote functionality (e.g., scripts, forms, iframes, buttons and inputs). If a node has all of its children removed, becomes a leaf node and is a non-functional tag, it is discarded as well. With this approach, we significantly reduce the tree size and thus the processing time required for the NDD computation, while maintaining the important parts of the tree structure that we should consider when deciding whether two pages should be clustered.

Moreover, this approach also helps us avoid not clustering similar pages together due to insignificant differences. For instance, consider two article pages that should be clustered together as they offer the exact same functionality and one of them includes a list of comments, while the other does not have any yet; this would lead to *not* clustering these pages together due to these (irrelevant) nodes. It is important to stress at this point, that two pages that do not include any functionality denoting nodes (e.g., two static pages) would be oversimplified and incorrectly clustered together; however, we employ our NDD variant *only* for pages that also share a similar URL, thus avoiding clustering together irrelevant pages. We refer to our NDD variant as *mNDD*. If the *mNDD* between two DOMs is lower than a predefined threshold, which we identified experimentally, they are considered similar.

**Algorithm.** We use an example application that includes a set of similar and different pages under the same base URL to illustrate our URL clustering algorithm. Assume an application that lists two different products pages, with URLs of the form `/products.php?pid=X` and `pid` denoting each product's unique ID (either 1 or 2). For the remainder of this section, all example query strings will refer to the `products.php` page, which we omit for brevity. These pages include links to product-specific pages (`?pid=X&action=Y`) that offer different functionalities. The `action` parameter can be `edit`, `review` or `add`, and leads to a page that has a form for editing, reviewing, or adding the product to the cart, respectively.

At a high level, the core idea of the URL clustering algorithm is to prevent scanners from learning the existence of redundant URLs that point to similar pages and offer the same

functionality. In this example, the scanner should either learn (i.e., receive a regular response) for `pid=1` or `pid=2` but not both. Moreover, the algorithm must ensure that the scanner will learn all other functionalities and sub-URLs for *the same product ID only*. For instance, if the scanner initially learns `?pid=1`, then it should also learn all actions for that product ID only. This is required since executing such a functionality will most likely have an effect only on that product's page. Thus, the algorithm needs to keep track of the different URL parameters and their values for which the scanner got a response. Finally, it is crucial that the algorithm also accounts for the arbitrary order the scanner might request such URLs; e.g., it is not restrained to requesting the `?pid=X` URLs first and then moving on to their specific sub-URLs. Having defined the algorithm's goals and an example scenario, we next detail the specifics of our URL clustering algorithm.

Firstly, the algorithm considers only GET requests that include query parameters. We do not include other types of requests (e.g., POST) as they do not carry information that is useful for the clustering process and would only lead to longer running times. The first time the scanner requests one of these URLs it will get a regular response and we store all parameters and their values that it learned. For subsequent requests towards the same base URL with differing query parameters (either with a different set of parameters or different values or both) the algorithm will perform the following steps:

1. Collect all parameters that we have seen before but have a *different* value than the one(s) the scanner has learned. We represent these parameters with set  $\{X\}$ , and another empty set with  $\{X'\}$ .
2. If there are any new parameters that the scanner has not requested before, leave them "as is" and store their values.
3. Swap the unknown values for the  $\{X \setminus X'\}$  parameters (the difference of the two sets) with those previously learned by the scanner (initially all values are unknown).
4. Internally fetch the swapped and original URLs (if not already fetched) and compute the mNDD between them.
5. For similar pages generate a clustering rule (described below) and stop.
6. If the pages are *not* similar: (i) reset  $\{X'\}$ , (ii) pick a parameter to preserve its original value, (iii) add it to  $\{X'\}$ , and (iv) go to (3). If no more single parameters remain to preserve their value, iterate through all combinations of two parameters, then combinations of three, and so on.
7. If there are no more parameters to swap, stop, as this is indeed a new page and should be served normally. Store the original URL's parameter values for subsequent comparison and clustering decisions.

Assuming the scanner initially requested and learned `?pid=1`, we next provide a series of cases based on our example scenario to demonstrate our algorithm's correctness and to also describe the generated clustering rules and how they are applied. If `?pid=2` is requested next, since the `pid` parameter has already been seen before but with a different value (step

1), it will be swapped with the known value 1. Then both URLs will be fetched internally, their mNDD score will be computed and they will be found similar, and the clustering rule shown in Listing 2.1 will be generated (steps 1-5).

```
{  "pid" : "*",
  "redirect" : "?pid=1"}
```

Listing 2.1: Example of a simple clustering rule.

Essentially, this rule indicates that any incoming request that only has the pid parameter, regardless of its value, should be redirected to ?pid=1, thus effectively preventing the scanner from ever getting a response for the other product. This is achieved by sending a crafted HTTP response back to the scanner, with a 301 status code and the Location header set to the appropriate URL. The only exception is when the scanner requests ?pid=1, which will be served as normal so as to obtain a fresh response for that product's page.

If the scanner requests ?pid=2&action=edit next, we have a newly seen parameter (i.e., action) which is not tampered with throughout the process (step 2). The known pid parameter will be swapped and processed like before and the scanner will internally fetch the original URL and also ?pid=1&action=edit, compare them using mNDD and infer that they are again similar, setting the rule shown in Listing 2.2.

```
{  "pid" : "*",
  "action" : "edit",
  "redirect" : "?pid=1&action=edit" }
```

Listing 2.2: Example of a subsequent clustering rule.

It is important to note, that the scanner might not have requested or even seen the swapped URL before; this, however, does not affect the algorithm's operation, as it can *proactively* infer its existence by considering the previously seen parameters and their values. It is also necessary, in order to ensure that the scanner will learn the editing functionality for the same product it learned before.

If the scanner then requests ?pid=1&action=review, the pid parameter already includes a known value and is therefore left as-is. The action parameter, while known, has a different unknown value compared to before. Thus, it will be swapped with the value edit, the two URLs will be compared and the algorithm will infer that they are in fact different pages, since their mNDD score is higher than the threshold. This leads to not setting any new rules and the originally requested URL should be served normally (step 7) while storing the newly seen value for the action parameter.

Finally, in a more complex case, if ?pid=2&action=add is requested, both parameters are known but yet contain values that the scanner has not learned. As a result, all parameters will be initially swapped and the URL ?pid=1&action=edit will be fetched and compared. Since it leads to a truly different page, no rules will be generated and one of

the parameters will be picked randomly (e.g., `pid`) to preserve its original value (step 6). The swapped URL that occurs is `?pid=2&action=edit`, which also leads to a different page. Next, the last remaining parameter will maintain its value while swapping the other one (`?pid=1&action=add`). This indeed lands on a similar page as the original URL and a rule similar to Listing 2.2 will be generated, for the add functionality. Even if these requests occurred in a completely different order, since scanners can prioritize them differently (e.g., based on when the URLs were discovered in the application during the crawl), the algorithm would end up inferring the exact same clustering rules, with a slightly different order and steps, depending on the known parameters and their values at the time of processing.

At this point, it is worth noting that a large number of URL parameters could *potentially* lead to a state explosion or a prohibitive processing time, due to the numerous parameter combinations that would have to be tested. In practice, however, this is a highly unlikely scenario due to several reasons. First, since the algorithm initially ignores newly seen parameters and parameters with known values, it would not have to test and compare *all* combinations. In addition, for that to happen all combinations would need to lead to different pages, as the algorithm stops if it finds a pair of similar pages. Finally, we have not come across any such case during our experimental evaluation. In §2.3.2 we experimentally measure the performance gain our algorithm offers, and also evaluate its correctness in terms of achieved code coverage and discovered vulnerabilities.

**Implementation details.** Our URL clustering functionality can be enabled with a simple command line parameter; no further configuration is needed. Initially, the Orchestrator inspects the scanner requests and locates each GET that includes a query. It then checks if any of the generated rules applies to the URL; if so, it immediately crafts an HTTP redirect (9 in Figure 2.1) and sends it to the request's proxy thread, which eventually sends it back to the scanner. If no rules apply to the URL (initially because none have been generated) it marks the request as pending and passes it to the clustering module. It then moves on to subsequent requests that need to be served, so as not to remain idle while pending requests are processed, and periodically checks if a new rule has been generated and applies to the pending URL or whether the request should be passed on to the browser workers.

The clustering module is comprised of a configurable number of threads that handle pending requests concurrently. When a request arrives, a thread reserves it and runs the algorithm to decide whether to serve it normally or infer clustering rules. These threads do not maintain their own browser instances, as it would be too expensive, but need a way to fetch the original and swapped URLs that occur during their operation. To that end, we leverage the BG Worker, which apart from ISD-detection also serves these requests by the mNDD threads. It also caches the responses in case the same URL is requested later on. The BG worker constantly checks requests, as they are issued from the scanner and need to be served as soon as possible.

### 2.2.4 API Abstraction for Future Scanners

As aforementioned, while ReScan can be transparently leveraged by any scanner as a black-box middleware, *future* scanners could greatly benefit from the ability to access ReScan’s internal knowledge and alter its behavior based on their runtime needs. As such, in order to unleash ReScan’s full potential and enable such a symbiotic interaction, we design and implement an abstraction layer in the form of an API. A scanner opting to use the API can request access to ReScan’s internal data, such as the entire app navigation model, detected ISDs, discovered XSS and more. Moreover, it can alter ReScan’s behavior, by enabling or disabling any module at runtime, e.g., disabling ISD detection and sink collection when testing for vulnerabilities that do not have ISD effects. We detail the various API endpoints in Appendix A.2.

## 2.3 Experimental Evaluation

**Experimental setup.** For our system’s evaluation, we use state-of-the-art vulnerability scanners that have seen wide adoption, allowing us to perform a direct comparison. Specifically, we evaluate ReScan on w3af [222], wapiti [259], Enemy of the State [132] and ZAP [89], which have been extensively evaluated by prior work [132, 133, 138, 141, 213]. We refrain from evaluating simple crawlers, e.g., wget [204], CRAWLJAX [197], even though they could benefit from ReScan, since our main goal is to enhance vulnerability scanners and measure both their coverage *and* detection capabilities. w3af and wapiti serve as a benchmark for more traditional vulnerability scanners as they mainly use raw HTTP requests and feature from minimal to no-Javascript execution engines, while ZAP has more advanced capabilities that are better suited for modern scanning requirements. Enemy of the state is a popular state-of-the-art academic scanner, which also introduced the concept of modelling application states. To be able to evaluate Enemy, we had to make some slight modifications so as to proxy all of its traffic through ReScan; its core functionality was left as-is. Finally, it is worth mentioning that we also attempted to setup jÄk [213] for our evaluation, but were unable to do so, due to the use of certain outdated packages that prevented it from executing properly. We contacted the authors to aid us in the setup process, but did not receive a response.

To obtain better code and functionality coverage in the tested applications, we run authenticated scans by configuring each scanner to log into them. For ReScan’s configuration, we enabled four headless Chrome browser workers and all enhancement techniques described in §2.2 (i.e., ISD detection, event triggering, as well as the authentication helper and URL clustering modules). For the event discovery process we consider the events by [141], (i.e., *(on)input*, *onchange* and *compositionstart*) and also extend them to include another set of prominent events that can trigger requests and cause DOM changes, namely *(on)click* and *(on)submit*.

When running the standalone scanners without ReScan, we enabled the audit plugins both for reflected and stored XSS. However, when evaluating ReScan we disabled the stored XSS plugins, as we rely on our ISD module as a substitute. Similarly, we enabled the AJAX spider plugin for ZAP when running without our system, but disabled it with ReScan, due to our event discovery module. These configuration changes however, do not work in favor of ReScan; on the contrary, by disabling modules we can only limit our coverage and detected vulnerabilities. Despite that risk, and in favor of reducing redundant operations as well as proving our approaches' practicality, we opt to rely on our own techniques. We also set the HTTP timeout for all scanners to 999 seconds. Finally, to avoid long lasting scans that may occur for more complex applications and complete our evaluation in a reasonable time-frame, we set each scanner's maximum scan time to one day. It is important to stress that apart from the aforementioned configuration options, all scanners and applications were configured in the *exact same way* when running with and without ReScan. We provide more details on the specific configuration options in Appendix A.3.

In Table 2.2 we list the web applications and the specific versions we used during our evaluation. We opted to use the same set of applications as [141], since it includes both legacy and intentionally vulnerable apps as well as modern, widely used applications. In addition, using the same applications allows for direct comparison. We created an individual Docker container for each application, allowing us to reset it back to a clean state after each scan; all containers have been released to facilitate further research in the field [59]. We also enabled XDebug [221] in each application for capturing precise coverage information in terms of unique lines of code (LoC) executed during each scan. Finally, we note that for each application we excluded any URLs that might affect the application's correct deployment (e.g., user deletion functionalities, version upgrading), as done in prior work as well (e.g., [137]) In more detail, we initially identified common URLs manually. Then, during our test runs our authentication helper module allowed us to identify more URLs (i.e., when being logged out and not able to re-login). Inspecting the traces showed that some functionality broke the app or the user was disabled/blocked. While such endpoints might also suffer from vulnerabilities, they pose a risk to correctly auditing other (and usually more) functionalities. In practice a separate scan should be performed for those endpoints. This is inherent to black-box scanning and is not a limitation of ReScan; it is rather a matter of correct scanner configuration.

Experiments were done on a commodity desktop with an 8-core Intel Core i7-4790 CPU 3.60GHz and 12 GB of RAM.

**Discovered vulnerabilities.** Table 2.2 details the results of our evaluation; we manually verified every discovered vulnerability and report on the true positives. To deduplicate scanners' results and provide a fair comparison, we cluster vulnerabilities following the same approach as prior work [141]. Regarding false positives, we found that wapiti reports only one, while ZAP is the scanner most prone to FPs as it reports 16 FPs across all apps by itself and

Table 2.2: Number and type of unique vulnerabilities discovered by each scanner without (left) and with ReScan (right) for each app.

Scanner Vulnerability	w3af		wapiti		Enemy		ZAP	
	R-XSS	S-XSS	R-XSS	S-XSS	R-XSS	S-XSS	R-XSS	S-XSS
SCARF (2007)	-/-	4/8	-/-	3/7	-/-	-/4	-/-	3/6
WackoPicko (-)	1/2	-/1	2/3	1/1	2/2	1/1	2/2	1/1
Wordpress (5.1)	-/-	-/1	-/1	-/1*	-/-	-/-	-/1	-/1*
osCommerce (2.3.4.1)	-/2	-/2	3/3	5/16	-/-	-/-	-/-	2/2
Vanilla (2.0.17)	-/-	-/1	-/-	-/1	-/-	-/-	-/-	-/1
PhpBB (2.0.23)	-/-	-/-	-/-	-/2 <sup>†</sup>	-/-	-/-	-/-	-/4 <sup>†</sup>
Prestashop (1.7.5.1)	-/1*	-/-	-/1*	-/-	-/-	-/-	-/1*	-/-
Joomla (3.9.6)	-/-	-/-	-/-	-/-	-/-	-/-	-/-	-/-
Drupal (8.6.15)	-/-	-/-	-/-	-/-	-/-	-/-	-/-	-/-
HotCRP (2.102)	-/1	-/-	-/1	-/-	-/-	-/-	-/-	-/-
Total	1/6	4/13	5/9	9/28	2/2	1/5	2/4	6/15

\* The scanner was able to identify the vulnerability *only* with ReScan, but not during the maximum scan time.

<sup>†</sup> One of the vulnerabilities was found in a URL that broke the app and was eventually excluded.

20 with ReScan. This increase is expected, as the scanner audits a larger area of the application when enhanced by our system. Nonetheless, ReScan is able to identify these injections as *potential* FPs due to ZAP reusing the same payload. Regarding true positives, in most cases ReScan effectively enhances the underlying scanner and facilitates the detection of more vulnerabilities, both for reflected and stored XSS. We also observe that the detection capabilities improve both for benchmark, and more recent applications. When considering the aggregated results per scanner for all applications, we find that w3af reports five more reflected XSS with ReScan and nine additional stored XSS. Moreover, wapiti located four more reflected and another 19 stored injections, while ZAP has an improvement of two and nine additional flaws respectively. Enemy of the State exhibited the least improvement but still located four additional stored XSS; this highlights that while ReScan is naturally dependent on the underlying scanner’s capabilities, it can still effectively facilitate vulnerability detection. Overall, the standalone scanners reported six *unique* reflected and 13 stored XSS among all applications, while with ReScan they reported 10 reflected and 34 stored XSS respectively. Even in the few cases where no additional flaws were detected, the presence of ReScan does not negatively affect scanning as the same vulnerabilities were detected both with and without our system. Detecting the same flaws does not necessarily mean that the *standalone* scanner has detected all endpoints of the application or that it has sufficiently tested it; it might simply mean that while ReScan covers a larger area of the application, no other vulnerabilities exist for it to detect. To uncover more insights, we need to examine the code coverage that was achieved in each case.

**Code coverage.** Table 2.3 shows the precise coverage achieved by our system, as unique LoC executed on the server-side during the scan, and compares it to the coverage of each individual scanner. ReScan achieves better coverage in all cases and offers an improve-

Table 2.3: Total lines of code (LoC) executed by ReScan (R), the standalone scanner (S), and common to both of them ( $R \cap S$ ).

App / Scanner	w3af			wapiti			Enemy			ZAP		
	R	$R \cap S$	S	R	$R \cap S$	S	R	$R \cap S$	S	R	$R \cap S$	S
SCARF	662	533	548	659	596	611	623	261	288	613	578	599
WackoPicko	1,009	888	907	911	692	710	873	433	452	819	684	784
Wordpress	51,612	30,779	30,805	53,974	30,862	31,134	43,731	28,908	29,266	54,329	33,514	34,484
osCommerce	7,056	2,066	2,074	7,179	6,947	7,140	5,194	2,067	2,067	7,270	6,247	6,925
Vanilla	12,247	8,073	8,137	12,138	7,936	8,717	12,404	2,477	2,479	12,951	8,774	9,568
PhpBB	9,803	2,321	2,330	9,942	3,069	3,091	8,225	6,780	7,018	10,487	4,816	5,259
Prestashop	93,361	14,544	14,709	96,712	14,916	14,926	28,209	19,062	19,062	103,955	10,043	10,409
Joomla	43,094	14,822	14,895	54,048	16,505	17,476	20,113	15,527	15,876	54,711	15,448	16,149
Drupal	80,195	26,251	28,655	80,620	23,290	25,105	70,998	59,998	68,236	74,428	28,272	30,291
HotCRP	19,109	8,772	8,777	17,737	10,517	11,415	17,063	14,871	14,918	15,647	5,463	5,509

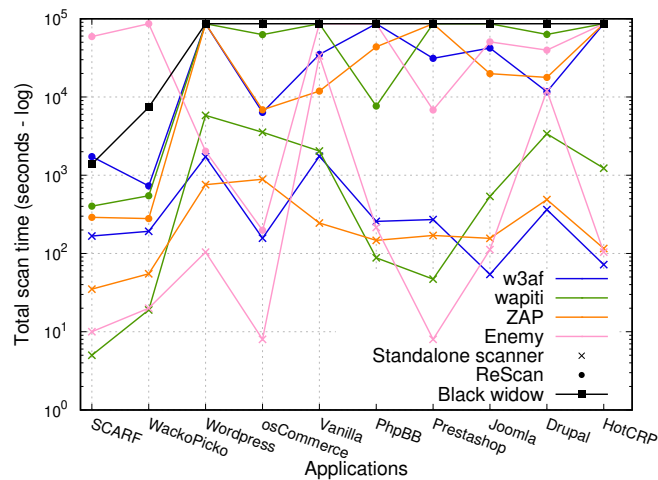


Figure 2.3: Total scan time in seconds for each app/scanner pair with and without ReScan.

ment of at least 3% and at most 935%, with an average of 168%. To validate the quality of the increased coverage, we manually sampled and inspected LoC executed only by ReScan and found that in many cases ReScan-enabled runs reached and tested critical functionality that the standalone scanners did not. Indicatively, ZAP could not reach the categories' editing functionality in osCommerce, while none of the scanners could post and read draft discussions in the Vanilla app; both cases led to missing XSS flaws. We also observe that vanilla scanners reach some LoC which are not executed by ReScan. After inspecting these cases, we found that they belong to unauthenticated parts of the application, indicating that the scanner was logged out and continued as such. ReScan's authentication helper module ensured that the scanner remained authenticated throughout the scan, as intended.

**Performance analysis.** In Figure 2.3 we present the total time required to scan each application both with and without ReScan. The overhead induced by our system is considerable; however this is expected due to the numerous enhancement techniques we apply for each and every intercepted request and the fact that a full-fledged browsing environment is leveraged. In addition, the maximum scan time of one day was reached by all scanners for



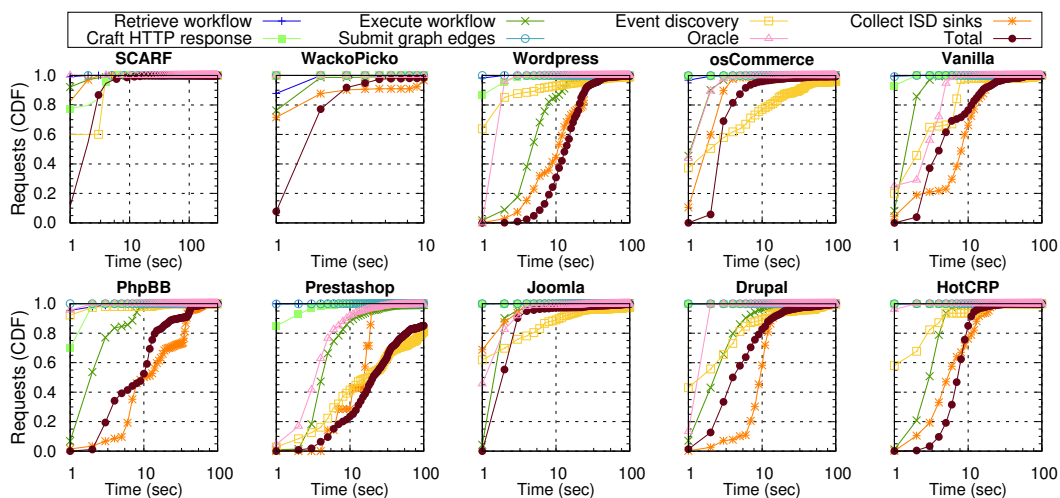


Figure 2.4: Requests' CDF per application, in terms of total as well as individual components' processing time.

one of the applications (HotCRP) while in total, 15 of the 40 ReScan-enabled runs reached this limit. In Figure 2.4, we show the total processing time required for each request handled by ReScan per application and per individual component. We note that each CDF has been calculated using the aggregated requests from *all* scanners for that application. This is due to the fact that the time required to handle each request is irrelevant to the scanner that initiated it, but heavily depends on the application's characteristics (e.g. usage and number of Javascript events, number of detected ISD sinks). Additionally, the totals that were used to calculate the different components' CDF differ, as not all of them are executed on each request. For instance, retrieving and executing a workflow is applicable to all requests. However, triggering events is only relevant to pages that include them and have not been explored before, and collecting ISD sinks is only relevant to edges for which we have detected them. Most notably, event discovery can be quite expensive for applications that heavily rely on Javascript and client-side events, i.e., on average it takes one to three seconds for nine of the apps, while Prestashop required 19 seconds. Similarly, fetching ISD sinks on average took less than two seconds for five applications, while in the worst cases (Wordpress, Prestashop) it took 11 and 16 seconds respectively. Other system components, such as fetching and executing the workflow, constructing the navigation model and crafting the final HTTP response introduce negligible overhead in most cases, i.e., less than a second. Interestingly, while executing the crucially-important authentication oracle after every request might seem costly, our analysis showed that it only takes up to two seconds for 99% of requests. Overall, each request can be completed on average within three seconds for four applications and five seconds for another two, while Prestashop generally has slower response

times and can take up to 21 seconds. In summary, while the performance overhead is non-negligible when compared to the standalone scanners, the significant improvements both in code coverage and vulnerability detection render this a viable and acceptable trade-off.

**Prominent use cases.** Next we outline interesting use cases that highlight the benefits of using our framework.

*Vanilla FP.* When wapiti scanned the Vanilla forums with ReScan it *incorrectly* reported an XSS vulnerability due to the payload appearing inside a `textarea` element (i.e., was not executed). However, while ReScan detected the injection attempt, due to our FP elimination mechanism it *correctly* did not report a vulnerability.

*Wordpress FN.* One vulnerability is a reflected XSS stemming from the submission of a vulnerable field in an AJAX request. The AJAX response is then reflected in the same page, the payload is executed and is then dynamically removed from the DOM; thus neither wapiti, nor ZAP are able to detect their otherwise successful injection. Since ReScan is agnostic to the presence and structure of the attempted payloads, instead relying on code execution, it is able to detect and report this missed vulnerability. This clearly highlights the need for dynamic vulnerability verification. Moreover, we note that this AJAX call requires a valid CSRF token, demonstrating the importance of our correct workflow execution.

*Vanilla ISD detection.* The Vanilla forums' vulnerability is a stored XSS that occurs when saving a new discussion as a draft, and is triggered when viewing the author's drafts. However, the sink of the injection (`/drafts`) is not visible anywhere on the application *before* saving the first draft and is only added to the home page afterwards. Therefore, the scanner would need to re-visit the home page and actively search for new URLs and visit them, in order for ReScan to discover the ISD link, all while before the scanner started fuzzing the source. This highlights the practicality of the background worker, which operates independently of the active scan and attempts to find such links before the scanner starts fuzzing the corresponding ISD source.

### 2.3.1 Other Vulnerabilities

While our evaluation focused on XSS, as they are the most prevalent bug among our applications and also allow for a direct comparison with recent work [141], ReScan aims to support any vulnerability type that scanners might test for. To that end, we conducted another set of experiments, where we picked known vulnerabilities from our applications and re-configured the scanners to use the corresponding auditing plugins, to assess whether our system can effectively facilitate their detection.

**File upload.** osCommerce suffers from an unrestricted file upload vulnerability [97], in the image-upload functionality for a new product category in the administration panel. Specifically, while the `.htaccess` file in the upload directory attempts to prevent access for a number of executable files, it does not prevent *all* of them. To uncover this vulnerability,

we configured w3af (with ReScan present) to use its `file_upload` plugin and pointed it to the vulnerable upload form, which led to the successful upload of an executable file and was detected by the scanner. w3af without ReScan, however, can never reach the upload form as it cannot authenticate in the application, which also justifies its rather low coverage in Table 2.3. While the root cause for w3af missing the vulnerability is not specific to the mechanics behind the vulnerability itself, this highlights the fact that having the necessary payload to trigger a vulnerability is only one aspect crucial to a scanner’s success.

**Login brute-forcing.** For brute-forcing weak account credentials we opted to use Prestashop, due to its irregular login form functionality. In more detail, while logins are carried out through a regular HTML form with its `action` attribute set to the login page’s URL, upon submission the form sends a request to a different endpoint; as such the scanner has no way of discovering it and successfully logging in. ReScan, correctly submits the form and follows all redirections, and manages to log into the application. To that end, we enabled w3af’s `form_auth` brute-force plugin and pointed it to the administrator’s login page. As expected, w3af by itself was unable to detect the correct credentials, while with ReScan it detected and reported the vulnerability.

**Blind SQL injection.** Since the applications in our dataset did not include any non-trivial SQL injection vulnerabilities (SQLi), we opted to install a vulnerable Wordpress plugin, namely GB Gallery Slideshow v1.5 [4]. One of the HTML forms generated by the plugin has a registered `onsubmit` event, which overrides the form’s default functionality by sending an AJAX request with a completely different payload. One of the AJAX parameters is vulnerable to a blind SQLi [72], where successful exploitation is not directly visible on the landing page but is inferred based on the time required to get a response. We configured Wapiti to leverage its `blindsqli` module and pointed it to the page containing the vulnerable form. As expected, wapiti without ReScan was able to fuzz the form’s default structure but could not uncover the actual AJAX request that is sent when *correctly* submitting the form. In contrast, ReScan identified the AJAX request through its event discovery process and transcribed it to a static HTML form in the HTTP response so Wapiti could identify it too. Later, the scanner fuzzed the correct request, which was replayed by ReScan through the navigation model, leading to the detection of the vulnerability.

Overall, while certain ReScan components are tailored towards specific types of vulnerabilities, such as the ISD detection and FP/FN elimination for (stored) XSS, the remaining components effectively facilitate the detection of other types of vulnerabilities as they are not tied to the vulnerability itself. For instance, realistic rendering and interaction through a SotA browser, event discovery, correct workflow execution and maintaining the authenticated state are of crucial importance for two main reasons. First, all these aid scanners in discovering further application endpoints and functionalities to audit. Second, they are necessary for properly executing certain functionalities, which is a necessary prerequisite for the scanner to be able to correctly test its payloads and deploy its fuzzing strategy regard-

less of the vulnerability type being tested. As such, our system can effectively be applied to a large class of different vulnerabilities; by open-sourcing our code, we hope to facilitate other researchers in the field of black-box web application testing.

### 2.3.2 URL Clustering

It is critical to ensure that our URL clustering algorithm does not result in scanners missing relevant parts of the application.

**mNDD threshold.** To identify the optimal threshold for our mNDD metric, we performed the following experiment. For each of the applications, we manually compiled three sets of pages. The first set included pages with completely different URLs and functionalities which should *not* be clustered together, while the second set included pages with similar URLs *and* functionalities that should be clustered. Finally, the third set incorporated pages that had similar URLs, but should *not* be clustered due to different functionalities. We then proceeded to compute the mNDD score for each pair of pages within each set, where a higher value denotes different pages and a lower value indicates some similarity. For the pages that should not be clustered (1st and 3rd set), the minimum mNDD value was 0.014, setting an upper bound for the threshold. For the pages that should be grouped together, the maximum mNDD value was 0.009, indicating a lower bound for the threshold. As there is no overlap between the range of values for different and similar pages, we opt to use the lower of the two as our page similarity threshold (i.e., 0.009) to ensure that we avoid false positives where different pages that happen to have an mNDD slightly less than 0.014 are clustered. During this process we encountered a single false positive, in osCommerce, where two different pages were incorrectly clustered. This was due to the fact that the two pages were identical in structure even though they had different functionalities (adding an item and editing an existing item respectively). However, it is important to note, that this false positive is not limited to our mNDD approach, as the tree edit distance value is 0 for the regular NDD as well. We also note that while this threshold might not work for all applications, it is well-suited for most cases as our empirical analysis was conducted on a dataset that incorporates a diverse set of applications.

**Correctness.** To assess the correctness of our algorithm, we relied on the clustering rules that were set for each application during our main evaluation runs. Specifically, we inspected the different parameters that were clustered and proceeded to visit the corresponding pages both with the values that the scanner requested but were redirected, as well as the final redirection value too. We then observed whether the pages were in fact similar to each other or if they were incorrectly clustered together. Among all applications, this process yielded two cases of false positives. As expected, the first case was the aforementioned issue with osCommerce described during our similarity threshold experiment. The second case was for PhpBB, but only for w3af's scan. After analyzing the cause for this false positive,

Table 2.4: Qualitative differences between ReScan and Black Widow.

Feature / System	Black widow	ReScan
<b>Browser support</b>	●	●
<b>Navigation model</b>	●	●
<b>Inter-state dependencies</b>	●	●
<b>Event triggering</b>	◐	●
- Handle XHR payloads	○	●
<b>Authentication helper</b>	◐	●
- Detect/configure credentials	○	●
- Dynamic state oracle	○	●
- Re-login	◐	●
- Retry failed edges	○	●
<b>URL clustering</b>	○	●
<b>Concurrent workers</b>	○	●

we deduced that it was not caused by our algorithm or the mNDD metric, but instead, was caused due to w3af’s inability to maintain an authenticated session in PhpBB, even with ReScan. In more detail, PhpBB uses a randomly generated *sid* URL parameter in all administrator URLs and also sets the same value in a cookie. After logging in, w3af sent another request *without* cookies, ReScan’s authentication helper re-established the session and the scanner ended up with two different values for *sid* and the cookie, effectively creating a mismatch between them and leading to unauthenticated responses. As a result, while it initially discovered the existence of post-login URLs, it could not properly request them and it kept getting an *invalid session* message in all responses, leading to the clustering of different pages. This, however, was not the case for the other two scanners, which got proper, authenticated responses for these pages and did not cluster them incorrectly.

**Performance gain.** To measure performance gain we further analyze wapiti’s run on osCommerce as a representative case, since the application includes several similar pages that should be clustered and wapiti also takes the longest among all scanners to complete its operation. We then proceeded to re-run the scan *without* the URL clustering module and without a maximum scan time. The scan with URL clustering enabled took 62,690 seconds (17.4 hours) while the other scan took 418,622 seconds (116.3 hours), resulting in a ~6.7x speedup.

### 2.3.3 State-of-the-Art Comparison

Our system adopts a novel approach that allows it to leverage *any* underlying scanner. Nonetheless, we opt to compare it to Black Widow (BW) [141], a state-of-the-art scanner, that highlighted the need to combine features from multiple other systems and offers certain comparable features. This comparison highlights that even though BW was designed to incorporate multiple ideas from prior approaches, it was still built as a standalone tool, and thus is susceptible to the inherent limitations of a monolithic approach. In contrast, ReScan

Table 2.5: Detection and coverage comparison between the best run of ReScan and Black Widow for each app.

System App	Detection				Coverage	
	ReScan		Black Widow		ReScan	Black Widow
	R-XSS	S-XSS	R-XSS	S-XSS	LoC	
SCARF	-	<b>8</b>	-	4	<b>662</b>	593
WackoPicko	<b>3</b>	1	2	<b>2</b>	<b>1,009</b>	1,003
Wordpress	<b>1</b>	1	-	1	54,329	<b>62,281</b>
osCommerce	<b>3</b>	<b>16</b>	-	11	7,270	<b>12,193</b>
Vanilla	-	1	-	1	<b>12,951</b>	10,108
PhpBB	-	<b>4</b>	-	-	<b>10,487</b>	8,072
Prestashop	<b>1</b>	-	-	-	<b>103,955</b>	23,166
Joomla	-	-	-	-	<b>54,711</b>	50,240
Drupal	-	-	-	-	<b>80,620</b>	39,247
HotCRP	<b>1</b>	-	-	-	19,109	<b>23,241</b>

is designed to provide researchers with flexibility, allowing them to leverage the capabilities of any existing system of their choice, due to its middleware architecture.

**Setup.** To compare against BW, we downloaded and ran it on all applications. One minor change we made to its source code was to support user-defined credentials, as their system uses hardcoded values for *all* input fields, including usernames or emails and passwords. Moreover, we had to fix a few minor runtime exceptions that halted the tool’s operation, and write a custom parser that de-duplicates the scanner’s results. We stress that these changes were strictly limited to necessary modifications for the tool to execute properly and did not interfere with its overall approach or methodology. While their study ran each scanner for a maximum of eight hours in the evaluation, we opted to let BW run for up to one day.

**Qualitative differences.** As can be seen in Table 2.4, both systems use a fully-fledged browser, create a navigation model based on which they execute workflows and also uncover ISD links. While ISD detection is conceptually similar, ReScan does not have any control over when to crawl or fuzz each endpoint, highlighting the necessity of our BG worker approach. In more detail, BW always prioritizes form submissions over other edges and re-fetches *all* GET edges right before initiating the scanning phase; this results in it first fuzzing an ISD source and *then* visiting the corresponding sink, allowing for the timely detection of ISD links. In contrast, ReScan cannot make this assumption as it depends on the internals of each individual scanner; thus, we need to be more generalizable when handling ISD detection and need to account for arbitrary fuzzing and crawling orderings due to the underlying scanner design. Additionally, both systems can discover events and capture asynchronous requests and DOM changes. However, BW lacks the ability to set such requests’ payloads dynamically, while ReScan leverages an internal proxy to do so after the request has left the browser. Regarding authentication, while BW can submit a login form (with hardcoded credentials), it cannot infer whether the login was successful or not. Moreover, while it can re-login to the application if needed, it only does so when presented with

a login form. Applications' behavior varies and accessing an authenticated resource or exercising an authenticated functionality when logged out does not always result in a login page; thus their system will miss authenticated parts of target applications. On the other hand, ReScan automatically deduces a robust authentication oracle and consults it after the execution of *every* request and retries any operation that might have failed due to a broken session. Regarding similar pages, BW clusters pages and imposes a hard limit on how many similar pages they will test (if they share the same path but with possibly different URL query parameters) without considering the actual pages' content and functionality. This can incorrectly cluster pages that should be audited separately. Another main difference is that ReScan operates in a concurrent fashion, while BW is sequential, directly affecting its performance as seen in Figure 2.3. Finally, BW is a standalone tool that tests only for reflected and stored XSS, leaving a plethora of other flaws undetected. In contrast, our system operates as a generic enhancement middleware framework that can accurately replicate and potentially enhance virtually *any* test performed by scanners.

**Quantitative differences.** BW reported two *unique* reflected and 19 stored XSS among all applications, compared to ten and 34 detected by ReScan, as shown in Tables 2.2 and 2.5. Excluding Drupal and Joomla, for which no vulnerabilities were detected by any scanner, in all but two of the remaining cases there is at least one ReScan-enabled scan that outperforms BW. In WackoPicko, BW manages to detect a stored XSS through a comment which needs to be previewed first and *then* posted. However, the vulnerable field is not present in the final submission form, only in the intermediary preview form. Due to this irregular structure and since scanners attack each form separately, despite ReScan correctly modeling and executing the workflow for the final submission form, scanners cannot detect the flaw as they try to fuzz that form's fields only. Regarding coverage, ReScan outperforms BW in seven out of ten applications, and overall offers an average improvement of 46% in reached LoC. For the three remaining cases where BW achieved better coverage, it was either due to the max scan time being reached by all scanners, and since BW prioritizes form submissions over other edges, it likely managed to execute more functionalities, BW visited URLs that were excluded from the other scanners, as stated in § 2.3, or it visited unauthenticated parts of the application due to a broken session. Nonetheless, ReScan still managed to detect *more vulnerabilities* in these cases as well. Performance-wise, as can be seen in Figure 2.3, BW reached the max scan time of one day in eight of the applications, highlighting the shortcomings of their sequential execution.

## 2.4 Limitations and Future Work

**Categorization issues.** In certain cases scanners might report a stored XSS as reflected: When ReScan appends an ISD sink in the HTTP response, the scanner will detect the vulnerability as a reflected XSS, as both the injection and its reflection occurred in the same

request-response pair from the scanner's perspective. However, there is also a significant advantage to our technique, i.e., inspecting ISD sinks right after the injection in contrast to scanners' default behavior. Scanners will either visit discovered URLs at the end of the scan, looking for previous stored injections, or will try to re-inject their payloads and then inspect the URLs. These approaches, however, are not robust since a change in the application's state or content might render the detection impossible at this point. This could be due to a successful payload being overwritten by one that does not trigger the vulnerability, or the reflection page or injection point not being available anymore; directly checking ISD sinks solves this issue. Moreover, this miscategorization may also occur during scanners' regular operation, if a stored injection is reflected directly in the HTTP response. In this case, the scanner will initially classify the flaw as a reflected XSS, but later on, when checking for stored XSS it might be missed as one, due to the aforementioned reasons.

During our evaluation, we reported the discovered vulnerabilities based on their *actual* nature, despite being misclassified by the scanner. The rationale behind this decision is three-fold. First, this issue is inherent to scanners' behavior and further exacerbated by ReScan's techniques. More importantly, the vulnerable parameter has been detected, and the effort needed to patch it is the same regardless of the reported XSS type. Finally, ReScan's results provide all necessary information about each vulnerability, i.e., identifying the vulnerable parameter, the URL in which it is located, and where the injection is triggered.

**Session sharing.** As stated in § 2.2, workers share the authenticated session based solely on the website's cookies. While this is sufficient for our experimental setup and application set, in practice, applications might utilize other APIs and functionalities for their state and session management (e.g., local/session storage, service workers etc). We plan to augment our session sharing among workers to support such alternative approaches as part of our future work.

**False positives & negatives.** ReScan aims to eliminate FPs and FNs specifically for XSS flaws. Implementing this capability for other types of vulnerabilities requires inferring *what* vulnerability is being tested in each request and *how* successful exploitations would be verified. We consider the development of such modules for different classes of flaws as part of our future work.

**Overhead.** Leveraging a fully-fledged browser to appropriately execute every request, and employing our numerous enhancement techniques, imposes a considerable overhead in the overall scanning time. However, due to the significant improvement in code coverage and vulnerability detection, as well as due to ReScan outperforming the current state of the art in most cases [141], we believe this to be an acceptable trade-off which renders the deployment of our system feasible. Nonetheless, we consider the exploration of additional optimization techniques an interesting future direction. For instance, ReScan could identify requests that do not require our enhancement techniques (e.g., edges with no ISD effects or that do not require precise workflow execution) and directly proxy them to the



web application.

**Ethical considerations.** We note that all vulnerabilities detected during our evaluation have already been disclosed to the corresponding vendors by prior studies or researchers.



## Chapter 3

# Black-box Auditing for Web Authentication and Authorization Flaws

Web services have become treasure troves of sensitive data, rendering user accounts high-value targets for attackers. Recently, authentication flaws in popular web applications (or “apps”) exposed sensitive data and allowed access to critical functionality of millions of accounts [9, 10]. Reports have even implicated nation-state adversaries in attacks that ultimately aimed to steal user credentials [11, 12]. As such, authentication and authorization flaws in web apps are of great importance [272, 284] as they pose a significant threat. However, detecting such flaws is challenging.

As new technologies and features continue to emerge, web apps are becoming increasingly complicated. This complexity is exacerbated by their rapid evolution and the addition of new functionality and modules [127, 132]. This can result in the introduction of semantic bugs whose composite nature [250] renders detection a challenging task [132, 214]. Moreover, the massive codebase that comprises modern web apps is often developed by separate teams, which can have a negative impact [219] and result in fragmented auditing procedures that do not fully capture the side effects that arise from the interoperability of different components. Web apps can also include legacy code, which is often a significant source of new vulnerabilities [122], further complicating internal auditing procedures. To make matters worse, applicable security mechanisms are often deployed in an incomplete or incorrect manner [118, 153, 173, 237, 275]. As a result, external auditing initiatives from researchers can significantly contribute to the overall hygiene of the web ecosystem by discovering vulnerabilities. However, the sheer scale of this issue and the prevalence of obfuscation [230, 241] mandate an automated, black-box dynamic analysis.

In this work we adopt such an approach and focus on flaws that lead to the exposure of authentication cookies that allow adversaries to access sensitive data or account functionality. While recent studies have demonstrated that such flaws exist even in the most popular websites [115, 147, 238], these studies relied on significant manual effort and were, thus, inherently small-scale covering a very limited number of domains. With surveys reporting

that Internet users in the US now have ~150 password-protected accounts [6], and tens of thousands of websites streamlining account creation through Single Sign-On [147], it is apparent that manual efforts are not sufficient. To that end, we develop a completely automated black-box auditing framework that detects authentication and authorization flaws in web apps and identifies what sensitive/personal user information can be harvested by attackers. Our system is designed to handle *every step* of the process, including account creation and user-level interactions. Specifically, our framework analyzes the characteristics and infers the access privileges granted to cookies, while also evaluating the deployment of security mechanisms that can prevent cookie-hijacking attacks.

The main design goal of our framework is to automatically audit web apps in a black-box manner, without any prior knowledge of the underlying app’s structure or code. The framework is driven by XDriver, our custom browser-automation tool built on top of Selenium, designed for robustness and fault-tolerance during prolonged interactions with web apps. As XDriver is geared towards security-related tasks, we have implemented modules for evaluating security mechanisms that are pertinent to our study (e.g., HSTS). The black-box auditing process is handled by a series of components dedicated to specific phases of our workflow, including components that employ differential analysis and a series of oracles for inferring the account’s “state” reached by requests depending on the cookies submitted and the level of account access granted to those cookies. This requires identifying which cookies are used for authentication and exploring the conditions for different attack vectors under which they can be hijacked. Finally, our framework includes a novel module that analyzes web apps and detects personal user data (e.g., name, email, phone number) that is accessible using hijacked cookies. This is achieved through an in-depth investigation that analyzes the app’s client-side source, storage, and URL parameters to detect the exposure of sensitive data.

Using our framework we conduct the first *fully automated*, comprehensive, large-scale analysis of cookie hijacking in the wild. First, we crawl 1.5 million domains, and identify over 200 thousand domains that support account creation. Subsequently, our framework manages to fully audit almost *25 thousand* (~12%) of the domains, requiring 8.5 minutes per domain on average. Our experiments reveal that 50.3% of those domains expose their cookies under different scenarios and, thus, suffer from authentication or authorization flaws. To make matters worse, we find that security mechanisms that could prevent these attacks are not widely adopted (only 11.8% of vulnerable domains do so) or are often deployed in an erroneous manner. In more detail, we find that 10,921 domains expose authentication cookies over unencrypted connections, which can be hijacked by passive eavesdroppers and used to access users’ accounts. Moreover, 5,099 domains do not protect their authentication cookies from JavaScript-based access while simultaneously including embedded, non-isolated, third party scripts that run in the first party’s origin. With these scripts being fetched from 2,463 unique third party domains, users currently face a considerable risk of

malicious, compromised, or honest-but-curious third parties reading their authentication cookies.

Due to the severity of the flaws detected by our system, it is crucial that our findings are made available to developers so they can patch their systems. While we have notified several vulnerable domains, finding an appropriate contact point for such a vast number of domains is infeasible; thus, we have set up a notification service that allows developers to access the auditing results.

### 3.1 Background and Threat Model

Our framework focuses on detecting authentication and authorization flaws that stem from the incorrect handling or protection of cookies. While cookie hijacking is not a new attack vector, it can still affect even the most popular websites (e.g., Google, Facebook) and expose users to significant threats [238] including complete account takeover [147]. We consider the following types of attackers.

**Passive network attacker.** This attacker, referred to as an *eavesdropper*, has the ability to intercept and inspect unencrypted HTTP traffic (but does not attempt to modify it). We assume this attacker cannot intercept HTTPS traffic, and do not explore more elaborate, active attacks (e.g., SSL-stripping [192], cookie-overwriting [279]). This means that any cookies that are not protected with the *secure* flag can be intercepted by this attacker when appended to an HTTP request. This can, e.g., occur naturally while a user browses a website (since many websites serve certain resources over HTTP). An important detail that amplifies the practicality of this attack is that even when a domain supports HTTPS, browsers will by default attempt to access the domain over HTTP before being redirected by the web server to HTTPS [238]. While this can be prevented with mechanisms like HSTS, they are still not widely adopted and are often deployed incorrectly [173, 237].

**Web attacker.** This attacker can execute some JavaScript code within the origin of the web app, e.g., through a cross-site scripting (XSS) attack [149]. Another attack vector is introduced if the web app includes a script from a third party domain without “isolating” it in an iframe, effectively allowing it to execute in the first party’s origin [202]; malicious scripts (e.g., malvertising [185]) or compromised script providers can then read first party cookies [101]. We define as third-party any scripts that are loaded from a different domain [226, 253, 254], where the term *domain* will be used to refer to the eTLD+1 domain throughout the chapter. Consequently, cookies that are not protected with the *httpOnly* flag will be readable by client-side code and can be obtained by the attacker. We refer to these two attack vectors as *JS cookie stealing*.

It is important to stress that our framework does not search for XSS bugs or malicious third party scripts; our system focuses on automatically inferring the feasibility of stealing authentication cookies through JavaScript due to insufficient protection, and exploring the

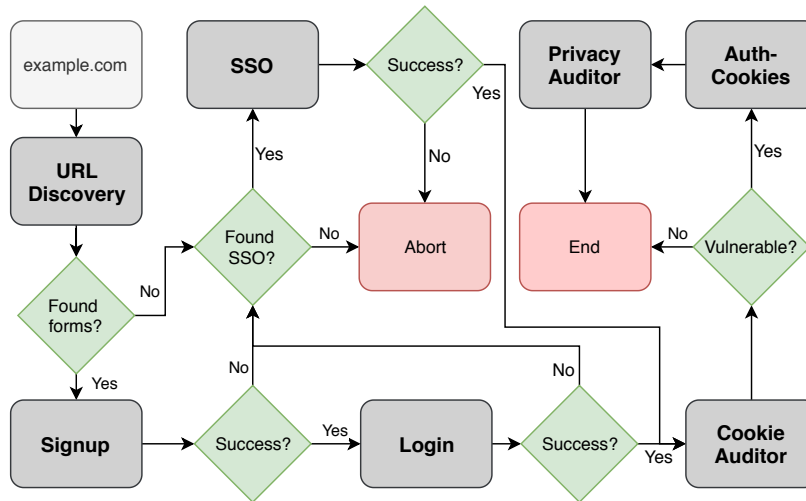


Figure 3.1: Major phases in our auditing workflow.

subsequent privacy implications for users. As such, the numbers reported on JavaScript-based cookie stealing are an upper bound that is contingent on the presence of XSS vulnerabilities or *malicious* third party scripts. Nonetheless, XSS vulnerabilities remain one of the most common attacks against web applications [5] and a plethora of detection systems have been proposed (e.g., [103, 253, 256]). Similarly, recent work has highlighted the prevalence of (suspicious) third party scripts [158, 176].

## 3.2 System Design and Implementation

Here we present our framework and the methodology of the core components of our black-box auditing process. Figure 3.1 depicts a high-level view of the workflow for clarity, and to facilitate presentation. In the following subsections we highlight each component in our pipeline and provide design and implementation details.

### 3.2.1 Automated Account Setup

The first phase in our workflow is to automatically create accounts.

**URL Discovery.** This module follows a straightforward process of crawling domains and terminating when *both* a login and a signup form have been located. As a first step it explores the URLs included in the public dataset by Ghasemisharif et al. [147]. If it does not locate both types of forms, next it will crawl the target web application. The crawl starts at the landing page and goes to a depth of 2 – we opt for a more shallow crawl to reduce the crawl’s duration and enable our large-scale study. Our framework collects all links included in each page that point to the same domain, and subsequently visits and inspects them. This step prioritizes links that contain an account-related keyword (e.g., signin, reg-

ister etc.) and follows a breadth-first search (BFS) approach. If both types of forms are yet to be found, the final step is to collect the first 30 links from the homepage and inspect them, excluding previously-visited URLs. This is based on the intuition that such pages are typically easily accessible to users and not hidden behind multiple menus, and are usually at the top of the page.

For each visited page, we extract any forms that resemble a login or signup process, and a series of heuristics are employed for detecting such forms within a page's code. Specifically, for each form we first count the number of text, email, password, checkbox and radio type input fields. We also check which of those are visible following the custom heuristics proposed by SSOScan [281]. If there are no password fields we skip the form since it probably is not a login or signup form (e.g., contact forms are common). If it contains more than one password field we label it as a signup form since such forms usually require the user to retype the password for verification. If there is a single password field and a single text field we label it as a login form, as this is the typical structure of such forms. If there are more than one text fields or checkbox/radio fields (accounting for the "*remember me*" option in login forms) the form is labeled as a signup form. If the form has a more irregular structure and has not been identified with these heuristics, our system resorts to using two sets of regular expressions (one for login and one for signup) for analyzing the HTML code and detecting elements that allow us to label the form accordingly.

**Automated signup.** Automating the account creation process in an application-agnostic way is a challenging task. This is due to the fact that websites have different requirements and constraints regarding the type and format of information for the fields needed for completing the registration. These vary and pertain to the number and type of fields (e.g., email, password, username etc.), as well as to the different restrictions in what is considered a valid input. For instance, a website might consider "+1 012 345 6789" a valid US number while another might require a different format.

The Signup module iterates over the discovered signup pages and attempts to fill each candidate form appropriately. We use a manually-curated set of regular expressions that try to detect what type of information each input element is expecting (e.g., email, postal address, date). We first carefully assign labels to each of the input elements by checking the *for* attribute of *label* elements, since we expect them to be the most descriptive. If there is no match, we move on to the element's HTML code (i.e., its attributes), which can reveal useful information about its type (e.g., an element of type `email` or with a descriptive `id` like `last_name`). If our module has yet to identify what type of information is expected, we consider the text content preceding the element. While this is the most common convention for labeling elements, developers are not constrained and can structure their forms differently. We, thus, follow a conservative strategy and consider these assigned labels as *possible* labels, since we cannot be certain of the form structure – in some cases the input element's accompanying text might be after the element. This is also why we prioritize any previously

identified labels, and consider the “possible” labels as a last resort.

If there is still no match, we use Google Translate to translate any labels assigned to the element in English and repeat the aforementioned process. This is needed since our analysis is not limited to English websites and foreign content is common. We refrain from using Google Translate initially, since the previous steps might reveal the type of field, allowing us to avoid the unnecessary API calls. Finally, we resort to either a random string for *text* inputs or a random selection for *select* and *radio* elements. To generate valid inputs after having detected the element’s type, we use Python’s Faker package. We also infer the input’s expected size by inspecting its `size` and `maxLength` attributes and adjust our value accordingly. After filling out the inputs we submit the form. At this point we need to infer whether the signup attempt was successful or not. We employ the following oracle that deems the signup process successful if any step yields a positive result:

- Visit the homepage and check if any of the submitted identifiers appear. The intuition is that if signup failed, websites would not store the provided information. We refrain from making the same check at the landing page after the form submission, since a website might display identifiers in an error message.
- Visit the form’s URL and check if it is still displayed. The intuition is that after a successful signup the website will not keep displaying the form. However, we have observed cases where the signup was successful, but the signup form was still displayed.
- Check if we received any emails from the domain. The intuition is that a failed signup attempt would not trigger an email delivery.
- Attempt to login to the website with our automated Login module (described further down). A successful login attempt indicates that the signup was successful.

If the signup is deemed successful we store the filled values and end the signup process. Otherwise, we try to identify any required fields in the form (i.e., by checking for the HTML *required* attribute or an asterisk or the *required* keyword in the element’s labels) and attempt to resubmit the form using only those, to reduce the probability of error. If that fails once again, we move on to the next form, until a successful registration is detected or all forms have been processed. After registration we also handle any emails sent by the domain, typically pertaining to account verification, to ensure that our newly created account is valid. As we cannot be certain of those emails’ structure or of any action that might be required, we extract and visit all URLs included in the email and try to detect commonly used keywords and phrases pertaining to successful verification. Through empirical analysis we observed that several websites might require the user to additionally click on a button in that page to finish the process. Therefore, if we do not detect any of the above keywords, we resort to clicking all displayed clickable elements in the page.

**Automated login.** For us to complete the login process, we visit the discovered login URLs (i.e., the ones that contain a login form) and submit each candidate form with our test account credentials. Concluding whether the login attempt has been successful is straight-



forward in most cases; the login oracle re-fetches the page with the login form and checks whether the submitted form remains in the page. If not, the login attempt is considered successful. During our empirical analysis we observed that several poorly designed websites kept displaying the form even after a successful login; to account for such cases, if the form persists, our login oracle additionally checks if any of our test account's identifiers (e.g., email, username etc.) are now present in the homepage's source code. Similarly, it uses a set of heuristics for detecting whether any logout buttons are displayed in the homepage. If either process yields a result the login is deemed successful.

**SSO Fallback.** If our system is not able to successfully complete the traditional account creation process, it alternatively identifies whether the app supports Single Sign-On with one of the most popular Identity Providers (*IdPs*) – we currently support Facebook and Google. If SSO elements are discovered it attempts to automatically complete the SSO process using test accounts that have been registered in the IdPs. First we need to identify if the site actually supports SSO; we have created a set of regular expressions that identify potential HTML elements in a page that can be used for performing SSO. The detection of such elements is performed during the execution of the URLDiscovery module. The module terminates if both login and signup forms have been located, regardless of the discovery of potential SSO elements. This is due to the fact that the available SSO options usually accompany the account related forms (if a traditional login scheme is supported). Thus, when locating a login or signup form we also detect if the site also supports SSO.

For each URL, we iterate over the candidate SSO elements and click them. We prioritize elements that are displayed, based on the intuition that sites are usually upfront about the available login options. For displayed elements we use Selenium's `click` method, effectively replicating a user's action. For hidden elements we refrain from trying to make those elements appear, which would involve clicking over other elements and potentially leading to unintended behavior and considerably increasing the process' duration. Instead, we try to trigger their `onClick` method via JavaScript. While this is generally effective, in some cases the candidate element is an outer wrapper element (e.g., a `<div>` element which contains an `<a>` element), and clicking it via JavaScript will not trigger SSO. Thus, for each non-displayed candidate element we also consider its children elements. While this leads to additional elements that need to be tested, we can quickly click on elements and decide if one is an actual SSO element; the overhead induced by this approach is negligible in practice.

The straightforward approach for inferring whether we clicked the correct element is to wait for the appearance of a predefined element, as a button that authorizes the app to access user data on the IdP should appear. However, this is inefficient and expensive as we would need to wait a sufficiently long time after clicking on every element to ensure that the necessary steps (and background server-communication) of the SSO protocol actually completed. We opt for a more elaborate approach that relies on the fact that an HTTP request is issued towards the IdP's SSO endpoint when the correct element is clicked. We

setup a modified proxy in passive mode which notifies our framework if such an outgoing request is observed. This allows us to quickly iterate over all candidate elements. The first time our system logs into a website we authorize the app in the IdP by following a few easily-automated steps.

It is worth noting that inferring whether the SSO process was successful is not necessarily equivalent to determining if our system is logged in the web app. For instance, a website might require a few extra steps to be taken (typically pertaining to account setup) after the user clicks on the SSO button and authorizes the app in the IdP; in this case our system will be in an intermediary state where the user is not yet fully logged in. We employ two separate oracles to decide if SSO completed and if we are logged in. The *SSO oracle* first checks if the SSO element we clicked on is still displayed. If not, the SSO was (most likely) successful. However, as some websites keep displaying the elements even after a successful SSO, the SSO oracle utilizes the *SSO login oracle* for further verifying the successful completion of the SSO process. This oracle searches for *displayed* account identifiers, logout buttons, and our IdP test account's profile photo which is often fetched from the IdP. If any of those checks is positive, the SSO login is deemed successful. This oracle focuses only on displayed elements, because we found cases where a website that was authorized in the IdP loaded identifiers provided by the IdP and displayed them in the page's source (e.g., in an inline JavaScript object) without having logged the user in.

Some websites require a few extra steps pertaining to account setup to be taken in order to complete the SSO. We detect and automate this process as well, using a modified Signup module that has a few minor changes in its workflow and oracle, which address SSO-specific variations in the process. Typically, websites display two options for completing the account setup after a successful SSO, the first being to link the new SSO identity with an existing account and the second about creating a new account. We detect any clickable elements that indicate the latter using regular expressions and iteratively click them. We then collect *all* forms displayed in the page, as we do not have any knowledge of their structure (i.e., it is common that such an account setup form might not even include a password field). Finally, we iterate over the discovered forms, fill and submit them, and consult our modified Signup oracle for each submission. As such, the oracle has been modified so the check for identifiers is done only on displayed elements, for the same reason with the SSO login oracle. In addition, if all other checks fail, we check if any password type fields were submitted in the signup form. If that is the case, we proceed by performing a *generic* login attempt using the discovered login forms.

**False Positive/Ambiguous Login Elimination.** After creating an account, we perform a final step to eliminate cases where our oracles yield a false positive (i.e., consider a login attempt to be successful despite not actually being logged in) or are not able to disambiguate between being logged in or not for a specific website. We send an HTTP request without appending *any* cookies and consult our login oracle once again; if it claims we are still logged

in we mark the website as a false positive and abort the process. This happens when a website does not follow any of the development “conventions” that our oracles anticipate, or other mechanisms interfere with the session’s state (e.g., a website displays an identifier that was stored in `localStorage` even when no cookies are submitted). It is worth noting that while it is straightforward to clear such storage mechanisms, we refrain from doing so since this can have unexpected effects on a website’s intended functionality and impact the operation of subsequent modules.

**Captchas.** Protecting account creation through captchas is common practice and, therefore, creating a captcha solver can considerably improve our system’s coverage. Initially, we implemented a solver based on recent attacks against Google’s audio reCaptcha [107, 247]. Unfortunately, reCaptcha’s advanced risk analysis system currently detects the use of Web-Driver, which results in Google not serving captchas to our framework. Since building a stealthier captcha solver is out of the scope of our work, and funding human captcha-solving services to create accounts presents an ethical dilemma, we opted to not handle such cases. However, due to the popularity of domains that employ captchas, in our evaluation we include a set of popular domains for which we completed the account creation process manually. We stress, however, that the ~25K domains that comprise the bulk of our evaluation *did not* require *any* manual intervention.

### 3.2.2 Cookie Auditor

To investigate whether users are exposed to session hijacking attacks due to flawed or vulnerable authentication practices, the next phase of our framework’s workflow relies on modules that analyze the cookies set by a specific web app and identify potential hijacking opportunities based on their attributes. As we require a method for deducing with minimal overhead which cookies provide some form of authentication, we design and implement a simple, yet effective, algorithm that we present in Algorithm 1. The core idea is to inspect whether the discovered cookies are protected with the appropriate security-related attributes and subsequently infer which of those cookies are used for authentication.

**Cookie attributes.** Our CookieAuditor algorithm begins by identifying which cookies set by the website are protected with the *secure* and *httpOnly* attributes and groups them accordingly (line 2). If a cookie has both attributes enabled, it will be included in both sets. It then iterates over these cookie sets (6) and infers whether the website is vulnerable to a specific attack from our threat model based on the corresponding attribute. Before actually evaluating a cookie set, it first checks if the set is empty. This indicates that the site is vulnerable to the attack, e.g., if none of the cookies has the *secure* flag set, an eavesdropper could successfully perform a cookie hijacking attack (7-8), as described in prior manual studies [238]. On the other hand, if the attribute is present in one or more cookies, the algorithm will either infer the result from the previously tested set or evaluate this cookie set.

**Algorithm 1** *CookieAuditor* algorithm

---

```

1: Function AUDIT
2:   critical_cookies ← {
      'secure' ← ['cookieA', 'cookieB', ...],
      'httpOnly' ← ['cookieD', 'cookieF', ...]
3:   }
4:   vulnerable ← { 'secure' ← NULL, 'httpOnly' ← NULL, }
5:   tested ← [ ]
6:   for attr, cookies in critical_cookies do
7:     if cookies.is_empty() then
8:       vulnerable[attr] ← True
9:     else
10:      for tested_attr in tested do
11:        tested_set ← critical_cookies[tested_attr]
12:        if cookies == tested_set then
13:          vulnerable[attr] ← vulnerable[tested_attr]
14:        else if vulnerable[tested_attr] AND cookies.is_subset(tested_set) then
15:          vulnerable[attr] ← True
16:        end if
17:      end for
18:      if vulnerable[attr] == NULL then
19:        vulnerable[attr] = EVAL(cookies)
20:      end if
21:    end if
22:    tested.append(attr)
23:  end for
24:  return vulnerable
25: Function EVAL (cookie_set)
26:   BROWSER.remove_cookies(cookie_set)
27:   BROWSER.refresh()
28:   return login_oracle()

```

---

Evaluating a set means that we exclude it from the browser's cookie jar (i.e., those cookies will not be sent in the subsequent request), issue a new HTTP request to the website, and consult the login oracle to determine if we are still logged in (26-28). As can be easily deduced, being logged in while excluding all cookies with a specific attribute means that the website is indeed vulnerable to the specific attack. However, if the exact same cookie set has been tested before we can directly conclude whether the website is vulnerable or not (12-13). Finally, in cases where the cookie set is a subset of a previously tested set where our test account remained logged in, we can again safely conclude that the website is vulnerable for this attack as well (14-15). For instance, if we excluded the set  $[A, B, C]$  and we were still logged in (i.e., vulnerable) then testing the set  $[A, C]$  would also result in a logged in state, since we would now send even more cookies than before. This is why we prioritize larger cookie sets (we omitted this part of our algorithm for brevity). Finally, after evaluating a cookie set, we send another request containing all the cookies, to make sure our session is still valid. (only if we were logged out after the test). If the session has been invalidated by the server, we login again and update our cookie values with those of the new session. This allows us to efficiently identify if a website is susceptible to cookie hijacking and, if so, via what means. In the worst case scenario, our approach would need 6 requests, i.e., 3 requests per security-related cookie attribute. It is important to note that this technique has

the drawback of not revealing which of the cookies are actually authentication cookies.

**Authentication Cookies.** To further analyze the root causes of authentication flaws, our framework needs to be able to identify the subset of authentication cookies among all the cookies that are set. Mundada et al. [200] proposed an algorithm, however, their approach overlooks certain cases and can lead to incorrect results. We build upon the core algorithm they proposed and modify it to correctly handle additional cases. Their proposed algorithm starts by considering only the cookies set at login time (*login cookies*) and generating a partially ordered set (*POSET*) of every possible combination. Since the search space is exponential, and in many cases infeasible to test all combinations, the algorithm establishes a series of *rules* based on the outcome of certain tests to reduce the testing time. The core algorithm works as follows:

- Alternate by testing one round from the bottom of the POSET (i.e., disabling cookies from a full cookie set) followed by a round from the top of the POSET (i.e., enabling cookies from an empty cookie set). According to their description, rounds are followed in an incremental manner and all cookie sets for a given round are tested consecutively (e.g., all cookie sets where only 1 cookie is disabled, then all cookie sets where 1 is enabled etc.). This is also the root cause that leads to incorrect results in certain cases, as we detail next.
- If a disabled cookie set causes the test to fail (i.e., the user is logged out), then all subsequent cookie sets that do not contain this set can be skipped.
- If an enabled cookie set is found to cause the test to succeed (i.e., the user remains logged in), then all subsequent cookie sets that contain this set can be skipped.
- If a cookie that was not set at login time is detected to be part of an authentication combination, a similar nested process is executed for the *non-login cookies* and the login cookie array is expanded to include these cookies.

While this approach is generally effective, we have identified scenarios where it yields incorrect results. To illustrate such a case, consider the following example: if a website has two authentication cookie combinations, e.g.,  $[A,B]$  and  $[C,D]$ , the algorithm will first set a rule when disabling two cookies. Specifically, when disabling  $[A,C]$  none of the authentication cookie combinations we are looking for will be complete, and the user will be logged out of the web app. This results in establishing the rule “*any cookie set that does not include  $[A,C]$  should be skipped*”. Later on, when disabling the set  $[B,D]$  (which satisfies the first rule), the user will again be logged out, leading to a similar rule for this set as well. At this point the ruleset dictates that *any set that does not include  $[A,C]$  or  $[B,D]$  will be skipped*. However, in the very next round (i.e., when enabling two cookies), when checking whether the actual authentication cookie combinations should be tested, the algorithm will skip them as they do not satisfy the above ruleset. As a result, the actual authentication cookie combinations will not be inferred.

Thus, we cannot blindly follow such rules when enabling cookie sets. This, however, introduces the risk of a major performance penalty. Consider a second example of a website

that has two authentication combinations, e.g.,  $[A]$  and  $[B]$ . The first rule the algorithm will set will be when enabling a single cookie. Specifically, when only enabling  $[A]$  the user will be logged in and a rule will be set, dictating that “*any cookie set that includes  $[A]$  should be skipped*”. Likewise, when enabling  $[B]$  a similar rule will be set. In the next round (i.e., when disabling two cookies) the only set that will be tested will be the one not containing  $[A]$  and  $[B]$ , as it is the only one that respects the current ruleset, and the user will be logged out. This results in the rule “*any cookie set that does not include  $[A]$  or  $[B]$  should be skipped*” being set. Next, when enabling two cookies, and having established that we cannot follow the last rule when enabling cookies, the algorithm will then test *all* sets of length two that do not contain any of the two authentication cookies. The following rounds of the algorithm behave similarly (i.e., disabling/enabling three cookies and so on). However, we can tell that the algorithm has already detected the authentication cookie combinations and should not try any more tests.

To avoid this performance issue, we modify the algorithm to respect such rules when enabling cookies, but in a slightly different manner: cookie sets that result in the user being logged out when disabled are flattened into a vector (e.g., the ruleset  $[[A,C], [B,D]]$  from the first example becomes  $[A, B, C, D]$ ) and we safely skip the cookie sets that do not include **any** of these cookies. In our first example this results in the authentication cookie combinations being detected. In the second example it results in not testing any sets that are redundant after detecting the correct combinations.

We also note that while we label them as *authentication* cookies, since they lead to the exposure of user identifiers, this might be the result of flaws in the web app’s authorization policies, and not due to them actually being designed as (or intended for) authentication. Nonetheless, our goal is not to infer the developers’ intention but to identify which cookies lead to (full or partial) authentication.

### 3.2.3 Privacy Leakage Auditor

Apart from automatically detecting flaws that expose authentication cookies, our goal is to also identify what personal or sensitive user data attackers can obtain. We develop PrivacyAuditor for locating leaked user information following a differential analysis methodology. Our framework first effectively replicates a session hijacking attack; it creates a fresh browser instance and includes all *stolen* cookies, i.e., the ones that are not protected with the corresponding cookie attributes. If our system has labelled a specific web app as susceptible to both eavesdropping and JS cookie stealing attacks we only simulate the eavesdropping attack to demonstrate the privacy threat posed by attackers that are *less sophisticated*. Our system also deploys a logged-out browser alongside the authenticated browser and then proceeds with collecting links of interest. The module focuses on URLs that match account related keywords (e.g. *profile, settings*) and also collects the top 30 links that appear

in the main browser but not in the logged-out one (or less if not that many exist). Typically, we expect those links to point to restricted areas of the website where user information, possibly sensitive, will be stored.

We check each page for user information that was supplied during the signup process. If SSO was used, our system also checks for information that the web app might have pulled from the IdP (we have populated our Facebook and Google profiles with additional information). We inspect the rendered page source once JavaScript-generated content has finished loading. Since user data can be leaked in ways that are not directly visible to the attacker, our system also inspects other potential leakage points, including cookies, local and session storage, and the page's URL (we do not look at outgoing connections since we are not interested in what information is shared with third parties, and leaked identifiers will already be present in one of the locations we search). To account for cases where user information may be "obfuscated", we also check for encoded values of all the identifiers using common encoding (base64, base32, hex, URL encodings) and hashing techniques (MD5, SHA1, SHA256, SHA512). While we are able to capture obfuscated values of all user-specific information, in our experimental evaluation we only discuss obfuscated passwords and emails; this is due to their sensitive nature and because hashed emails can constitute PII and in certain cases are easily reversible [7, 130, 193].

### 3.2.4 Browser Automation

At the heart of any web app auditing framework lies the browser and, thus, it is imperative that our framework is orchestrated by a robust browser automation component. In practice, while Selenium is a powerful tool, it is better suited for testing scenarios when the web app's *structure* and *behavior* are known in advance. However, when conducting a complex, large-scale analysis there is no *a priori* knowledge of either. There are also numerous scenarios where unexpected behavior, structure changes, or software crashes impact browser automation functionality. For instance, at any moment during the execution of a module there might be an unexpected popup (e.g., an alert). This can block all other functionality, such as fetching and interacting with elements in the page. Moreover, current error raising and handling support can lead to ambiguous states; e.g., when Selenium's Chromedriver crashes (which is a common issue) a `TimeoutException` might be raised, which is also what happens when a website actually times out. Thus, we need a way to handle such obstacles efficiently whenever they occur without aborting and restarting the whole process. Finally, while other well-designed options exist, e.g., Selenium-based OpenWPM [139], we find that they focus on the browser setup, management and synchronization parts of automation, with little focus on dynamic interaction (e.g., element clicking, form submission) which is a critical aspect of our study. In addition, while Puppeteer [22] does offer interaction functionality, it suffers from the same robustness issues as Selenium, which our system tackles (e.g.,

element staleness, crash recovery, robust error handling). Moreover, Puppeteer is specifically designed for Chrome/Chromium, while we aim to make our automation component compatible with different browsers.

To address these limitations we develop XDriver, a custom browser automation tool designed for security-oriented tasks that offers improved fault-tolerance during prolonged black-box interactions with web apps. XDriver is built on top of Selenium and the official Chrome and Firefox WebDrivers [18, 20], and has been made open source [31]. We extend Selenium’s high level WebDriver class to enhance our system’s robustness by addressing the aforementioned challenges in a way that is *transparent* to the caller scripts. In the following paragraphs, we present the most prominent exceptions and how our system handles them, as well as a number of useful auxiliary mechanisms we implement. Our extensions amount to approximately 1,500 lines of code.

**Invocation.** XDriver extends Selenium’s WebDriver class and declares a custom `invoke` method which accepts a parent class method as an argument (e.g., `WebDriver.findElement`) and an arbitrary number of named and unnamed arguments. `Invoke` then calls the passed method in a `try-except` block, catches any raised exception and either calls the appropriate exception handler or returns a default value. XDriver then overrides all of WebDriver’s methods to call their parent class counterparts via `invoke`.

**Element staleness.** As our auditing requires prolonged, multi-phase interaction with web apps, page elements frequently become “stale”, which creates complications and can lead to crashes. XDriver is designed to handle such cases transparently and robustly. All interactions start by fetching a page element, e.g., based on the `id` attribute, and proceed with processing that element. If in the meantime this element is deleted or, more commonly, an asynchronous page load or redirection occurs, a `StaleElementReferenceException` is raised when interacting with the element, indicating that it is no longer attached to the DOM. However, while from a user’s perspective the element might still be present in the page, from Selenium’s point of view it is a new element under a new object reference, with no relation to the previously returned element. To handle this, when a `find_element_by` method is invoked, the returned element’s object reference is stored as the key in a hash table, with a tuple containing the invoked method and its arguments as the value. Then, whenever such an exception occurs, the given element’s reference is retrieved from that hash table and XDriver attempts to re-fetch it by invoking the stored method. If the element is found, the **old** element’s object is updated transparently with the newly returned element, and the initial requested operation that raised the exception is retried. Otherwise, the exception is raised since the element truly does not appear in the page.

**Handling crashes and timeouts.** When Chromedriver or some other component (e.g., intermediate proxy) crashes and a `TimeoutException` is raised, our XDriver module detects the crash, transparently restores the browser instance and state and eventually fulfills any module’s request that was interrupted by the crash. Specifically, it launches a new browser



instance, reloads the current browser profile to maintain state and updates its own object reference with that of the new one, so as to transparently update all references of the driver held by the framework modules. It also obtains the last known URL and retries the interrupted operation. The `StaleElementReferenceException` handler is extremely useful in this case, since all retrieved web element objects will have become stale due to the browser reboot.

**Unexpected Alerts.** If an alert popup appears and an `UnexpectedAlertPresentException` is raised during the invoked method, the execution context is switched temporarily to the alert box, which is then dismissed, and the method is retried. To prevent other alerts from appearing in the current page's context, the `window.alert` method is overridden.

**Retry mode.** We have developed a *retry mode*, which is used by `XDriver` whenever it needs to perform an action it can retry in case of failure; this is done without having to return control back to the caller, e.g., when a page's links or login forms are requested. Specifically, if an exception is raised while performing the operation, `XDriver` will retry the operation for a certain amount of times before raising the exception or returning a default value.

**Built-in crawler.** Our custom browser automation tool includes a built-in crawler for streamlining crawl-based tasks, a functionality that is especially vital in security-related studies. In our framework's context it is useful for our `URLDiscovery` and `PrivacyAuditor` modules for crawling and processing websites. Modules that want to initiate a crawl only need to call the `crawl_init` method with the desired configuration options and then iteratively call the `crawl_next` method, where all logic of the crawl is transparently implemented. The following configuration options are currently supported by our system: (i) Crawl depth, (ii) DFS or BFS mode, (iii) optional support for a set of regular expressions that dictate which URLs and even subdomains to follow or not follow (e.g., focus only on login related URLs or crawl a specific subdomain), and (iv) an optional break function that is applied after every fetched URL to determine whether the crawl should stop (e.g., if a specific type of form is found).

**Return values.** Additionally, to simplify the checks that the caller modules have to make for determining whether a requested operation was successful, we refrain from raising Selenium-level exceptions and, instead, return default boolean values. Only in cases where our handling mechanisms cannot resolve an issue we consider the exception to be fatal and raise it. For instance, when a module attempts to interact with an element that is not currently interactable (e.g., clicking an invisible element) a `False` value is returned instead of raising the default `ElementNotVisibleException`.

Overall, all of the above enhancements allow for more fault-tolerant interaction with web apps, reduce code complexity, and allow our main framework modules to focus on their specific tasks.

**Security mechanisms.** Another important feature is the detection and evaluation of security mechanisms pertinent to our study. HTTP Strict Transport Security (HSTS) instructs

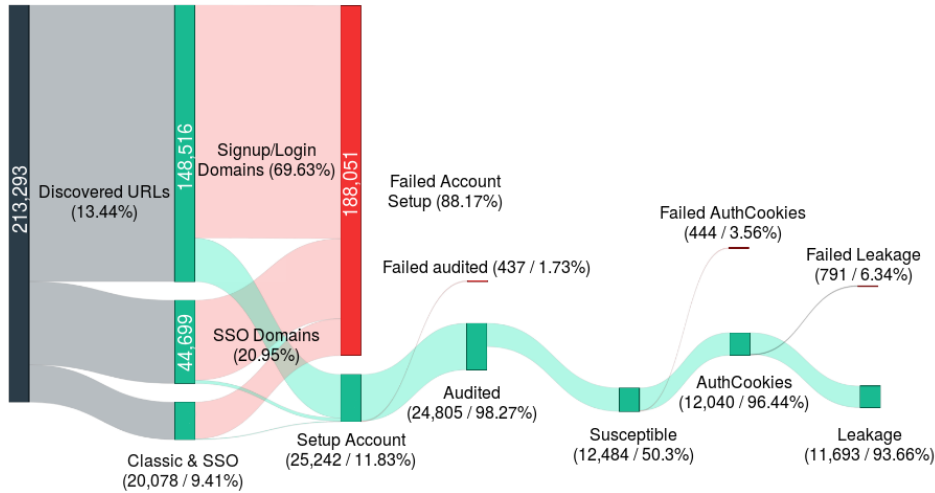


Figure 3.2: Success rate for different workflow phases.

a user’s browser to connect to the HSTS-enabled domain only over HTTPS for a specified amount of time, even if an explicit HTTP URL is followed or typed in the address bar by the user. While this seems fairly straightforward to deploy, domains often do so incorrectly or partially [173, 237, 238]. To evaluate deployment and detect misconfigurations, our module first checks whether the domain is in the Chromium preload list [23] and, if not, uses a passive proxy to capture the target website’s redirection flow from its HTTP endpoint to HTTPS. For each redirection, it stores the HSTS policy (if one is sent) and assesses whether the (sub)domain is indeed protected. Our module detects all the misconfigurations and errors presented in [173]. We note that while we implement mechanisms that are relevant to this work, XDriver’s modular design streamlines the addition of other security mechanisms.

### 3.3 Experimental Evaluation

We experimentally evaluate our black-box auditing framework and present our findings from the largest study on cookie-based authentication and authorization flaws in the wild.

**Datasets.** We use two different versions of the Alexa Top 1 million list. The first dataset was fetched on 09/14/2017; this dataset was useful for guiding the design and implementation of our framework. However, since recent work has revealed that domain ranking lists exhibit significant fluctuation even within short periods of time [231], we also obtained a second up-to-date version on 05/07/2019, when it was time to conduct the final evaluation. All the experiments presented here were conducted between May-October 2019 on a combined dataset that included a total of 1,585,964 unique domains.

**Workflow statistics.** One of our main goals is the ability to conduct automated black-box auditing of modern web apps without knowledge of their structure, access to the source code, or input from developers. The complexity and often ad-hoc nature of web devel-

opment render this a challenging task, and various obstacles can prevent the successful completion of a given module. Figure 3.2 provides statistics on the number of domains for which each phase of our workflow was successful. In general, our auditing modules are highly effective, successfully completing their analysis for 93-98% of the domains they handle. The failures in these modules are attributed to websites timing out (or being generally unresponsive) after several auditing tests and network failures. Also, when re-evaluating these domains other factors can affect the execution of our modules, such as our test account being deactivated, expired domains etc. As expected, automated account creation presents the most considerable obstacle; namely, out of the 168,594 domains for which we identified a signup option, we successfully registered and logged into 13.7% of them, while in 2,066 cases our system managed to login via SSO, out of which 346 were a fallback after a failed signup attempt. It is worth noting that for domains where we detected a signup option but were not able to create an account, 19,491 (~13.8%) embedded Google's reCaptcha. Yet our framework is still able to create accounts on 25,242 domains, accounting for almost 12% of the domains for which we have identified a signup option – for comparison, prior related studies analyzed 25 [238] and 149 [200] domains. In studies with a different focus, Zhou and Evans used SSO to audit 1,621 domains for SSO implementation flaws, while DeBlasio et al. [129] explored the risk of password reuse by creating accounts in over 2,300 domains. In other words, our study is several orders of magnitude larger than prior studies with a similar focus, and at least one order of magnitude larger than studies that employed some form of automated account creation. It is worth noting that the automated account creation process is the biggest challenge for our framework due to two reasons. First, the registration process may include predicates that significantly complicate the automated input generation due to input format constraints. For instance, the registration may include a mandatory field (e.g., postal address) that requires a valid value for a specific location/country. Iteratively testing different input formats can prohibitively increase the duration of the auditing process at the scale of our analysis. Second, registration might require access to a specific resource (e.g., phone number or credit card) that is not feasible to obtain for a study of our scale.

*False negatives.* To obtain more insights about our framework's effectiveness we perform an indicative experiment where we investigate the false negative rates (FN) of the different modules in our system. Specifically, we randomly sample 20 websites per module, where the module's execution did not complete successfully, and manually inspect whether these failures were actual true negatives or not. For our URL discovery module, we identified only four FNs, i.e. in four cases there was a login option that our system failed to detect. Our generic account setup component yielded 3 FNs, i.e. we successfully signed up and/or logged in the website, but were not able to infer the state. Similarly, the SSO module had 5 FNs. The Cookie Auditor yielded zero FN, meaning that there was not a single case where our system identified a website as secure against an attack, while it really was vulnerable.

Table 3.1: Number of unique domains that do not adequately protect their cookies from specific attacks.

Attack	# of Domains (%)
<b>Eavesdropping</b>	12,014 (48.43%)
No HSTS	10,495 (87.36%)
HSTS Preloaded	64 (0.53%)
Full HSTS	188 (1.56%)
Faulty HSTS	
- Protected	736 (6.13%)
- Vulnerable	426 (3.55%)
Final Vulnerable	10,921 (90.9%)
<b>JS cookie stealing</b>	5,680 (22.9%)
<b>Total</b>	<b>12,484 (50.33%)</b>

Finally, the Privacy Auditor had 4 FNs, i.e. there was account information that we provided during the signup process that was not detected as being leaked. We did not measure the Authentication Cookies FN rates, as manually identifying *all* authentication cookies and combinations is prohibitively time consuming or even infeasible in many cases.

*URL discovery effectiveness.* As mentioned, our URL discovery module initially explores the URLs provided by [147] before falling back to our own crawling approach. As such, it is of interest to quantify how useful this dataset was and, more importantly, how effective our system was in cases where it had to employ our own approach. For all the websites where we identified a signup option, 23.1% were fully discovered using the dataset from [147], while for the remaining 76.9% we had to fall back to crawling the websites (43.1% were included in both datasets, while 33.8% were not included in [147]).

*Failed registrations.* In an attempt to better understand the reasons behind failed registrations, we manually inspected 50 randomly selected websites. In 22 cases, there was some form of an anti-bot challenge that our system was not able to solve and, thus, could not proceed with registration. In 23 websites one of the fields was rejected due to inappropriate formatting, e.g. mobile phones, addresses, passwords etc. Finally, the remaining 5 websites failed due to unexpected or complex form behavior, e.g. after filling in a specific field, a custom drop down list appeared that also needed to be detected and filled out.

**Cookies.** Audited domains set an average of 14.02 cookies, while susceptible domains set 1.21 authentication cookies and have 1.1 authentication combinations on average. In Table 3.1, we show the number of domains that expose their authentication cookies, i.e., do not protect them with the corresponding cookie attributes.

**Eavesdropping.** We find that 12,014 unique domains do not protect their authentication cookies with the *secure* flag, even though 1,815 of those set the flag for at least one of their cookies. However, web apps might make use of HTTP-Strict-Transport-Security (HSTS), which can prevent the leakage of those, otherwise exposed cookies. Merely check-

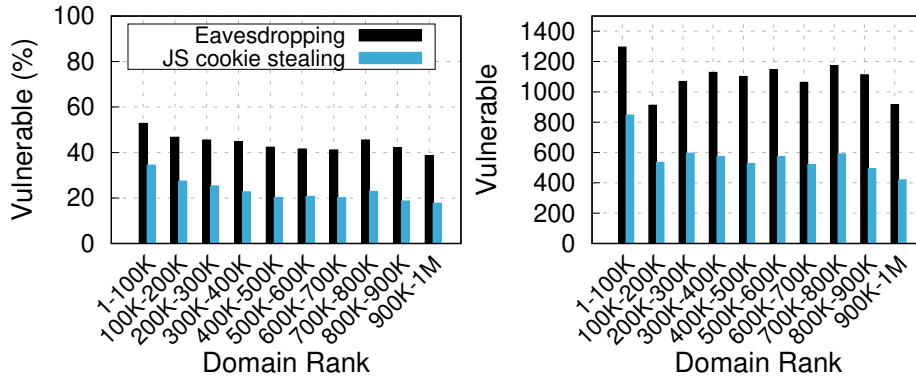
ing for the presence of HSTS headers in the web app’s responses is not sufficient, since prior studies have found that developers often deploy HSTS incorrectly [173, 237] or do not adequately protect their entire domain [238]. As such, our framework includes a module for evaluating the correctness and coverage of HSTS deployment for domains that are vulnerable to eavesdropping (the other attacks are not affected by HSTS).

We find that the situation has not improved much compared to prior studies, as the vast majority of domains do not deploy HSTS. While flawed HSTS deployment remains common, we find that 63.3% of the domains that have a faulty deployment do manage to prevent our cookie hijacking attacks. This is because the set of (sub)domains the auth cookies are sent to are protected by HSTS. For instance, if *example.com* deploys HSTS properly on the *www* subdomain, but leaves the base domain unprotected, and at least one auth cookie has its *domain* attribute set to *www.example.com*, then there is no way for an eavesdropper to retrieve this cookie. The most common misconfiguration is not enabling HSTS on the base domain (696 domains), out of which 143 attempted to set HSTS over HTTP. The remaining domains, while properly setting HSTS on their main domain, did not use the *includeSubdomains* directive, thus potentially leaving certain subdomains exposed. We also find that out of the remaining domains only 99 employ CSP’s *upgrade-insecure-requests* directive. While this reduces the attack surface, these domains remain vulnerable since this mechanism does not upgrade top-level navigational requests from third-party sites or the initial request (e.g., when a user opens a new tab and visits a site). Overall, 10,921 domains are vulnerable and expose cookies to eavesdroppers even when accounting for the presence of relevant security mechanisms. We further correlate these domains with the Single Sign On data released by [147] and found that four of these domains are also SSO identity providers (Amazon, Bitly, DeviantArt, GoodReads) and have at least 1,346 unique relying parties, out of which 138 have been audited by our system; 87 were found secure and 51 vulnerable to at least one of our attacks.

**JS cookie stealing.** We find that users face a considerable threat due to their authentication cookies being accessible via (malicious) JavaScript, as a total of 5,680 domains do not protect them with the *httpOnly* flag. Our framework’s analysis of those domains reveals that 5,099 include at least one embedded 3rd party script (i.e., not isolated in an *iframe*) that runs in the 1st party’s origin and has “permission” to read the user’s 1st party cookies. These are fetched from 2,463 unique 3rd party domains. To make matters worse, only 239 of those use the Subresource Integrity (SRI) feature [93] to prevent the manipulation of fetched scripts, and only one domain protects all loaded scripts. Similarly to [117], we find that all SRI-protected scripts are libraries (e.g., *jquery*). It is important to emphasize that this attack explores the potential threat from compromised or rogue 3rd parties, and that our numbers do not reflect active attacks currently underway in the wild. While our study’s focus is not on detecting malicious scripts actually stealing users’ cookies, we consider this an interesting future direction.

Table 3.2: Number of domains for different values of authentication cookies and combinations of authentication cookies.

	1	2	3	4	5	6	7
<b>Auth combos</b>	10,878	1,110	39	10	3	-	-
<b>Auth cookies</b>	9,912	1,700	364	54	7	2	1

Figure 3.3: Percentage (**left**) and absolute number (**right**) of vulnerable domains per ranking bin.

We emphasize that the 5,680 domains are not necessarily vulnerable to session hijacking through XSS, since other prevention mechanisms might be in place. For instance, Web Application Firewalls (WAFs) [131, 175] or Content Security Policies (CSP) [275] could be deployed to mitigate XSS attacks which could also prevent cookie stealing. Nonetheless, recent work has shown that even such defense mechanisms can be bypassed [182]. As such, our findings constitute an upper bound for web apps that are vulnerable to cookie-stealing via XSS. Nonetheless, while adoption of *httpOnly* is not as limited as in the past [280], it remains an important issue.

**Auth combos.** Table 3.2 breaks down the AuthCookies results and reports the number of domains with the corresponding number of authentication cookies and combinations. An interesting observation is that 435 of the domains that have *more than one* combination contain at least one secure combination among them, yet remain susceptible to attacks due to other combination(s) being exposed. This highlights how the ever-increasing complexity in web apps leads to authorization flaws. We also find that 76 domains contain cookie combinations that are correctly detected by our approach for which the algorithm from [200] returns incorrect results.

**Popularity.** We break down the vulnerable domains based on their Alexa rank in Fig-

Table 3.3: Most common categories of susceptible domains.

Category	#domains	Category	#domains
Online Shopping	3,725	Soft/Hardware	252
Business	1,117	Sports	234
Marketing/Merch.	1,100	Job Search	229
Internet Services	642	Pornography	194
Entertainment	586	News	187
Education/Reference	558	Real Estate	178
Blogs/Wiki	393	Public Info	153
Fashion/Beauty	322	Health	148

ure 3.3. In general, our framework detects more vulnerable domains in the highest ranking bin. This can be partially attributed to popular websites being more likely to support account creation (we find twice as many such domains in the most popular bin compared to the least popular one), while the process succeeds for roughly 11 – 13% of domains across all bins.

**Domain categorization.** Table 3.3 reports the top domain categories (classified using McAfee’s URL Ticketing System [88]) that are vulnerable to at least one attack. We find that online shopping is the most prevalent category of susceptible domains, highlighting the privacy threat of cookie hijacking. These services include a plethora of personal data (e.g., address), while recommendations and prior purchases can reveal sensitive user traits (e.g., sexual orientation, religion). We also find 148 and 194 domains that provide health-related functionality and adult content respectively, which potentially enable access to extremely sensitive user data.

**Privacy leakage.** In Table 3.4, we break down the personal or sensitive information that an attacker can acquire upon successfully hijacking a user’s cookies, as detected by our PrivacyAuditor module. We also report the total number of domains leaking such information, grouped per sensitive field (e.g., email) and also based on the source of leakage (e.g., page source). While a domain might appear in different columns of the same sensitive field, or different rows of the same source of leakage, it is only counted once in the corresponding totals. In general, we find that the page’s source is the most common avenue of exposure, but passwords are typically exposed through cookies. Furthermore, 59 out of the 68 hashed passwords detected by our system are MD5 hashes, which do not offer much protection against offline brute-forcing attacks. In practice, the attacker could potentially recover the password and obtain full control over the victim’s account in those services; password reuse [15, 212] can result in attackers accessing accounts in other services as well. Apart from common identifiers like emails and usernames, many domains expose highly sensitive data like home addresses and phone numbers. Overall, an abundance of data is exposed that can be used for doxxing [245], and a plethora of scams including targeted phishing [157] and identity theft [106].

Table 3.4: Personal user data that can be obtained by attackers.

Data	Source	Cookies	Storage	URL	Total (%)
Email	6,894	776	174	51	7,130 (61)
Email hash	885	68	10	0	930 (7.98)
Fullname	4,287	198	170	44	4,330 (37)
Firstname	648	58	8	10	686 (5.9)
Lastname	618	86	19	13	665 (5.7)
Username	1,856	339	48	175	1,956 (16.7)
Password	2	20	0	0	22 (0.19)
Pswd hash	12	57	0	0	68 (0.6)
Phone	1,594	8	7	2	1,598 (13.7)
Address	656	0	0	1	656 (5.6)
VAT	17	0	0	0	17 (0.15)
Workplace	540	3	3	1	543 (4.6)
<b>Total (%)</b>	<b>9,122 (78)</b>	<b>1,236 (10.6)</b>	<b>314 (2.7)</b>	<b>290 (2.5)</b>	

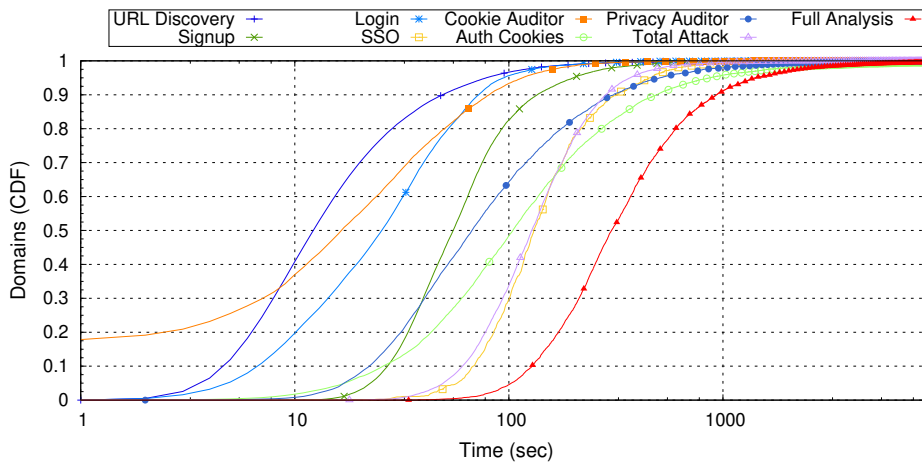


Figure 3.4: Time required by each module of our system.

**System performance.** In Figure 3.4 we show the total time in seconds required by each module in our framework. Since some modules might fail for certain domains, the different CDFs have been calculated using their corresponding totals. The total time required for auditing websites for attacks (i.e., all modules up to CookieAuditor) is denoted as *Total Attack*. The total time required for the analysis including the execution of AuthCookies and PrivacyAuditor is denoted as *Full Analysis*. We find that our framework’s performance is suitable for large-scale studies as half of the domains can be completely audited within 5 minutes and 90% in less than 17 minutes. While certain domains in the long tail of the distribution require considerably more time, this is typically due to latency issues with their specific servers. While Webdriver crashes can affect performance, our XDriver optimizations mini-



mize their impact by transparently recovering the browser’s state.

**Popular domains.** While our main goal is to automatically explore the feasibility of cookie hijacking at scale, popular domains are of particular interest because they are used by hundreds of millions of users and, thus, can have a greater impact if vulnerable. Considering that our framework’s entire workflow is fully automated and that app-agnostic account setup is extremely challenging, we opt to manually assist with the account setup for a subset of the most popular domains. Specifically, we consider the top 1K domains, where we identified 698 account-based websites. Out of those, 95 were already fully handled by our framework. For the rest, we manage to manually create accounts in 206 domains, which we provided to our framework to complete the automated auditing process. The remaining domains either protected their login forms with reCAPTCHAs, detected the presence of our webdriver, or requested information during signup that we were unable to provide (e.g., phone numbers for SMS verification, valid SSN etc.). Moreover, for 45 websites our Login Oracle could not disambiguate between being logged in and logged out; when sending a HTTP request without any cookies our account would still appear to be logged in. In total, we audited 301 popular websites (the additional 206 domains were not included in our previously reported numbers, thus, pushing our total analysis to over 25K domains).

We find that 149 are vulnerable to eavesdropping, 46 of which were fully handled by our framework. Only 10 domains deploy HSTS effectively, while another 30 (20.13%) use HSTS but remain susceptible due to faulty deployment. For JS cookie stealing, 115 domains were found susceptible and 104 include at least one embedded 3rd party script (from 266 domains) – only five make use of SRI. Overall, 57.81% of the domains do *not* provide adequate defenses, which is alarming considering their massive user base.

**Hijacking validation.** To manually validate our results and ensure that an attacker can actually access victims’ accounts, we conduct an exploratory experiment on domains that were *fully handled* by our framework. We randomly select ten and hand-pick another ten domains out of Alexa’s Top-1K, and randomly select another ten from the remaining domains, and simulate cookie hijacking attacks. We setup a browser instance where we log in the website and capture all cookies that are exposed depending on the threat model. Next, we launch a new browser with different characteristics (user agent etc.) on a different machine, in a different network subnet, where we include the stolen cookies and visit the website. We manually interact with the website to detect the extent of access the attacker obtains. We do not set a time limit; instead we opt for an exhaustive approach where we try to identify all user-specific functionality that should be tested. We detail our findings in Table 3.5. For the Top-1K random subset, we get full account access for seven domains (i.e., all tested operations succeeded), and partial access for three domains. For the other random subset we get full access in nine out of ten domains. Indicatively we can view and modify account settings, preferences, shopping lists, orders and subscriptions and post comments. In five of all the domains we could also change the user’s password *without* knowledge of

Table 3.5: Manually validated domains and hijacking capabilities.

Domain	Read	Write	Settings	Exposed information & functionality
<b>Top-1K (hand-picked)</b>				
amazon.com	◐	◐	x	View/edit cart, ad preferences, vouchers/coupons, shopping list, email subscriptions, deals & notifications, browsing history and recommendations
aliexpress.com	●	●	x	View/edit favorite stores, wish list, cart, profile photo, full name, follow sellers. View messages, order history, coupons
ebay.com	●	◐	x	View/edit cart, watchlist, saved searches/sellers, messages, address, profile photo. View recently viewed items, active bids/offers, purchase history, own items for sale
alibaba.com	●	●	x	View/edit cart, full name, phone number, gender, address, job information, favorites, profile photo. View messages, orders, transactions, contacts, recommendations
reddit.com	●	●	x	View/edit posts, comments, saved, display name, about section, profile photo, inbox, email notifications, block users
bing.com	◐	◐	x	View/edit search history, interests. View first name, profile photo
bestbuy.com	◐	◐	x	View/edit cart, saved items. View shopping history, orders
banggood.com	●	●	x	View/edit cart, wishlist, address, full name, gender, phone number, messages, reviews, comments, download full activity record. View orders, coupons, gifcards, search history
wish.com	●	●	x	View/edit cart, wishlist, full name, birthdate, email, notification settings. View orders, recently viewed items
cloudflare.com	○	○	x	None. The attack only succeeds when performed from the same PC
<b>Top-1K (randomly selected)</b>				
indeed.com	◐	◐	x	View/edit saved job offers, job applications, scheduled interviews, visited jobs
hotels.com	◐	◐	x	View/edit favorites, searches
vidio.com	●	●	✓	View/edit phone number, comments, followed channels, password. View transaction history, watch history
nature.com	●	◐	x	View/edit full name, professional information, subscriptions. View email
sciencedirect.com	●	●	x	View/edit full name, email, job information, phone number, address. View recommendations, history
1fichier.com	●	●	N/A	View/edit files, folders, full name, address, phone number.
bitly.com	●	●	x	View/edit bitlinks, link statistics, email address, delete account. View API key, session history (and disconnect all sessions)
cdiscount.com	●	●	x	View/edit subscriptions, wish/favorites list, address, phone number. View email, birth-date, orders, messages, vouchers, credit card info
elsevier.com	●	●	x	View/edit cart, full name, email, address, phone number, partial payment information, add new credit card
espnricinfo.com	●	●	x	View/edit full name, email, phone number, gender, address, delete account
<b>Any-rank (randomly selected)</b>				
sendatext.co	●	●	N/A	View/edit SMS texts (sent and replies), calls, address book
metzlerviolins.com	●	●	✓	View/edit address, cart, wish list, password. View orders
swotanalysis.com	●	●	x	View/edit teams and members, billing history, projects
kokpit.aero	●	●	✓	View/edit full name, email, phone number, password, comments
brauchekondome.com	●	●	x	View/edit full name, address. View email, birth date, orders
socccerage.com	●	●	x	View/edit username, email, company name, address, cart, wish list, delete profile
packlane.com	●	○	N/A	View orders, saved designs
doggiesolutions.co.uk	●	●	x	View/edit full name, email, address, cart, delete profile. View order history
jellyfields.com	●	●	✓	View/edit email, username, website, favorites, password
helmetstickers.com	●	●	✓	View/edit full name, address, cart, password, delete profile. View order history

Access: full ●, partial ◐, none ○

the current password. For the manually selected popular domains, we get full access in five domains and partial access in four.

This highlights a significant advantage of cookie-based account hijacking over credential-based (e.g., phishing): additional fraud-detection checks employed during login [109] (e.g., IP geo-location [217], comparison of browser fingerprints [164]) are omitted because the cookies are part of a session that has already been verified as legitimate (i.e, when the victim logged in). While certain attackers can pass geo-location checks (e.g., using an IP address near the user’s location [205]), deceiving browser-based security checks is significantly more challenging. While spoofing the victim’s fingerprints has been theorized [102] it has not been demonstrated in practice. Surprisingly, throughout all our experiments we identified only one domain (Cloudflare) where we could not access the victim’s account from the attacker’s machine, indicating additional machine-specific checks that we have not come across in any other domain.

### 3.4 Discussion

**Automated account creation.** Our experimental evaluation revealed that automatically creating accounts is a significant challenge. While our current implementation allowed us to audit orders-of-magnitude more domains than prior manual studies [115, 238], we plan to explore the adoption of more sophisticated heuristics that automatically infer the predicates of account generation in a specific web app and create corresponding inputs. Automatically detecting and parsing error messages returned by the app can be used as feedback for inferring which form fields' format is violated. This, however, is a challenging task as, again, web developers are not constrained to a specific format or structure for returning such messages. Furthermore, each form input variation requires a form submission, which can lead to a significant impact to the overall performance and also trigger anti-bot mechanisms. Certain mandatory resources can also prevent our system from completing the process, e.g., an app may require a valid phone number in a specific country. While attackers can leverage “shady” phone providers [262], this remains an important obstacle for researchers.

**Privacy leakage inference.** Our system evaluates the leakage of personal or sensitive user information by detecting specific identifiers. In practice, information can be implicitly leaked, e.g., personalized results in search engines or e-commerce systems can reveal sensitive data (typically exposed through site-specific functionality). As part of our future work, we plan to explore the use of user-action templates that are based on the website's category (e.g., search engine, e-commerce), intended to elicit personalized results. Additionally, it is possible that some user information might already be publicly available on the same or a different website and, thus, the detected identifiers do not constitute actual *leakage*. While leakage can be highly contextual (e.g., a user's email address being publicly available in general versus a local eavesdropper being able to match that person to their email address) we consider this an interesting challenge and plan to explore the feasibility of detection schemes that disambiguate between public and private information.

**Countermeasures, disclosure, ethics.** Our framework discovered flaws that are exposing millions of users to significant threat. We emphasize that *no user accounts were affected* during our experiments – we only used test accounts. It is also crucial that developers are informed of our findings and address them. While the adoption of cookie security flags is more straightforward, correctly deploying HTTPS and HSTS will likely be more challenging for developers [118, 172–174]. For disclosure we leveraged the insight provided by prior work [184, 226, 255] and sent direct notifications to the affected domains for which we could find a valid contact email address. Specifically, we initially collected `security.txt` files [17], that typically include such contact points. This method proved to be the most ineffective, as such files are not widely adopted, i.e., only 23 domains had them. We then used an off-the-shelf email harvester tool for search engines [13]. Next, we crawled the websites starting

from their home page and visiting all contact related URLs, as well as the top 10 first level links. We also collected each domain's WHOIS record and searched for registered abuse addresses. We filtered all collected email addresses to ensure that they belong to the susceptible domain, so as to avoid sending our security-sensitive findings to unrelated parties. Overall, this process yielded 5,373 email addresses which we used for notification. For the remaining domains we sent our notification to standard aliases (`security`, `abuse`, `webmaster`, `info`) [226, 255]. We also manually searched for contact points for all domains we explicitly name in our work (apart from 2 that did not have a contact email or form). For the notification process we used an institutional email address to increase credibility and provided additional details and remediation advice to all websites that responded. All the responses we received acknowledged our findings, except one case where the developer persistently misunderstood the technical aspects of cookie hijacking. While we followed a best-effort approach to directly notify affected domains, it is infeasible to do so for all of them. Thus, we have also setup a notification service [19] where developers can obtain our reports after proving ownership of a given domain.

**HSTS issue.** During our experiments we uncovered an unexpected behavior in Chrome with HSTS preloading; we observed that it did not work as expected in slightly older Chrome versions and the initial request to a preloaded domain was, in fact, over HTTP. After communication with the Chromium team they informed us that their policy dictates that *any Chrome version more than 70 days old does not enforce HSTS preloading* because such hard-coded information is considered stale. This has significant implications for users that do not update their software on time, which is common behavior [194, 267, 274]. To the best of our knowledge this issue with HSTS has not been mentioned in prior studies.

**Code sharing.** Our browser automation tool has been made open source [31] as it can facilitate various research projects, especially those focused on Web security. However, publicly releasing our automated account creation modules poses a significant risk, as they are directly applicable to a plethora of real world attacks and could be misused for malicious purposes; the capabilities of our system far surpass the capabilities of such tools typically found in underground markets [209]. To that end, and to further contribute to the community, we have opted to make these modules available to vetted researchers upon request.

## Chapter 4

# Robust and Real-Time JavaScript Attribution

The web is driven by complex relationships and interdependencies of different parties. One of the earliest and most prominent scenarios for such dependencies are *third-party (3P)* scripts, where a *first-party (1P)* site includes JavaScript (JS) files from other domains. These scripts offer a multitude of additional functionalities, ranging from analytics [83] and advertisements [70], to Single Sign-On [36, 48] and fingerprinting [82]. 3P script inclusions have been prevalent for over a decade [202], and recent studies have found that inclusions have become much more prevalent and complicated, as 3P scripts can *implicitly* load additional scripts [158, 176, 266].

While 3P scripts facilitate development and offer rich functionalities, they come at a significant cost: once loaded, they operate with the *same privileges* as the 1P and can access the same information and functionality, making them seemingly *indistinguishable* at runtime. This is exacerbated by JS' dynamic nature, allowing scripts to execute further scripts at will [120, 160]. As such, buggy, compromised, or malicious 3P scripts can severely affect websites' security and privacy posture. For instance, they can introduce client-side vulnerabilities [158, 201], prevent HTTPS deployment [176], carry out cookie-stealing attacks [134] or invasively track and fingerprint users [119, 139, 159, 229, 257]. Therefore, robust 3P script attribution capabilities are *crucial* for building effective security countermeasures and conducting accurate web security measurements which often rely on analyzing 3P scripts' behavior. Unfortunately, no well-defined, standardized method for achieving this exists, and prior approaches do not fully achieve the desired results.

In more detail, existing countermeasures and 3P analyses relying on JS instrumentation [161, 201, 235, 266] often rely solely on *naive* stack walking for attribution, which is susceptible to bypasses when handling *dynamically* injected scripts [124, 160]. Another common pitfall is overlooking several dynamic JS-inclusion methods. Systems built with in-browser instrumentation are generally more robust when it comes to attribution, due to the browser engine's rich, low-level information (e.g., PageGraph [239], ScriptChecker [190]).

Unfortunately, such systems suffer from certain fundamental limitations. First, low-level modifications are inherently tied to a single browser version, hindering wide and fast adoption, while also requiring users to build their browser from source to utilize them. Moreover, wide deployment would require adoption by browsers *and* website developers [190, 273, 278, 282], which rarely occurs in practice [162], further diminishing their practicality. They can also require expensive, *offline* pre-processing [242]. As such, while extremely useful for certain measurements and studies, such systems cannot support real-time privacy and security enhancements for end users. Crucially, we also experimentally demonstrate that bypasses are feasible against such systems as well.

Motivated by the core problem of achieving *robust* and *real-time* 3P script attribution, we develop StyxJS. Our system, realized as a browser extension, operates at the JS layer without requiring any modifications to the browser. StyxJS' workflow includes stack inspection, on-the-fly script rewriting and JS API overriding, and is able to capture *all* 3P scripts in a page, including dynamically loaded ones. It achieves this in real-time and without any *a priori* knowledge of the encountered scripts. In addition, carefully designed tamper-proofing mechanisms ensure that our system is robust against evasion. At the same time, our design was guided by extreme precaution for respecting security mechanisms deployed by web apps. Moreover, due to its plugin-based design, StyxJS provides a convenient way for retrofitting existing approaches and countermeasures, as well as creating new ones.

We experimentally evaluate StyxJS and find that it does not disrupt the user experience, while incurring negligible performance overhead in terms of page load time in the vast majority of evaluated websites. Finally, we retrofit three existing systems as custom StyxJS plugins, one from the browser- and two from the JS-instrumentation family, which suffer from at least one of the aforementioned limitations (SugarCoat [242], LeakInspector [233], and ScriptProtect [201]). We experimentally demonstrate that the StyxJS-adaptation significantly outperforms the original system in its respective goal, while addressing *all* of their inherent limitations.

Overall, the concept of an *origin* is, arguably, the cornerstone of any notion of security on the web. While the ability to embed scripts from other origins is vital for the operation of the modern web ecosystem, it has also blurred the boundaries between origins. Consequently, the ability to effectively disambiguate between first- and third-party scripts, which is a *fundamental* prerequisite for countless security and privacy frameworks and countermeasures, has become exceedingly challenging. As such, we believe that StyxJS can provide the critical underpinning they require and have, thus, released it as an open source project [92].

## 4.1 Motivation

One of the core challenges of conducting a study or developing a countermeasure that revolves around JS execution, is correctly and robustly disambiguating 3P from 1P code. Systems leveraging JS instrumentation [161, 201, 263] must rely on *naive* stack walking to disambiguate 3P code, i.e., using the `Error.stack` object [47]. This approach is known to be susceptible to evasion [124, 160], as 3P scripts can *dynamically* inject new code, mask their identity, and evade attribution. For instance, as shown in Listing 4.1, a 3P script makes a call to `setTimeout` with a string argument, and the newly evaluated code outputs the current stack trace. As can be seen, the *original* 3P script is not included in the stack trace, thus preventing attribution and allowing the 3P script to conceal its malicious activities. Prior browser-instrumented systems [120, 160, 190, 230, 242] have rich, low-level information on seemingly every aspect of a page’s life cycle (e.g., DOM interactions) that can result in effective 3P code attribution. Unfortunately, such systems are tied to a single, modified browser, and possibly a single version, having no immediate impact on end users. Also, in §4.4.1 we demonstrate that low-level instrumentation is *not* self-sufficient for robust attribution, as faulty design choices can still lead to evasion and complete defense bypasses. Moreover, certain systems of both categories need a non-negligible amount of “offline” pre-processing to operate correctly [201, 242]. For instance, SugarCoat [242] requires a researcher to first *sufficiently* browse a website to generate replacements for intrusive 3P scripts, which, by design, can only cover executed code paths. To make matters worse, this pre-processing must be run for *each* new script, before its replacement is adopted by existing content blocking tools. Thus, it becomes clear that such systems’ real-world adoption and benefits for end users are significantly limited.

```
1 /* https://3p.com/evade.js */
2 setTimeout("console.log(new Error().stack);");
3 /* Output */
4 Error
5 at <anonymous>:1:13
```

Listing 4.1: The dynamically evaluated code via `setTimeout` cannot be attributed to the injecting script with `Error.stack`.

Motivated by the aforementioned limitations, we propose *StyxJS*, a JS-based solution which, at its core, aims to provide real-time and robust 3P script attribution. Realized as a browser extension, our system runs *without* any a priori knowledge or processing of the visited websites in *real time* and covers any executed code path. Along with a number of additional functionalities, such as its plugin-based architecture and instrumenting 3P code via on-the-fly, arbitrary code injection, *StyxJS* can be used to *properly* replicate certain prior approaches, as well as facilitate new pipelines, as we extensively demonstrate through real-world use cases (§4.4).

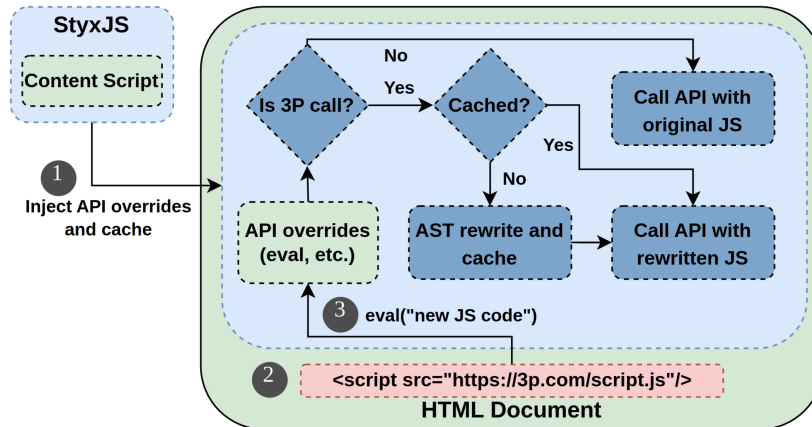


Figure 4.1: Architecture and workflow of StyxJS.

StyxJS targets both security and privacy practitioners, who can leverage it as the basis for new frameworks, as well as any researchers measuring and analyzing the web ecosystem. Most importantly, StyxJS can be promptly adopted by end users for employing the privacy enhancements we discuss in §4.4, or other defenses developed in the future.

## 4.2 Design and Implementation

StyxJS is comprised of different components for robustly capturing *all* 3P scripts in a page, regardless of *how* they were embedded. Before diving into the details of our approach, we provide a brief overview of StyxJS’ architecture and operation to clearly outline each component’s goals. A high-level depiction of StyxJS can be seen in Figure 4.1.

**Content script.** Our extension’s content script is injected in *all* documents and frames, and is configured to run *before* any other code in the page. This design choice allows us to inject our page scripts (which we describe next) in a timely manner, before malicious or privacy-intrusive scripts can execute. We provide more details on how we tamperproof our system against evasive scripts in §4.2.3. The content script’s main goal is to inject our attribution logic into the page itself, while also maintaining a cache of 3P scripts that were processed during previous visits to the same domain, so as to avoid redundant operations ❶.

**Page scripts.** The injected page scripts implement the entirety of StyxJS’s robust and real-time attribution capabilities. In more detail, a series of API overrides is set up, so as to capture *dynamically* injected 3P scripts, such as `eval` or setting a `script` element’s contents. Each script is then rewritten to include our attribution code, as we detail in §4.2.2 ❸. Moreover, StyxJS adopts a plugin-based architecture, allowing for additional, custom JS rewrites and API overrides. This, coupled with our robust attribution capabilities, constitutes a major advancement, as it enables *fine-grained* and *accurate* 3P code instrumentation, which is crucial for a wide range of past and future research efforts focusing on 3P scripts. We show how to leverage this capability to retrofit existing systems in §4.4, further highlighting



StyxJS' impact.

**Manifest V3.** Chrome has rolled out Manifest V3 (MV3) [55] and reportedly aims to force migration for all extensions, starting with Chrome's *pre-stable* versions in June 2024 [61]. Through empirical analysis we found that MV3 does not provide a foolproof way to inject scripts *before* any other code executes in the page, enabling potential StyxJS-evasion vectors. Specifically, a key change is that MV3 does not allow fetching and executing remote code. While our page scripts are listed as Web Accessible Resources (WARs) [67] and reside *locally* in the extension's directory, this change prevents us from fetching and adding them as inline scripts in the page. Adding them as `script` elements with their `src` attribute set to their local URL is allowed, but causes the browser to execute them *asynchronously* and possibly *after* page-controlled code has executed. While the newly introduced *userScripts* API [76] solves this issue, we found that it does not inject scripts in *blank* iframes. This also allows for trivial bypasses, as evasive scripts can acquire references to unprotected APIs through such frames. Therefore, since our main goal is *robustness*, we decided to develop StyxJS leveraging the well-established MV2. Nonetheless, we stress that our system has been designed with MV3 requirements in mind, and will be readily portable once the aforementioned *userScripts* issue is resolved. In the remainder of the text we will highlight the few operations that need porting to MV3 in the corresponding sections. Finally, we note that several browsers will maintain MV2 support [16, 33], further rendering our system practical until its migration to MV3.

### 4.2.1 Capturing 3P Scripts

Prior to performing attribution, we need to ensure that we capture *all* 3P scripts loaded in a page. Next we define our notion of 3P scripts, and detail possible script inclusion methods and how they are handled by StyxJS.

**3P scripts.** Strictly speaking, a 3P script is any remote JS file that does not share the same origin [62] with the visited page. However, websites often leverage servers with different origins under *their* ownership, for loading resources (e.g., CDNs to boost performance). For instance, Google services load scripts from *gstatic.com*, a resource server belonging to Google. Since the 1P is in control of such servers, it essentially makes their resources 1P and should be distinguished from *real* third parties. To that end, we leverage EFF's PrivacyBadger's list of *same-entity* domains, which is designed and maintained for this exact purpose [57].

**Standard inclusion.** A page can initially load a 3P script by embedding a script element with its `src` attribute set to the script's remote URL; this will cause the browser to send a HTTP request, fetch the script and evaluate it ②. For the remainder of this paper, we will refer to such scripts as *top-level* scripts. Attributing API calls back to such scripts is rather trivial and does not require any special handling. Specifically, we rely on JS' built-in Error object to acquire a stack trace and inspect the script URL in each stack

frame, if any. This approach was leveraged by prior work as the *sole* means for script attribution [201, 233, 235]. Unfortunately, while it is able to capture top-level scripts, it cannot robustly capture and attribute *dynamically* injected scripts, rendering prior approaches susceptible to evasion [124, 160].

```
1 let original = setTimeout;
2 setTimeout = function(){
3   let js = arguments[0];
4   if(is_3P_call()){ // Only for 3P calls
5     let cached = isCached(js);
6     if(!cached) js = rewriteJS(js);
7     else js = cached;
8   }
9   return original(js);
10 }
```

Listing 4.2: Simplified override for setTimeout.

**Runtime inclusion.** Scripts can introduce additional 3P code *at runtime* by utilizing APIs and DOM properties, e.g., adding a text node as a child on a script element, or calling `eval` ③. For brevity, in the remainder of this work we will overload the term API to describe all such inclusion methods, and refer to such scripts as *dynamic* scripts. We refer the reader to Table B.1 (Appendix B) for a complete list of JS inclusion APIs. We used the list provided by [201] as the starting point, but extended it to include overlooked APIs. Overall, our system handles 30 JS inclusion APIs from nine different interfaces. In order to capture 3P scripts injected via these APIs, we leverage the page scripts injected by our content script which runs on every new document *before* any other script (§4.2.3). Specifically, the content script, which has access to the DOM, injects a series of scripts that override each of these APIs. This ensures that whenever a 3P script calls such an API, our override code will execute before the original.

We provide an example API override in Listing 4.2, which captures dynamic scripts evaluated via `setTimeout`. Initially, to disambiguate whether the call was initiated by the 1P or a top-level 3P script, we simply inspect the Error stack trace (line 4). We detail how dynamic 3P scripts are attributed in the following paragraphs. If the call was indeed initiated by 1P code, we directly call the original API with the provided argument (line 9). For 3P calls, each override’s operation depends on the API’s nature. For APIs that expect raw JS code, e.g., `eval`, we directly rewrite the provided argument before calling the original API (line 6). For markup APIs, e.g., `document.write`, we parse the given markup, locate scripts, inline event handlers and `javascript: URLs`, and rewrite these specific parts. Finally, each rewritten piece of code (or markup) is cached in our extension’s storage space, keyed with the original source’s SHA-256 hash. Then, in subsequent visits to the same domain, each override computes the given argument’s hash value and queries the cache so as to avoid redundant

rewrites (line 5).

### 4.2.2 AST rewriting

Having gained control over when a 3P script is dynamically evaluated in the page, we then need to rewrite each one so as to add our attribution code, while maintaining its original functionality. We provide an example dynamic script which we assume to be injected via `document.write`, and its rewritten version, in Listings 4.3 and 4.4 respectively. Originally, the script declares and calls an asynchronous function, which awaits a *promise* and dynamically evaluates a new script. To achieve our goal, we utilize the *acorn* [69], *estraparse* [25] and *aststring* [38] libraries and perform the following steps. First, we parse the 3P source code to generate its AST, enabling us to perform fine-grained rewrites. Next, we wrap the *entire* script, and each function body, in a `try/finally` statement (lines 2-13, 5-8), where we prepend our entry code before the `try` (lines 1, 4) and our exit code inside the `finally` clause (lines 9,14). This approach guarantees that when the script or one of the functions exit, either gracefully or due to an unhandled exception, our exit code will *always* execute before anything else. Finally, we generate the final source code from the modified AST and return it.

```

1  async func foo(){
2    await ResolveAfter5Seconds();
3    eval("alert('Dynamically injected!');");
4  }
5  foo();

```

Listing 4.3: Example dynamic 3P script.

```

1  let gx = S.push("doc.write", ID); // entry code
2  try{ // Try/finally wrapper
3    async func foo(){
4      let fx = S.push("doc.write",ID,"foo"); // function entry code
5      try{ // Function try/finally wrapper
6        await xxawait(fx, ResolveAfter5Seconds());
7        eval("alert('Dynamically injected!');");
8      }finally{
9        S.pop(fx); // function exit code
10     }
11   }
12   foo();
13 }finally{
14   S.pop(gx); // exit code
15 }

```

Listing 4.4: Example dynamic 3P script after StyxJS' rewrites.

**Attribution code.** In order to attribute dynamic 3P scripts at runtime, we need a real-time stack trace, akin to the Error approach for top-level scripts. In other words, we need a 3P script or function to push onto a stack when executing, and pop from that stack when exiting. To that end, our page scripts also create a globally accessible stack interface; we detail how we conceal and protect our stack from malicious 3P scripts in §4.2.3. As shown in Listing 4.4, when a script is evaluated, our entry code will push a new frame in our 3P stack, including the API that injected the script and a unique script ID, *and* store its stack frame's index (line 1). The same applies for functions, which also store their name or a unique identifier for anonymous functions (line 4). Moreover, when a new frame is pushed, we mark it as the currently active frame *and* store the previous active frame, if any, as its parent. The exit code (lines 9,14), which aims to remove the frame, operates differently from a regular pop operation to account for asynchronous (*and* generator) function calls. If the exiting frame is not *last* but is followed by more recent ones, we can deduce that it must be, or was followed by, an asynchronous function that had suspended its operation until a promise was resolved. In the meantime, other dynamic 3P scripts or functions might have been executed and pushed as new frames in the stack. In such cases, we cannot simply remove the stack frame, as doing so would invalidate the stack indices stored by the following, more recent calls. Instead, we mark the frame as *completed* and return, as depicted in Figure 4.2. When, however, an exiting frame *is* the last active call, it simply pops itself from the stack *and* removes any previous, consecutive *completed* calls, so as to discard unneeded frames. Finally, upon exiting, we also iterate the chain of parent frames to find the currently active one; if none are found our active stack frame is set to null, indicating that *dynamic* 3P execution has halted.

**Async await calls.** Another challenge is the use of the `await` operator. In contrast to traditional promise-chaining, `await` suspends the execution of the *current function* until its operand is resolved. If in the meantime the 1P calls one of the overridden APIs, our 3P stack would be non-empty and we would wrongfully attribute 1P to 3P code. To tackle this, during rewriting we enclose each `await` operand as an argument into a custom function (line 6). This function marks the current frame as *suspended*, wraps the original awaited promise in a new promise and returns it to maintain functionality (Figure 4.2 (a) and (c)). When the original promise resolves, the wrapper promise will also resolve, propagate the results to the awaiting function, and mark the frame as *resumed* (Figure 4.2 (d) and (e)). As such, even if a 1P call to an API occurs, the 3P function is marked as suspended and, therefore, the API call is *not* attributed to the 3P. It is important to note that while static, this method is resilient against obfuscation, as `await` is a reserved keyword and cannot be concealed.

**Generators.** A challenge similar to the `await` issue stems from *generator* functions, which return an *iterator*. Such functions can use the `yield` keyword to gradually return values when the iterator's next method is called, and then suspend their operation until next is called again. To tackle this, we wrap each `yield`'s expression in a custom function

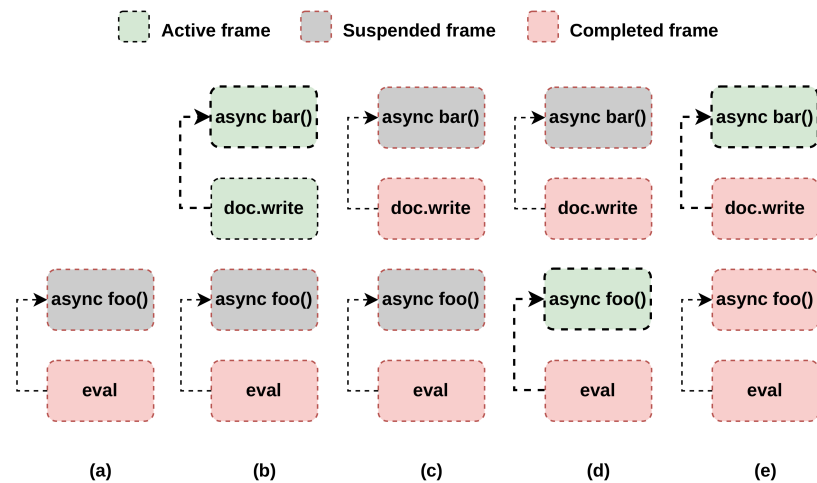


Figure 4.2: Example stack snapshot for `async/await` functions. (a) An `eval` script (completed) has previously called `foo`, which is now suspended due to `await`. (b) Later, a script injected with `document.write` executes and calls the asynchronous function `bar`. (c) `bar` is also suspended and the dynamic script exits. (d) `foo` resumes execution and exits. (e) `bar` resumes execution, exits and all completed frames are removed.

that marks the frame as *suspended* right before yielding the value. We also wrap the entire `yield` statement in a `try/finally` and mark the frame as *resumed* in the `finally` clause, as it is the first statement that executes when resuming the function.

**Nested dynamic code injection.** As can be seen in Listing 4.4 (line 7), the call to `eval` has *not* been rewritten. This is due to the fact that such *nested* dynamic code injections will be handled right before being evaluated, as described in the previous paragraphs. Specifically, the corresponding API override will check if our custom 3P stack has an active frame set, i.e., not *completed* or *suspended*, indicating that the call was initiated by a *dynamic* 3P script, and will proceed with rewriting the new script. We also construct a *JS Inclusion Graph* (JIG), a directed graph detailing the relationships between 3P scripts in the page, suitable both for offline *and* runtime analysis. In more detail, each graph node corresponds to a top-level or dynamic 3P script and is annotated with that script’s URL or inclusion method, as well as a unique ID and its parent’s ID so as to maintain script inclusion relationships.

**Dynamic naming.** We note that all pieces of code injected by StyxJS in these examples, e.g., stack interface and indices, `await`’s wrapper function, have been named statically for the sake of simplicity. In reality, as we outline next, they are dynamically named so they cannot be guessed and tampered with by evasive or malicious 3P scripts.

### 4.2.3 StyxJS Concealment

To guarantee robust attribution, it is crucial to ensure that StyxJS cannot be bypassed by malicious 3P scripts. To achieve this, we employ existing configuration and design choices [170, 201, 242], as well as novel techniques tailored to our system’s architecture and *modus operandi*.

**Basic protection.** First, we need to ensure that our content script runs *before* anything else in a document and promptly injects our attribution logic and overrides. To that end, we set the `run_at` field in the extension’s manifest file to `document_start`. To prevent iframe-related attacks, e.g., retrieving original APIs from a sub-document, we set `all_frames: true` and `match_about_blank: true`, so as to also capture nested frames. Furthermore, to restrict access to the original APIs and StyxJS’ variables, we utilize *Immediately Invoked Function Expressions* (IIFE), private blocks and block-scoped variables. To prevent malicious scripts from tampering with StyxJS’ operations by overriding built-in APIs, we maintain and use references to the *original* APIs, e.g., `Array.push`. Finally, to avoid interference with other installed extensions and ensure StyxJS executes first, we leverage the approach by Picazo-Sanchez et al. [216]. Specifically, we disable and then re-enable all other extensions, so as to appear as they were installed *after* StyxJS, and execute as such. We note that our API overrides *do not* disrupt other extensions’ functionalities, as they only consider calls initiated by 3P scripts.

**UXID token.** Our design necessitates the concealment of several operations, which is achieved through the use of a secret token. UXID is generated by the content script for every new document, and is sufficiently long (128 bits) so as to resist brute-force attacks. If a 3P script were to somehow learn its value, it would be able to circumvent StyxJS and avoid attribution. The content script sets each injected page script’s `id` attribute to UXID’s value so it can learn it too. After executing, each script removes itself from the DOM, so as not to leave any traces of its execution or UXID.

**Dynamic variable naming.** StyxJS needs to expose certain functions and variables that must be globally accessible by all overrides and the rewritten scripts (e.g., our stack interface and the original APIs). Naming these statically would enable trivial bypasses, as a 3P script could access the original APIs or pollute our stack. To that end, we bundle all functionalities under a single object named after the secret UXID, making it unguessable by malicious 3P code, and set it as a non-enumerable property of the window object. Moreover, since `Object.getOwnPropertyNames` and `Reflect.ownKeys` also return non-enumerable properties, we override them and strip our object from the returned property list. Finally, we perform the same steps for the JS libraries that our extension uses in a page [25, 34, 38, 69].

**Web Accessible Resources.** All page scripts, overrides and auxiliary libraries are listed as *web accessible resources* (WARs) [67] in our extension, making them accessible by *any* origin. As such, a script could probe for these WARs and detect StyxJS [169, 240]; however, it would not learn UXID as it is never statically referenced in any file. To prevent this, we lever-

age the fact that our content script executes before any other code in a document and we know precisely which WARs it will request. As such, we utilize the `webRequest` API [68] in our extension's background script and monitor the WARs requested by each unique document. If a WAR is requested more than once by the same document, we can deduce the request originates from a malicious script probing for StyxJS' presence and block the request. We note that this approach is not required for MV3, as it already provides the convenient `use_dynamic_url` manifest field [50] for preventing scripts from knowing and requesting WAR URLs.

**Detecting overrides.** API overrides can be detected by calling the `toString` method on them and identifying discrepancies [165]. To tackle this, we store the original string value for each overridden API and override `Function.prototype.toString` to return the expected result whenever it is called on one of the APIs. We also note that this approach covers `toString` itself; even if it is recursively called on itself, our override code will never be revealed.

**DOM exposing properties.** Another critical aspect that could lead to circumvention of StyxJS, are the inherent DOM modifications that occur when rewriting dynamically injected scripts. For instance, if a 3P script sets a script's `text` property, StyxJS' override for that API will rewrite the given JS code. The 3P script could then trivially access the rewritten property and learn UXID. We also note that a property's effects might be visible in other properties too, e.g., `innerHTML` will also reflect changes in `text`. Similarly, a rewritten node's ancestors will also reflect that node's real contents through such properties; we refer to all such properties as *exposing properties* (EPs) and detail them in Table B.1. To overcome this, we initially store the *original* value of all dynamic scripts, right before rewriting them. Moreover, after rewriting each script, we add two unique, randomly-generated markers as comments at the beginning and end of the modified source code. Then, we override all exposing properties' getter methods and internally retrieve the real, modified value when called. Finally, our overrides replace each piece of code encapsulated in our markers with the corresponding original value before returning it. We adopt the same approach for 3P functions, to prevent dynamic scripts from trivially reading their own (or each other's) rewritten functions and learning UXID.

**Frame polling.** A malicious script can continuously poll for the creation of iframes, so as to exploit a race condition between when a frame is created and when StyxJS' overrides are set, thus accessing the original, unprotected APIs. This can be achieved with a direct or recursive call to `setInterval` or `setTimeout`, respectively, with a zero delay. To prevent this, we initially block accesses to an iframe's `contentWindow` and `contentDocument` until the frame is fully loaded [170]. However, scripts can also access a frame's execution context by using `window[z]`, where `z` is a numeric index for each frame. Unfortunately, these indexed properties cannot be overridden to prevent direct access. To tackle this, whenever these functions are called, we wrap their callback in a custom function that inspects whether StyxJS has *not* been set up in any frame. In such cases, before calling the original

callback, it sets *all* protected APIs in the uninitialized frame as undefined to prevent evasive scripts from accessing them. It also stores the original APIs in a UXID-named variable in the frame itself, so StyxJS can restore and override them when being set up. To the best of our knowledge, this bypass vector has not been accounted for by prior work.

**Fingerprinting prevention.** Recent work [249] has demonstrated that extensions can be uniquely fingerprinted via their DOM modifications, even if they try to conceal them, e.g., by reverting them. This is achieved by leveraging the `MutationObserver` API [53], which records and inspects *all* DOM modifications with a given callback function. StyxJS' main operation is *not* affected by this approach, as we inject our logic *before* anything else in the page, i.e., before a `MutationObserver` can be setup. In addition, as mentioned previously, all API overrides call their original counterpart, thus not performing any extension-specific modifications. However, as we detail next (§4.2.4), there are a few cases where we need to set additional script attributes, so as to properly handle a page's deployed security mechanisms. To prevent StyxJS from being fingerprinted due to this behavior, we also override `MutationObserver` and wrap the page provided callback in a filtering function. This function removes mutations caused by StyxJS and calls the original callback with the filtered list of mutations. We also note that StyxJS is not affected by other extension fingerprinting techniques [178, 248], as it does not inject any stylesheets nor does it react to user actions. Moreover, our system is also secure against the attacks by [142], where malicious scripts escalate their privileges to the extension level via message-passing, as we do not exchange any messages. Finally, we note that detection through side-channels (e.g., timing-based) are out of scope for StyxJS, which is in line with prior work [56, 165, 170, 242].

#### 4.2.4 Maintaining Security Mechanisms

Agarwal et al. [100] recently found that several extensions carelessly modify page-deployed security mechanisms, diminishing their effects. In contrast, one of StyxJS' main requirements is to maintain the security guarantees offered by deployed mechanisms and *never* introduce new attack vectors, while still being able to operate as intended. As prior work has shown, composing such security policies is often challenging due to their complexity [226, 246] or third-party restrictions [252], resulting in developers' struggling with correct deployment [227] and subtle pitfalls that can even enable complete bypasses [275]. As such, properly *extending* these mechanisms to accommodate our system's operation also faces these challenges and requires careful design choices, so as to not overly relax them or completely break them. In the following, we detail how we handle the two mechanisms pertinent to our system's operation.

**Content Security Policy.** One of the most prominent security mechanisms is the *Content Security Policy* (CSP) [39], which allows a website to define what resources and from which origins they can be loaded in the page. For instance, it could dictate that scripts can



only be loaded from specific domains or if their source matches a set of predefined hash values. It is clear that if not handled correctly, such policies can hinder StyxJS' operation, as our rewritten scripts would not match any allowed hash value. On the other hand, overly relaxing a CSP would diminish its security benefits and potentially open up new attack vectors. Before delving into the details of our approach, we must note that we can only handle CSP headers upon receiving them, *before* the document is created, i.e., we do not have any knowledge of the 3P scripts that will be loaded in the page. Thus, whenever a main- or sub-frame request is intercepted, our background script detects any CSP headers and checks whether they set any directives pertinent to StyxJS' execution. Specifically, the following three directives regulate script execution; `script-src` [43] considers all JS execution in the page and serves as a fallback for the other two. `script-src-elem` [45] considers inline scripts, while `script-src-attr` [44] regards inline event handlers, e.g., `onclick`. Finally, `default-src` [42] serves as a generic fallback for all previous directives. If the CSP does not enforce any relevant restrictions, by not setting any of these directives, we do not need to perform any special handling.

*Inline scripts.* The first mechanism that regulates inline script execution are CSP-defined nonce values: if an inline script is *not* accompanied by a legal nonce, it will be blocked. In such cases, we do not need to perform any additional handling, since our operations do not affect nonces – it is up to the including script to assign scripts it injects with a valid nonce. Another way to restrict inline scripts, is defining specific hash values; if a script's contents' hash does not match one of the given values, it will be blocked. As such, StyxJS' rewritten scripts would not match any hash value and would always be blocked. To tackle this, we initially collect all hash values from the CSP and also extend it with a random nonce, unique to the document [54]. Then, whenever a new inline script is about to be injected, we compute its hash and compare it with the legal hashes. If it matches, indicating that the script should be executed, we add our custom nonce to the script and proceed with rewriting it as normal. This way, CSP's hash check will fail due to our rewrites, but the script will still execute due to our nonce addition. In case the script does not match any of the given hashes, we cannot be certain that it should be blocked, e.g., the including script might add a legal nonce to the new script *after* setting its contents. In such cases, we still rewrite the script, but do *not* add our nonce; if the script turns out to be allowed due to a nonce, it will still execute, otherwise it will be blocked as would happen without StyxJS.

*Inline event handlers.* Finally, the `script-src-attr` and `script-src` directives can specify inline event handlers' hash values with the `'unsafe-hashes'` keyword. The latter also regulates the execution of `javascript: URLs`. Similar to the inline scripts' approach, when an event handler is set on a node, e.g., via `innerHTML`, we perform the hash check internally. If the handler is allowed, we cannot extend it with our custom nonce, as it is not applicable in this case. Instead, we rewrite it and set it via `addEventListener`, which is not considered *inline* and is *not* bound to CSP restrictions [44]. If it does not match, we still rewrite

the handler, but set it via `setAttribute`, which is considered inline, so as to maintain CSP enforcement. Setting `javascript: URLs` in anchors' `href` attribute, essentially behaves as a click event. As such, we perform the same procedure, but instead of `href`, we set its `onclick` handler. Finally, for `javascript: URLs` in iframes' `src` attribute, we perform the hash check and, if allowed, we set the frame's `srcdoc` (which has precedence over `src` [1]) to an equivalent markup including the rewritten JS code as an inline script, carrying our custom nonce. We note that `srcdoc` iframes inherit their parent document's CSP [40], and thus, our nonce allows the script to execute.

**Trusted Types.** Another mechanism related to script execution are *Trusted Types* (TT) [95], which aim to prevent DOM-XSS vulnerabilities by restricting dangerous APIs to *only* accept trusted input. Specifically, a website can set two additional CSP directives: `require-trusted-types-for` to enable TTs, and `trusted-types` to define TT policy names to be used within the page. These policies are then defined in the page with the appropriate API, and can be used to create trusted inputs, i.e., by sanitizing the provided string argument and converting it to the corresponding TT [64–66]. If malicious code was to be injected in a DOM sink, it would either be sanitized or it would not be a TT; in any case, its execution would be blocked. To enable seamless StyxJS integration, we extend the `trusted-types` directive to also include a randomly generated policy name, which we also define in the page. This policy converts the given string to the appropriate TT without *any* modifications. Then, whenever a TT is provided in one of our overrides, StyxJS will first convert it to a string, so as to perform our rewrites, and then convert it back to its initial TT using our custom no-op policy, maintaining functionality. If the provided argument is *not* a TT but a regular string, we perform no conversions and handle it as such, maintaining the mechanism's security guarantees.

**CSP concealment.** We also account for scripts that might attempt to detect StyxJS by acquiring the modified CSP value and inspecting for our custom script nonce and TT policy. Specifically, a script can achieve this by registering a listener for `securitypolicyviolation` events [63], intentionally triggering a CSP violation and inspecting the resulting event's `originalPolicy` property. As such, we override the property's getter method and strip our modifications, effectively returning the original CSP. Finally, we perform the same procedure for script elements' `nonce` attribute.

**MV3 porting.** We note that our modifications on deployed CSPs, when needed, are performed on the fly using the `webRequest` API's blocking mode, which is no longer supported in MV3. In contrast, MV3's restrictive `declarativeNetRequest` (DNR) API [75] relies on predefined, coarse-grained rules to modify HTTP headers and does not currently support their conditional modification. Specifically, it only allows to entirely strip or replace an existing header, without providing more fine-grained control over its original contents. Nonetheless, this missing feature has been pointed out and requested by multiple extension developers and is planned to be integrated [73, 80, 81]. Once that is done, performing our CSP

modifications in MV3 will only require writing a simple regex-based rule that checks for the existence of allowlisted script hashes or the TT directive, and extending each one with our custom nonce and TT policy, respectively. Alternatively, we have also identified and experimentally verified another technique for achieving this. In essence, we can use the *non-blocking* `webRequest` API (available in MV3) to collect CSP header values and then completely strip them using the DNR API. Then, our content script can inject the modified CSPs as a `<meta>` element in the page, ensuring that all policies are correctly deployed before any code executes.

**CSP via `<meta>`.** While StyxJS is designed to carefully handle CSPs, it cannot handle CSPs deployed via `meta` elements in the HTML. This is because browsers will directly enforce the policy, and there is no foolproof way for us to modify it beforehand. Also, injecting additional `meta` CSPs can only make the initial policy *stricter* [41], while our approach aims to maintain the same level of security. This does not significantly impact our system, as prior work has shown that such CSPs are rare [226]. In addition, such CSPs can affect StyxJS *only* if they specify script hashes or the TT directive, since these are the only cases for which we need to extend the policy with our custom nonce or TT policy. Finally, even if we do not handle such a CSP, it would only cause breakage, but would not hurt attribution, i.e., scripts would be correctly captured by StyxJS, but they would be blocked by the browser.

#### 4.2.5 Summary

It is apparent that achieving *correct* and *robust* JavaScript attribution is not a simple task, as a multitude of different factors must be taken into account. Apart from the straightforward task of hooking relevant APIs for dynamic JS inclusions, one needs to effectively tackle several idiosyncrasies of the language itself, such as the `await` and `yield` keywords and their effects. Similarly, dynamic 3P scripts' (and functions') execution must be precisely monitored even in cases of early termination due to unhandled exceptions. Crucially, robustness further mandates careful design choices for tamperproofing the attribution scheme against evasive scripts. Finally, any system aiming to provide security guarantees should integrate seamlessly with *existing* security mechanisms deployed by websites and never diminish their own benefits.

### 4.3 Experimental Evaluation

Here we present our extensive evaluation of StyxJS. We experimentally prove the robustness and effectiveness of our system, measure the performance overhead it induces and, finally, study how its operation may effect real websites.

**Setup.** We ran all experiments using Chrome 122.0.6261.94, on a machine with a 16-core Intel Core i7-11700 @ 2.50GHz CPU and 62GB of RAM.

Table 4.1: Bypass techniques tested against StyxJS.

#	Bypass	Prevention
1	3P function toString	Uses override to remove rewrites
2	Access original APIs via iframe	StyxJS runs on all frames on document_start & blocks access until fully loaded
3	Enumerate window	Secret, non-enumerable properties
4	Injected scripts' sub-tree's and ancestry chain's EPs	Uses EPs' getter overrides to filter out rewrites

### 4.3.1 Validation

**Attribution.** To validate the efficacy of using StyxJS to capture and attribute 3P code, as well as to ensure that our system cannot be bypassed, we employed the following methodology. First, we created an HTML page that serves as the first party, which loads two external 3P scripts. These scripts declare and call a range of different types of functions (asynchronous, IIFEs etc.). Moreover, they leverage *all* JS inclusion methods detailed in Table B.1 and dynamically inject new scripts in the page. Each of these functions and scripts comprises a specific test. As aforementioned, we used the JS inclusion methods by [201] as our starting point and extended them to capture several overlooked cases. As such, while more comprehensive, there might still be cases we have missed. Nonetheless, StyxJS' modular design allows for the straightforward addition of missed inclusion methods, or even new ones that will be introduced in the future. In addition, all tests ultimately call `insertAdjacentHTML` so as to print their output on the 1P page, including the test's name and which top-level script initiated the action. We then created a plugin for StyxJS that overrides `insertAdjacentHTML` and validates whether the injected JS is correctly attributed to the appropriate top-level script, based on each test's information.

As an initial validation experiment, we first ran the embedded top-level 3P scripts sequentially, to allow each one to execute all of its tests in order. Next, to further stress the capabilities of our system, we also setup the two 3P scripts to perform their tests in an interleaved manner, by scheduling them via `setTimeout` in random intervals. Overall, this approach allowed us to verify that StyxJS passed all tests in *all* cases, indicating that it effectively captures both the top-level scripts and *all* dynamic scripts they inject, as well as all declared functions. In addition, StyxJS records the relationships between scripts and composes the JS inclusion graph; a shortened version of the JIG generated from the test page is depicted in Figure 4.3. All JIG nodes, corresponding to 3P scripts in the page, are annotated with their specific inclusion method (e.g., `eval` or the script's URL), whether they were fetched from the cache or were rewritten on the fly, along with their original source code. Moreover, each script is also annotated with a unique ID (*SUID*) and its parent's ID (*PSUID*) so as to maintain script inclusion relationships. As shown, StyxJS is able to cap-

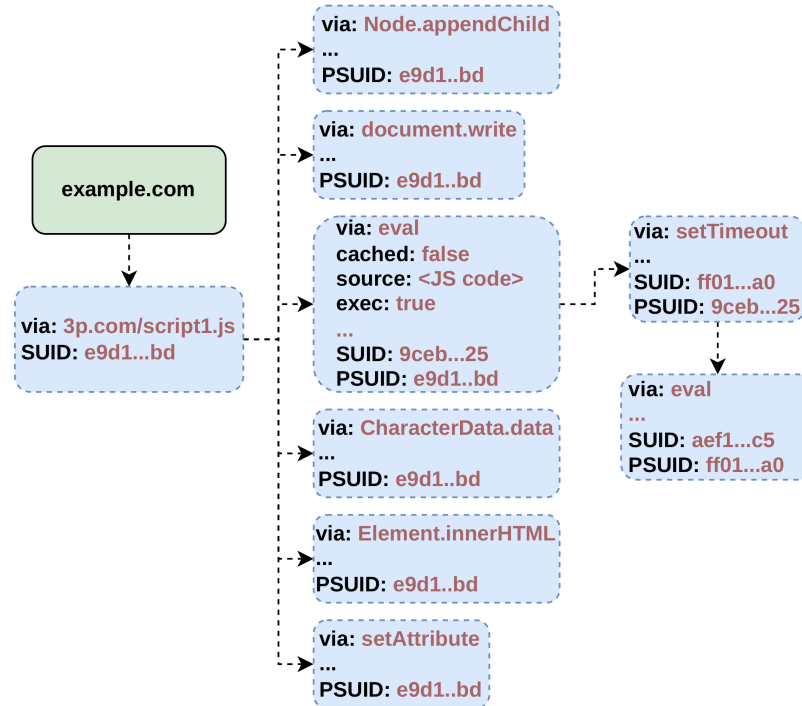


Figure 4.3: Test page’s JIG. We omit script node details for brevity.

ture dynamic script injection chains of arbitrary length. In this case, *script1.js* dynamically injects a new script via `eval`, which injects another script via `setTimeout`, which in turn evaluates another script. To account for JS that might have been injected but *not* executed (e.g., a 3P event handler that never fires), each node is marked as *executed* when the corresponding script is evaluated and its entry code runs. Finally, we also attempted to compare our system’s coverage against PageGraph [239] in real websites, but found cases where PageGraph yielded *incorrect* results, preventing us from performing a fair and correct validation; we provide more details in Appendix B.2.

**Robustness.** In order to validate our system’s resilience to bypass attempts, the 3P scripts perform additional tests, each trying to uncover UXID’s secret value. As discussed in §4.2.3, if UXID becomes known a malicious script can circumvent StyxJS. In Table 4.1 we detail each test and how StyxJS handles it. Specifically, the scripts attempt to read the rewritten 3P functions’ string representation, access unprotected APIs from same-origin frames, enumerate the window properties to uncover our stack interface and, finally, inspect the exposing properties of themselves, their sub-tree and their ancestry chain. Overall, UXID is never revealed due to the concealment techniques our system employs (detailed in §4.2.3).

**Security mechanisms.** To ensure that our approach described in §4.2.4 truly respects security mechanisms and also allows StyxJS to operate correctly, we setup our test page to deploy CSPs and TTs. For CSP we deployed a separate policy for each JS inclusion method, including inline event handlers and `javascript: URLs`, by specifying legal hash values.

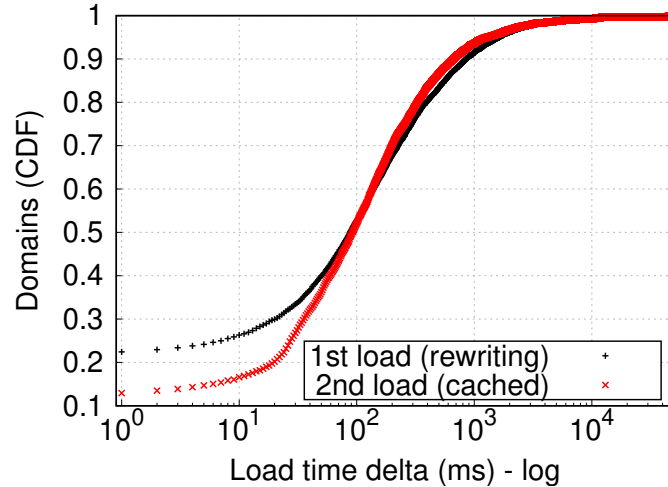


Figure 4.4: Page loading time delta for 1st and 2nd visit.

Specifically, we tested each inclusion method in isolation, i.e., only that method was allowed by the CSP, while all other scripts should be blocked. Similarly, we also adjusted the page to use TTs for the appropriate script inclusions. Then, for each mechanism setup, we loaded the test page both in a StyxJS-enabled and a vanilla browser, and verified that their results were consistent in all cases (i.e., legal scripts were correctly captured and executed, while illegal scripts were blocked).

### 4.3.2 Performance Evaluation

StyxJS' complex operation imposes a performance overhead on page load times. It is crucial to measure this overhead, since StyxJS can also be deployed in end user devices. To that end, we test our system on the top 10K domains on the Tranco list [37]. Specifically, we setup a browser instance without StyxJS, fetch the landing page and measure its loading time. Next, we refresh the page to measure the loading time after some resources have been possibly cached. Then, we perform the same steps with StyxJS in place; note that StyxJS caches dynamic scripts and does not rewrite them for subsequent page loads if they have not changed. We average the load times over five separate runs for each browser, to account for possible network delays or other issues, and measure the overhead StyxJS induces, i.e., the corresponding delta between the browser setups for the first and second page loads.

After excluding unresponsive websites (e.g., time out, DNS errors) and those without 3P scripts, we end up with 6,522 domains, shown in Figure 4.4. Regarding the first page load, where we expect a larger overhead since StyxJS needs to rewrite all dynamic 3P scripts, we find that 50% of domains require *at most* an additional 0.09 seconds to load, while 90% and 95% need up to 0.8 and 1.6 more seconds, respectively. For the second load, where our caching mechanism intervenes, half of the domains need up to an additional 0.09 seconds, 90% up to 0.65 seconds and 95% up to 1.2 seconds more to load. We observe that for half

of the domains the overhead remains the same for the first and second load, i.e., our cache does not seem to provide any benefits. This is expected, as we find that 95% of scripts can be rewritten in just 0.002 seconds, a minimal amount of time that cannot be significantly reduced. Yet, for a few domains (~10%), our caching mechanism slightly reduces the overhead, further improving the user experience and enabling wide adoption.

Overall, the overhead induced by our system for the majority of the domains is negligible, as StyxJS can accurately and robustly attribute web scripts, which is crucial for the accuracy of measurement studies and the effectiveness of countermeasures (as we demonstrate in §4.4). Nonetheless, StyxJS supports allowlisting domains in case a user observes prohibitively larger loading times. Finally, the storage footprint of our entire cache, spanning ~6.5K domains and ~188K dynamic 3P scripts, is roughly 400MB, making our approach suitable for end user deployment.

**Breakage.** To measure the breakage StyxJS might cause to real websites, we followed an approach similar to prior work [170]; while their work had a different focus, it also relied on a browser extension that heavily depended on API overriding. We randomly picked 50 out of the top 150 Tranco websites that include 3P scripts and proceeded to manually investigate them. We browsed each website *without* StyxJS, and recorded all encountered functionalities, e.g., signing up, logging in, editing settings, posting articles or comments etc. We stress that we did *not* set a time limit for this experiment, but tried to be exhaustive and uncover as many functionalities as possible. Next, having learned the website’s functionalities, behavior and appearance, we revisited it with StyxJS and exercised the same functionalities. After this procedure, we verified that StyxJS did not cause any breakage, as we did not encounter any appearance or functionality issues, thus not affecting the user experience. Coupled with the low, and typically negligible overhead, this further corroborates the suitability of StyxJS for end user deployment.

## 4.4 Use Cases

We designed StyxJS to have a plugin-based architecture, so as to simplify the creation of new pipelines on top of its robust and real-time attribution capabilities. Here we pick three existing systems, one that relies on browser instrumentation and two on JS instrumentation, as representative use cases. We present each system alongside its limitations, and demonstrate proof-of-concept (PoC) techniques for bypassing them. Next, we retrofit them as a plugin on top of StyxJS, to highlight how our system enables a wide range of studies and systems. We also provide additional insights and discuss our improvements over the original tools.

### 4.4.1 SugarCoat

**Overview.** SugarCoat [242] aims to address the privacy-functionality trade-off stemming from the explicit blocking of privacy-intrusive scripts, embedded in thousands of websites, since blocking them can cause significant breakage. This is achieved through safe *resource replacements* for such scripts, which can then be adopted by content blocking tools. In more detail, SugarCoat relies on a PageGraph-enabled browser [56] being driven by a privacy researcher so as to dynamically trace privacy-related API calls by harmful 3P scripts, and then map them to their JS source code. The target scripts are rewritten so as to redirect these API calls to *mock* API implementations, thus protecting privacy while also maintaining functionality.

**Limitations & bypasses.** While SugarCoat addresses a significant problem, it also suffers from certain core limitations. Initially, the generation of safe replacements happens offline; the replacements can only be adopted by existing content blocking tools at a *later* time. In other words, the system's benefits, while practical, are not immediate for end users. In addition, this offline pipeline must be run for *each* newly seen script, be it an updated or a completely new script. Finally, another core limitation, is the inherent shortcoming of its dynamic analysis approach, which *only* covers executed code paths. As such, SugarCoat's effectiveness in generating *complete* and *safe* script replacements, heavily depends on its pre-processing's coverage, as we outline next. Based on these limitations and our analysis, we have devised bypass techniques that a malicious 3P script can use to avoid attribution and maintain access to privacy-sensitive APIs.

*Executed code paths.* SugarCoat can only capture accesses to privacy sensitive APIs that were executed during its offline pre-processing. A malicious 3P script can exploit this limitation and alter its behavior depending on whether it executes in SugarCoat's PageGraph-instrumented browser or if it is a SugarCoat-generated replacement. To illustrate this, consider a 3P script that originally has the following structure:

```
1 func oracle(){ return !cookie.includes("abc"); } // Returns false
2
3 if(oracle()){ // false. 'else' clause is executed
4   func snoop(){ return storage.getItem("KEY"); }
5 }else{
6   func snoop(){ return storage.getItem("KEY"); } // Declare same function
7 }
8 let secret = snoop();
```

Listing 4.5: SugarCoat-evading 3P script.

The script leverages an oracle to determine whether it's a SugarCoat replacement or not. In this case, the 3P author has preemptively crawled a large number of domains embedding her malicious script, and has collected information on what cookies each one sets. As a result, if the script runs in the PageGraph browser, which does not impose any API restrictions,



it should be able to access this information (line 1). Then, regardless of the result, it performs the exact *same* operation, e.g., accessing a sensitive value in `localStorage`, through the `snoop` function (8); the difference lies in *where* this function is declared (4, 6). When the script is initially analyzed, the oracle will determine it is *not* a replacement and SugarCoat will map the `localStorage` access to the `snoop` function declared in the `else` clause (6). Therefore, the script replacement will have the following structure (simplified for brevity):

```
1 mockCookie={...}; mockStorage={...}; // SugarCoat mock APIs
2
3 // Returns true due to mock cookie API
4 func oracle(){ return !mockCookie.includes("abc"); }
5
6 if(oracle()){ // true. 'if' clause is executed
7   func snoop(){ return storage.getItem("KEY"); } // Not replaced with mock API
8 }else{
9   func snoop(){ return mockStorage.getItem("KEY"); }
10 }
11 let secret = snoop();
```

Listing 4.6: SugarCoat-evading 3P script replacement.

When adopted by a content blocking tool and ran in a user's browser, the replacement script will again call its oracle. However, the `cookie` API has been replaced with a mock implementation (line 4 in Listing 4.6) and the oracle will not be able to retrieve the expected cookie, indicating that this is in fact a SugarCoat replacement script. As such, the `snoop` function, now declared in the `if` statement (7), will access the *original* `localStorage` API, fetching the `secret` value and effectively bypassing SugarCoat's protection. We note that the oracle in this example is rather simplified for demonstration purposes. In practice, since the mock APIs adopted by content blocking tools are public knowledge, 3P script authors can implement much more elaborate oracles, e.g., by leveraging several APIs, looking for discrepancies that indicate if their script is in its original form or a replacement.

*Stringifying functions.* A more direct approach to bypass SugarCoat relies on its oversight to prevent scripts from retrieving their functions' string representation. Specifically, SugarCoat assigns the original API to a variable, so as to restore it after the script's execution, and dynamically names it to prevent scripts from guessing and using it directly. However, a script can achieve this by calling a function's `toString`, extract the variable's name and restore the original API right before using it. We also note that this approach can be used as a generic oracle for the first bypass technique we demonstrated, as a 3P script can trivially check if one of its functions has been rewritten by SugarCoat.

*PageGraph oversights.* Another limitation we identified, which also leads to a trivial bypass, is that PageGraph fails to correctly capture string-to-code evaluations via `setTimeout` and `setInterval` calls, as well as `javascript: URLs` in anchors' `href` attribute. Specifically, the system adds each script as a node in the final graph and connects it to the rest

of the graph with an *execute* edge, allowing it to attribute the injection back to another node. In contrast, the aforementioned script inclusions are not connected to any other node, preventing attribution. As a result, a 3P script can simply inject all of its functionality, including accesses to privacy sensitive APIs, through a single call to one of these functions and evade SugarCoat’s rewriting. It is important to note that this specific bypass technique is not tied to SugarCoat, but affects any system that relies on PageGraph for script provenance [120, 166, 230, 239, 243].

We stress that we experimentally verified each bypass technique, by writing PoC evading scripts that access sensitive APIs and then generating their replacements via SugarCoat. Our findings were disclosed to and acknowledged by the Brave browser [91], which issued fixes for the PageGraph-related bypasses and awarded us with a bounty.

**StyxJS adaptation.** Adapting SugarCoat on top of StyxJS was straightforward, as we only needed to implement the mock APIs, adjust them to utilize our 3P attribution and inject them on each page right after our overrides (§4.2.1). SugarCoat’s pre-processing is not applicable in our case, as StyxJS operates in real-time and does not require *any* prior knowledge of the websites or the scripts they include.

In more detail, we created mocks for the same APIs as [242] (storage, cookie, navigator, XMLHttpRequest, and Fetch), but instead of *replacing* and *restoring* the APIs, we override their prototypes’ properties and functions to disambiguate between 1P and 3P calls at runtime, by utilizing our 3P attribution scheme. Then, whenever an API is called by a target privacy-intrusive 3P script, the override will redirect the call to the mock API instead of the real one. This has two significant advantages; first, we do not need to locate the source locations where the API is called since we rely on dynamic code interception and, thus, cover *any* executed codepath. This makes our approach robust against evasion techniques such as the one presented in Listing 4.5. Moreover, it solves another limitation, related to async calls. As [242] mentions, if a 3P *async* function accesses a protected API and its execution is temporarily suspended, other benign code will access the mock API instead of the original, as it has not been restored yet, until the async function resumes and completes. Disambiguating API calls at runtime, coupled with our async/await handling (§4.2.2), solves this issue as the async function will be marked as suspended, enabling correct API call attribution. Overall, our SugarCoat plugin, including mock implementations and recording protected API calls for analysis, totals 274 lines of code (LoC), while the original mock APIs [30] need 465 LoC. This is because SugarCoat has to replicate each API in its entirety and properly mimic its behavior, due to its replace & restore approach. In contrast, StyxJS essentially extends the original API and can utilize existing properties. For instance, a script can create a regular XHR object without any intervention from StyxJS, but sending the request will be blocked by the corresponding override.

*Comparison.* Next, we utilized our plugin to measure its effectiveness and compare it to the original version. Initially, we collected the filter list rules generated by SugarCoat [29],

Table 4.2: API calls captured by SugarCoat and StyxJS.

	Network	Storage + cookie	Navigator
SugarCoat	2,494	15,577	9,103
StyxJS	3,547	76,904	64,865

which define specific privacy-invasive scripts that should not be blocked for compatibility. Next, we collected the current script *exception* rules from the same filter lists as [242] (EasyList [78], EasyPrivacy [79], Brave [84] and uBlock Origin (uBO) [96]), so as to find domains that likely included these scripts at the time of our experiment. This procedure yielded 3,025 domains. We then set up a browser with uBO, configured to redirect target 3P scripts to the *original* SugarCoat replacements [28, 29], as intended by their system. We also modified SugarCoat’s script replacements to record API accesses. Next, we visited each domain and waited for 5 seconds after the load event to allow scripts to execute. Finally, we repeated the same process for these domains with StyxJS’ plugin, targeting the *same* 3P scripts. Overall, we found 1,616 domains that were responsive and included at least one target script.

As shown in Table 4.2, the API accesses protected by the original SugarCoat scripts are significantly fewer than the ones covered by StyxJS. To further shed light on the underlying causes of this discrepancy, we sampled and manually inspected 20 of SugarCoat’s script replacements in an attempt to identify whether relevant API call sites were missed during its offline pre-processing. Through this process, we found that 18 out of 20 script replacements actually missed *several* privacy-invasive API calls and only covered a subset of them. For instance, for the popular Google Analytics script, SugarCoat captured a few calls to `navigator.userAgent` and `XMLHttpRequest`. However, the script also reads and writes cookies several times and sends additional requests, which were not replaced by SugarCoat’s mock APIs. In another script, apart from missing API calls to `storage`, `cookie` and `navigator` properties, the script replacement threw an exception due to SugarCoat’s rewrites, preventing it from fully executing and possibly reaching further API calls. These findings further highlight two crucial SugarCoat shortcomings. First, scripts do not always necessarily expose *all* of their functionalities, making offline pre-processing inadequate. Second, SugarCoat’s script replacements were generated over three years ago, making them obsolete and error-prone. It also highlights that generating one-off script replacements is not a robust, long term solution, as one needs to *constantly* generate replacements for new or updated scripts (and push them upstream to content blocking tools). To better illustrate this limitation, we collected the scripts that current filter lists exempt from blocking and visited them daily for one week. We found that from 238 scripts that were responsive, 10% were updated once, while another 10% changed multiple times. In other words, SugarCoat would have to pre-process these scripts several times in a single week to accurately capture their current version. In contrast, StyxJS does not suffer from these limitations as it captures the *current* 3P script versions in real time, without prior processing, and covers *all* executed code paths.

When comparing our plugin's performance overhead to the original system, we find that for the first page load it requires up to 0.16 and 1 seconds more for 50% and 90% of the domains respectively. For subsequent loads, the overhead is reduced to up to 0.12 and 0.77 seconds more for the same fraction of domains. We also performed a breakage comparison on 50 randomly chosen domains, to test whether StyxJS causes additional breakage. We visited each website with both the original system and StyxJS' plugin and exercised the various functionalities. We found one domain in which images would not load correctly with SugarCoat, possibly due to its script replacements being outdated. Most importantly, we encountered no issues stemming from StyxJS' operation.

*Further insights.* Since SugarCoat-generated rules and script replacements target *specific* scripts, without accounting for dynamic GET parameters or paths in script URLs (e.g., client IDs or version numbers), we relaxed the filter rules to account for such cases by replacing dynamic parts with wildcards. We also collected the scripts that are currently exempt from blocking in the various lists. We then set up StyxJS to run for all 3,025 domains, out of which 2,127 (70%) included at least one privacy-harming, but compatibility-critical script, which current filter lists allow. Overall, our plugin prevented 7,184 network related, 274,284 storage and cookie and 236,073 navigator API calls, *without any* a priori analysis of the scripts. This further corroborates StyxJS' practicality and immediate privacy benefits for users, as it can promptly target such scripts solely based on their URLs.

#### 4.4.2 LeakInspector

**Overview.** Recently, Senol et al. [233] studied the prevalence of online trackers exfiltrating personal data from HTML forms *before* their submission. Specifically, they measured the exfiltration of e-mail addresses and passwords and found several cases where trackers or session replay scripts extracted both, either intentionally or inadvertently. To aid users in remediating such privacy-invasive practices, they released LeakInspector [32]. This browser extension aims to detect and, optionally, block tracker or 3P scripts from sniffing sensitive elements, pertaining to personally identifiable information (PII), and sending their value to known tracker domains. This is achieved by detecting such input fields in a page, defining a custom value getter method on *each* one, and attributing the read operation to the calling script. It also collects the value of all protected fields and inspects outgoing requests to detect and block encoded, hashed and plaintext leakage to trackers.

**Limitations & bypasses.** LeakInspector relies on naive stack walking for attribution when detecting such read operations, by *solely* leveraging `Error.stack`. As mentioned in §4.1, this leads to trivial bypasses through the use of dynamically injected scripts, effectively nullifying its defenses. For instance, a malicious script can still read sensitive input fields' values by simply injecting a new script through `setTimeout` or `document.write`. Moreover, we have identified several other technical malpractices that enable other types of trivial by-

passes. First, as aforementioned, LeakInspector defines a custom value method on each PII input field, but does not take precaution to override the corresponding method on the `HTMLInputElement` prototype itself. As such, a malicious script can acquire the prototype's method and call that instead of LeakInspector's custom method. Even if this issue was addressed, the system is configured to run on each new document on `document_idle`, after attacker-controlled code has executed. Therefore, a malicious script would still be able to acquire an unprotected reference to `HTMLInputElement`'s `value` method. Similarly, the same can be achieved by acquiring the API reference through a sub-document, as the system has not been configured to run in *all* frames and does not defend against frame polling. In addition, LeakInspector uses a direct reference to `Error.stack`, allowing an evasive script to override the stack trace, remove its information, and avoid attribution. Finally, the system relies on common encodings and hashing algorithms to detect *and* block PII leakage at the *network level*. As such, a script can utilize any of the above bypass techniques to read PII values and avoid detection, and simply utilize unsupported or nested encodings to send them to tracker domains. As in the previous use case, we have setup a PoC script that employs all bypass techniques and experimentally verified each one.

**StyxJS adaptation.** We retrofitted LeakInspector's approach as a new plugin on top of StyxJS, amounting to 158 LoC. Moreover, we configured the plugin to also inject Mozilla's Fathom e-mail field detector [14] in each page, as done by the original system. Instead of relying on HTTP request inspection to detect and *block* PII leakage, which is prone to evasion, we decided to tackle the root cause of the issue. Specifically, we aim to *robustly* prevent 3P scripts from reading sensitive fields' values in the first place. To achieve this, we initially override the `HTMLInputElement` prototype's `value` getter method and attribute reads by 3P scripts using StyxJS' underlying infrastructure. Our override also identifies whether the input field being read constitutes PII, using the same heuristics as [32]. This covers several different types, such as e-mails, passwords and credit card numbers. If the call originates from a 3P script and targets a PII input field, our override will return a predefined dummy value instead of the real one and will also inform the user by highlighting the target element and annotating it with information about the sniffing attempt. In addition, we override the `InputEvent` prototype's `data` getter and perform the same steps, in order to prevent scripts from registering `oninput` event listeners and gradually constructing the element's value. We note that this exfiltration method has not been accounted for by LeakInspector. Moreover, our plugin targets *all* 3P scripts, but can be trivially adjusted to only prevent known tracker scripts from reading sensitive values. Finally, we leveraged our PoC script and verified that StyxJS effectively prevents *all* read operations that LeakInspector previously missed.

Table 4.3: Comparison between ScriptProtect and StyxJS.

	ScriptProtect	StyxJS
<b>Domains</b>	112	169
<b>Scripts</b>		
- Blocked	12	22
- Sanitized	102	161
<b>API accesses</b>		
- Blocked	18	140
- Sanitized	719	1,775

### 4.4.3 ScriptProtect

**Overview.** ScriptProtect [201] aims to prevent *benign-but-buggy* 3P scripts from accidentally introducing DOM-XSS vulnerabilities. It achieves this by overriding dangerous HTML sinks that can lead to new JS being evaluated (e.g., `innerHTML`) and sanitizing their input, while APIs that lead to direct JS execution (e.g., `setTimeout`) are entirely blocked. The system offers two modes of operation: for new or relatively short codebases, 1P developers can explicitly use *unsafe* API variants (e.g., `document.unsafeWrite`), essentially calling the original sink. For existing applications, where rewriting calls to unsafe counterparts can be costly, it dynamically disambiguates 3P from 1P calls inside the sink overrides by leveraging the stack trace provided by the Error object [46].

**Limitations.** Unfortunately, for the dynamic 3P attribution mode, ScriptProtect only inspects the top-level stack frame to decide whether a call to a DOM sink originates from a 3P and should be sanitized or blocked. While this makes sense in certain cases, a 3P script could *inadvertently* evade ScriptProtect and introduce a DOM-XSS vulnerability, in case 1P code calls one of its functions. Such 1P calls to 3P code is common practice for many 3P scripts such as analytics, SSO and DOM-manipulation libraries [36, 48, 49, 83]. Moreover, ScriptProtect has overlooked *several* JS inclusion methods (see Table B.1) and in some cases only partially captures them – we have found cases where their interception fails and new JS code is executed. This further allows 3P scripts to unintentionally circumvent its defenses and introduce DOM-XSS bugs. We have experimentally verified these limitations.

**StyxJS adaptation.** To retrofit ScriptProtect on top of StyxJS, we created a plugin (110 LoC) implementing all their dangerous sinks’ overrides, as well as the ones that they missed, and adjusted them to use our 3P attribution interface. It also injects DOMPurify [85, 154] in each page for the markup APIs’ sanitization step, as per the original system.

To showcase StyxJS’ ability to robustly and conveniently retrofit existing pipelines, but also combine them or introduce new ones, we decided to target outdated JS libraries for our experiment, as they are more likely to be vulnerable, an issue highlighted by prior studies [179, 201, 254]. To that end, we slightly modified the original system to target *specific* 3P scripts, and modified DOMPurify to use references to *original* APIs, as we found cases where ScriptProtect would wrongfully attribute such calls to 3P scripts. To construct a dataset of

outdated JS libraries we referred to [179], which reported the 30 most popular libraries of the time. We also gathered the 20 most currently-popular libraries (according to Wappalyzer [87]). Next we removed duplicates, NodeJS modules (which are irrelevant in our context), and well-known, functionality-critical libraries that might entirely break a website if tampered with (e.g., jQuery [49], RequireJS [58]); this left us with 32 JS libraries.

Next, we visited each library’s repository and gathered the version numbers that have not been updated in over two years. We also identified URLs commonly used to include each library in the wild, based on repository documentation and relevant StackOverflow questions. Based on these URLs and versions, we compiled a list of regex-based filter rules that we fed into our plugin and the original system, in order to capture these scripts specifically. We note that our dataset is rather limited, as we might have missed versions, URLs and even entire libraries. However, our goal is not to study the use of such libraries, but to compare StyxJS’ effectiveness over ScriptProtect in a realistic use case; as such, we believe the compiled dataset serves our goal. Finally, we visited the landing page of the top 3K domains with both systems and recorded protected API accesses; this process yielded 169 domains that included one of the outdated libraries.

As shown in Table 4.3, StyxJS offers a significant improvement over ScriptProtect as it blocks approximately 8 times more and sanitizes 2.5 times more API calls, while covering 57 more domains where at least one protected API was called. StyxJS also captures 10 and 59 more scripts which called at least one blocked or sanitized API, respectively. Regarding page load times, we observe that our plugin takes up to 0.07-0.12 and 0.7-0.73 seconds more for 50% and 90% of the domains respectively, for two subsequent page loads, once again highlighting the practicality of our approach.

To investigate possible breakage due to StyxJS, we analyzed 50 random domains with both systems. We found 19 cases where the original system threw an exception and caused minor breakage, such as appearance issues, submitting cookie policies, playing videos, and breaking the search functionality. Moreover, in five of these domains, ScriptProtect caused their navigation menus to not load properly, effectively disabling access to other pages. In contrast, we did not encounter any such issues with StyxJS in *any* of the domains.

## 4.5 Discussion & Limitations

In this section we discuss limitations of our approach and a potential direction for wider adoption.

**Eval.** A common issue in overriding `eval` is that it is not executed in the same scope as the code that called it [201, 263], which might cause breakage. Recently, [230] showed that only 1.9% of 1P *and* 3P scripts use `eval`; even this small fraction is an upper bound, as not all calls will cause breakage. Indeed, during our large-scale performance evaluation (§4.3.2) only ~11% of all `eval` calls threw an exception. Again, this is also an upper bound as we do

not expect all cases to be caused by StyxJS.

**JS bundling.** If a website bundles several JS files into one, StyxJS would be unable to attribute API calls to their true origin. The same applies if the 1P hosts 3P scripts in its own domain or inlines them in its HTML. This is an inherent limitation of *any* system attempting script attribution and is not tied to StyxJS' approach. Nonetheless, recent work [218] has shown that 3P bundles are way more prevalent than 1P ones; in such cases StyxJS can capture the entire bundle, but not distinguish individual scripts. Finally, as proposed by [218], studying bundles' API usage so as to reduce their security impact on websites is an interesting future direction that could significantly benefit from StyxJS' capabilities.

**Browser adoption.** Ideally, web script attribution could be addressed internally by browsers and exposed via a *native* JS API as a substitute for StyxJS' capabilities. Moreover, such an addition could potentially solve our aforementioned limitations, while also being extremely useful for security and privacy practitioners, as it would facilitate the development of a plethora of different countermeasures. As such, having established the significant improvements StyxJS offers, we hope our work can incentivize browser adoption.



# Chapter 5

## Related Work

### 5.1 Black-Box Web Application Scanning

Web application scanning has received considerable attention from the research community through the years, as both black-box [132, 137, 141, 167, 213, 253, 256] and white-or grey-box [104, 143, 150, 152, 155, 156, 185, 207, 224, 234, 264, 269] techniques have been proposed and thoroughly evaluated. Moreover, several studies have carried out extensive comparisons between black-box vulnerability scanners [105, 133, 208, 260, 270], collectively agreeing that such tools suffer from certain core limitations, such as detecting and correctly modelling all injection points, replaying the necessary steps to perform an injection, or their inadequacy on persisting the authentication state. In the following paragraphs, we relate our work to the most prominent black-box approaches proposed in prior work.

As early as 2006, Kals et al. [167] designed a simple web application vulnerability scanner that visited pages, extracted HTML forms and tested for common XSS and SQL injection payloads. Later, Doupé et al. [132] highlighted the importance of taking the application's state into account for better coverage and implemented their approach in a *state-aware crawler* which, however, only considered static HTML links and forms for detecting state changes, navigating *and* clustering similar pages together. As we outlined, these features only, fall short for navigation and can also incorrectly cluster pages that offer different functionality (e.g., through input elements or buttons *outside* an HTML form). In another line of work, Duchéne et al. [136, 137] inferred a control flow model for the application under test and an attack grammar to generate appropriate payloads for detecting XSS flaws; however, their approaches do not consider client-side events and require the ability to reset the application. Pellegrino et al. proposed jÄk [213], which considered client-side events towards covering a larger part of the application, but did not use a fully-fledged browser and only considered reflected XSS. More recently, Eriksson et al. [141] developed an XSS scanner that tackled some of the limitations we address in our work. We provide a detailed comparison in §2.3.3. Finally, a multitude of other works have focused on identifying specific flaws, such as client-side XSS [183, 253, 256], CSRF [112, 171] and unrestricted file uploads [121, 181].

The common thread among all these works is that they suffer from at least one of the

core limitations we highlight in this study. Moreover, they all constitute standalone tools that target a specific selection of vulnerability types. In contrast, ReScan aims to address these challenges in a scanner-agnostic manner irrespectively of the specific tests carried out by each scanner.

URL clustering has also been used in different domains, e.g., the detection of phishing pages. Such systems deduce page similarity either through *visual* analysis [98, 126, 188, 189] and comparing benign to suspicious webpages, or by utilizing URL and HTML related features that mainly focus on a page's textual contents [180, 186, 268]. Despite being successful for their respective goals, such approaches are not suitable in our context. Specifically, a visual representation of a page might not reflect the subtle yet important elements that denote different functionality, such as *hidden* buttons or input elements [187]. Moreover, two pages' textual contents might differ significantly even in conceptually similar pages (e.g., two product pages with different reviews). Fundamentally, our system aims to cluster pages that essentially offer the same *functionality* regardless of their specific contents and precise appearance.

## 5.2 Web Authentication and Authorization

**Cookies and Sessions.** Several prior studies have explored certain aspects of authentication and authorization flaws in web apps. Sivakorn et al. [238] manually audited 25 popular domains (and their respective mobile apps and browser extensions). Calzavara et al. [115] recently implemented black-box strategies for identifying session integrity flaws using a browser extension, and audited 20 popular websites where they found several vulnerabilities under different threat models. However, the most challenging parts of the process are not automated and app-agnostic (e.g. account creation, status oracles), rendering large-scale deployment and analysis infeasible. Neither of these studies included the JavaScript-based threats that we explore. In another work, Calzavara et al. [112] conducted a large-scale study on TLS vulnerabilities that can enable session hijacking. Kwon et al. [177] exploited the shortcomings of a specific TLS cipher suite and proved that, under certain assumptions, it is possible to disable cookie attributes in HTTPS traffic. Finally, Jonker et al. [151] proposed Shepherd, a system for automated login that can enable post-login studies. However, their system does not handle account creation which is the most challenging process. Instead, it relies on the BugMeNot service [74] for acquiring credentials, which further limits their system's coverage as that service explicitly prohibits websites that are high-value or have user accounts with sensitive information, thereby limiting the overall suitability of their system for representative large-scale studies. Moreover, they located valid credentials for 26,758 domains of Alexa's Top 1M and managed to log into 7,113 (26.6%) of them; this is more than three times less than our audited websites (which include numerous popular and interesting domains prohibited from BugMeNot). Calzavara et al. [251] leveraged and extended

Shepherd to perform a large-scale measurement on several session security aspects, including session fixation vulnerabilities, transmitting passwords over unencrypted connections and session invalidation malpractices upon logout and also revisited our session hijacking attacks, all one year later after our own work. The authors managed to log into 6.1K websites (1K less than Shepherd and four times less than our audited websites) and their approach suffered from the same inherent limitations. More recently, Rautenstrauch et al. [220] performed a comparative analysis on several security aspects in both the pre- and post-login states of websites, such as security headers, use of vulnerable JavaScript libraries and number of JavaScript inclusions and client-side XSS. However, their system relied on manual account registration, limiting their scope to ~200 websites. A notable exception is the study by Al Roomi et al. [225], carried out in 2023, which automatically evaluated website login policies at scale, e.g., hosting login pages over HTTP, storing plaintext passwords and the capability to enumerate registered users, which are orthogonal to our work. While their system adopted similar techniques to ours for automated account setup, it managed to register accounts in 45K websites. This improvement can be largely attributed to their system being able to solve CAPTCHAs, one of the main reasons for failed registrations in our case.

While these studies provide useful insights, they are inherently small-scale, require significant manual effort, or are complimentary to our work as they focus on different problems that enable session hijacking (e.g. TLS vulnerabilities). In contrast, our work achieves orders of magnitude larger coverage of audited domains, analyzes the root causes of such attacks and further explores the use of other defense mechanisms, as well as the privacy leakage users face. Orthogonal to our work are prior studies that proposed defenses against session hijacking attacks [99, 108, 113, 114, 125, 203, 265].

**Cookies and Browsers** Singh et al. [236] built a framework for analyzing the usage of browser features in the wild and detecting browsers' access-control flaws, e.g., *secure* cookies being sent over HTTP. Franken et al. [146] evaluated how different browsers and anti-tracking extensions handle third party requests and showed that cookie-bearing third party requests can be leaked by all browsers, even in the presence of protection mechanisms like *sameSite* cookies. Zheng et al. [279] studied how cookie integrity can be diminished by various adversaries due to specification violations in browser and server-side implementations, and demonstrated practical attacks on popular websites. Cookies are also commonly used for tracking, and Cahn et al. [110] explored their use through empirical large-scale measurements and reported the prevalence of third party cookies. Moreover, Englehardt et al. [140] showed that a passive eavesdropper can exploit third-party cookies to reconstruct up to 74% of a user's browsing history. These studies are orthogonal to our work since we do not examine browser shortcomings in terms of leaking cookies that can lead to session hijacking; instead, we explore the effects of developer malpractices which, however, can be exacerbated by browsers' inability to properly handle cookies.

**Security Headers and Policies** Chen et al. [118] examined the CORS specification, and

browser/server-side implementations, and found security issues in all cases, several previously unknown, which could even lead to data theft and account hijacking. Kranch et al. [173], performed the first in-depth study on HSTS and HPKP, identifying various misconfigurations in preloaded domains as well as Alexa's Top 1M. Mendoza et al. [196] examined HTTP header inconsistencies between websites and their mobile counterparts, and reported cases of mismatches in set cookie flags. Stock et al. [254] presented a longitudinal study on the Web's evolution and, among other things, measured the adoption of security mechanisms. Finally, Roth et al. [228] analyzed websites' security header inconsistencies stemming from different client configurations with varying characteristics, e.g., user agent, vantage point and language settings. While we leverage certain aspects of these studies [173], our goal is not to evaluate these mechanisms in a generic context; instead, we evaluate the deployment of the relevant mechanisms and how they either enable or prevent session hijacking specifically.

**SSO and Sessions** Several studies have focused on SSO-related vulnerabilities. Zhou and Evans [281] implemented SSOScan, a tool that detected vulnerabilities in Facebook's SSO scheme and found that of the 1,660 audited websites, 146 leaked credentials and 202 misused them. While SSOScan handles SSO authentication flows, several issues render it unsuitable for our study; however, we do incorporate one of their heuristics in our framework. Mainly, our system needs to handle non-SSO websites, which account for the vast majority of sites we audit (~92%); this necessitates more advanced and robust form-handling capabilities to address the more complex and diverse nature of non-SSO registration. For instance, SSOScan only uses an input element's `id` and `name` attributes to infer its type, while we leverage all of its attributes, dedicated `label` elements, as well as the input's preceding text as possible labels. Also, since SSOScan processes all input elements of a page at once, there is a chance that it uses an unrelated submit button; we avoid this by processing each form separately. Finally, if SSOScan is not able to locate a conventional submit button it will not be able to submit the form, while our system attempts to do so via Selenium's `submit` method. For SSO workflows, we identified several challenges that SSOScan was not able to handle. For instance, SSOScan's oracle relies on the SSO login button not being displayed after logging in, which, as aforementioned, is not always the case. We address this by separating our SSO and SSO Login oracles. In addition, SSOScan operates only on the homepage for locating candidate elements, while we employ a crawling approach to obtain better coverage. Finally, their tool only considers English sites.

Fett et al. [144] proposed and evaluated a formal model of the OAuth 2.0 protocol. Wang et al. [272] employed differential testing to identify logic flaws in SSO implementations and found several popular IdPs and RPs to be vulnerable. Calzavara et al. [111] implemented a lightweight browser-side monitor for web protocols (e.g., OAuth) that uses formalized protocol specifications to enforce confidentiality and integrity checks. Yang et al. [276] used symbolic execution to audit SSO SDK libraries and discovered seven classes of vulnerabili-

ties in 10 SDKs. Zuo et al. [284] proposed a tool to identify vulnerable authorization implementations in mobile apps, which relied on differential traffic analysis for identifying fields of interest in exchanged messages. They used Facebook’s SSO to audit ~5K apps (306 were vulnerable). They also explored data leakage in mobile apps [283] that use a cloud-based back-end, stemming from key misuse and authorization flaws. However, their leakage exploration focuses on a very limited set of information and they manually setup an account on only 30 apps. Ghasemisharif et al. [147] demonstrated that SSO magnifies the scale and stealthiness of account hijacking, while rendering remediation impossible in most cases. More recently, Ghasemisharif et al. [148] also analyzed flaws and malpractices in SSO deployments, stemming not from the protocols themselves, but from the interplay between the SSO and each RP’s local account and session management. Finally, Jannett et al [163] analyzed the implications of *dual-window* SSO deployments, which, in contrast to the traditional, redirection-based SSO, utilizes in-browser communication channels for the protocol flow, and found that in certain cases it can enable XSS attacks, identity theft and account takeovers. While we use SSO as an alternative way for registering test accounts, identifying flaws in SSO implementations and specifications is not our objective. Nonetheless, these studies shed light on a different problem that can lead to session hijacking.

### 5.3 Third-party Scripts and Attribution

A significant body of work has studied various aspects of 3P scripts and proposed countermeasures. Here we outline the most prominent studies, categorized by their core design.

**Browser instrumentation.** Several systems have been proposed for restricting scripts according to specified security policies [26, 128, 198, 273, 278, 282] and manually-curated task capabilities [190]. Apart from requiring heavy browser modifications, they all require significant manual work by developers, e.g., using new APIs, or have extravagant dependencies (e.g., virtual browsers [116]), limiting their practicality. In contrast, StyxJS is *transparent* to the hosting application and does not require any such interventions.

Prior work has also studied 3P scripts from the angle of user tracking and content blocking. Iqbal et al. [160] designed AdGraph, an ML approach relying on page structure, behavior and requests to detect ads and tracking resources; they also highlighted the need for robust script attribution. The following systems were built on top of PageGraph [239], which captures further page interactions and is more robust than AdGraph. Sjösten et al. [239] trained a classifier to detect ads in a language-agnostic fashion and generate filter lists. Chen et al. [120] relied on JS execution signatures to detect scripts evading filter lists, while Smith et al. [243] trained a classifier to measure the breakage caused by filter lists. Jueckstock et al. [166] also studied website breakage, focusing on 3P storage restrictions. Finally, Sarker et al. [230] relied on concealed API usage to detect obfuscated scripts; their system leveraged a combination of PageGraph, for script provenance, and VisibleV8 [165]

(VV8) for API call detection. The common thread among all these works is that PageGraph (and VV8) is a *passive* monitoring tool. In other words, while extremely useful for offline pre-processing or post-mortem analyses, it cannot be used as-is to provide real time protection mechanisms for end users, as we extensively outlined in §4.4.1. Aside from this fundamental characteristic, PageGraph is only integrated by the Brave browser [91], which has less than 5% market share [77], even when combined with other unpopular browsers. As such, even if PageGraph supported active script interception and tampering, users would need to switch to Brave or build their own instrumented browsers, which is highly unlikely.

**JS instrumentation.** A few systems have been proposed to sandbox JS, either holistically or for 3P scripts specifically. Phung et al. [215] designed a Firefox-specific JS reference monitor relying on prototype patching. Their system could not run in iframes and considered *all* scripts. Agten et al. [101], proposed JSand to sandbox 3P scripts through proxy objects; it is unclear how they handled objects that cannot be wrapped in proxies [165]. Moreover, both systems required developers to write specific security policies. Tran et al. [263] implemented JaTE, a Firefox extension to impose object-level access control among scripts; they also resorted to some straightforward script rewriting, but also utilized proxies and did not handle deployed CSPs to accommodate their rewritten scripts. Moreover, both JaTE and JSand had limited consideration of JS inclusion methods. Terrace et al. [261] sandboxed 3P scripts in a virtual DOM; however, 1P code needed to utilize custom methods to interact with 3P code. Several other works that performed different types of measurements [161, 195, 223, 244, 266] or proposed various countermeasures [159, 232, 235, 245], either relied on naive stack walking to capture 3P calls or did not even disambiguate between 1P and 3P scripts. As such, we believe several of these systems to be excellent candidates for retrofitting on top of StyxJS to enhance their capabilities.

# Chapter 6

## Conclusion

### 6.1 Summary

In this dissertation, we initially demonstrated how the Web's ever-increasing complexity and the inherent challenges that emerge, can severely undermine the effectiveness of existing security and privacy solutions that do not (fully) take them into consideration. As an immediate consequence, Web applications' security and users' privacy are left at stake and exposed to significant risk. On the flipside, we also showed how novel, black-box and context-agnostic techniques have become a necessity in order to effectively tackle several of these challenges, further fortifying the Web's overall security posture and users' privacy.

Specifically, with Web browsers and applications incorporating and supporting complex new features and functionality, vulnerability scanners that operate through raw HTTP requests are facing considerable obstacles that hinder their detection capabilities. Nonetheless, developing an alternative scanner for the modern Web ecosystem that replicates all the features offered by existing scanners would require an exorbitant and infeasible amount of engineering effort. Alternatively, we have opted for a strategy that allows for both *forward* and *backward* compatibility, as our system, ReScan, mediates communication between applications and scanners that already exist or will be developed in the future. Apart from mediating communication with a fully-fledged modern browser, our framework also includes enhancement modules that tackle multiple limitations that affect state-of-the-art scanners. Our experimental evaluation demonstrated how our framework significantly improves the detection of vulnerabilities, as well as the achieved code coverage, in both benchmark and modern Web applications.

Subsequently, we developed a completely automated black-box auditing framework for Web applications that detects authentication and authorization flaws revolving around the handling of cookies and stem from the incorrect, incomplete, or non-existent deployment of appropriate security mechanisms. Our framework is comprised of a series of modules that include novel mechanisms to differentially analyze Web applications, assess the deployment of security mechanisms, and detect what user data is exposed. At the heart of our framework lies a custom browser automation tool designed for robust and fault-tolerant

black-box interaction with Web applications. We used our framework to conduct the largest study on session hijacking to date and audit 25K domains, leading to a series of alarming findings. Despite the increasing adoption of HTTPS, HSTS is rarely deployed (correctly or at all), and ~11K domains are vulnerable to eavesdropping attacks that enable partial or full access to users' accounts. Furthermore, 23% of domains are susceptible to cookie hijacking through JavaScript, the majority of which also include third party scripts that execute in the first party origin. We also demonstrated how hijacked cookies allow access to sensitive and personal user information through various avenues of exposure. Our study reveals that cookie hijacking remains a severe and pressing threat, as adoption of appropriate security mechanisms remains limited and developers continue to struggle with correct deployment. In an effort to shed light on the scale of this threat, guide remediation efforts, and further incentivize the adoption of security mechanisms, we have managed to directly notify ~43% of the affected domains and have also deployed a service for providing reports.

Finally, the ability to correctly and robustly (i.e., even in the face of malicious evasive scripts) differentiate between 1P and 3P scripts is the linchpin of a multitude of security and privacy countermeasures and policy-enforcement mechanisms. However, the ability to deploy these mechanisms and guarantee their effectiveness is undermined when confronted with embedded 3P scripts, as they are executed within the boundaries of the first party's origin. This limitation also impacts web measurement studies, which may underestimate the magnitude and effect of certain threats due to the inability to truly disambiguate 1P and 3P code. To address that gap, we proposed StyxJS, a framework for effectively providing real-time web script attribution. Our system has been tailored to account for a multitude of dynamic script inclusion methods and to tackle JavaScript idiosyncrasies that can hinder correct attribution. At the same time, our design has been guided towards precisely preventing common and custom evasion tactics employed by malicious scripts, while also respecting page-deployed security mechanisms. We conducted an extensive experimental evaluation that highlights the effectiveness and capabilities of our system, and also demonstrated how vastly different security systems can be retrofitted on top of StyxJS. We believe that our system provides functionality that is invaluable for a wide range of security mechanisms and analysis pipelines, and we have open sourced it so that security researchers and practitioners can leverage it.

## 6.2 Future Work

We have outlined the limitations of our work in the corresponding sections and provided some guidelines on how we plan to address them. Here, we summarize the most prominent directions for our future work.

**ReScan FPs and FNs.** ReScan aims to eliminate or at the very least reduce the amount of false positives and negatives generated by existing scanners, regarding specifically XSS vul-



nerabilities. It is apparent that such a feature would certainly be invaluable for other types of vulnerabilities as well. However, achieving this requires the knowledge of specific details for each vulnerability type. For instance, we need to infer and account for *what* type of vulnerability is tested at any given HTTP request and *how* successful exploitation can be verified. As such, we plan to compose such vulnerability-specific modules as part of our future work.

**ReScan's overhead.** We believe the significant benefits and improvements ReScan offers to the underlying scanners, both in terms of vulnerability detection and code coverage, render its unavoidable performance overhead a necessary but acceptable trade-off. Ideally, however, we would like to further reduce the induced overhead and enhance the practicality of our system, e.g., by identifying specific HTTP requests that do *not* require precise workflow execution and proxying them directly to the Web application.

**Automated account creation.** While our approach described in Chapter 3 allowed us to audit orders of magnitude more domains than prior studies, automated account creation remains a significant obstacle. Therefore, we plan to explore additional techniques and heuristics to further improve the overall process' effectiveness. For instance, a new module could identify error messages by the application after a failed form submission, that might reveal or provide a hint on the expected format for a specific input field, further increasing the chances of successfully completing the registration process.

**Further countermeasures.** As extensively outlined in §4.4, StyxJS is capable of accommodating diverse pipelines as custom plugins, while solving their inherent limitations and safeguarding their operation against malicious and evasive scripts. As part of our future work, we plan to further leverage this capability and explore additional countermeasures that can be developed on top of our system, both existing and novel ones. For instance, possible use cases include anti-tracking mechanisms and preventing 3P scripts from reading authentication cookies and other sensitive information.



# Bibliography

- [1] 4.8.2 The iframe element - HTML5, 2010. <https://www.w3.org/TR/2010/WD-html5-20100624/the-iframe-element.html>.
- [2] HTML5 - Developer Guidelines—MDN, 2011. <https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5>.
- [3] spiderman — w3af - Open Source Web Application Security Scanner, 2013. [http://w3af.org/plugins/crawl/spider\\_man](http://w3af.org/plugins/crawl/spider_man).
- [4] GB Gallery Slideshow - WordPress plugin — WordPress.org, 2014. <https://wordpress.org/plugins/gb-gallery-slideshow/>.
- [5] Open Web Application Security Project - The OWASP Top 10 , 2017. <https://www.cloudflare.com/learning/security/threats/owasp-top-10/>.
- [6] Dashlane - world password day: How to improve your passwords, 2018. <https://blog.dashlane.com/world-password-day/>.
- [7] Four cents to deanonymize: Companies reverse hashed email addresses, 2018. <https://freedom-to-tinker.com/2018/04/09/four-cents-to-deanonymize-companies-reverse-hashed-email-addresses/>.
- [8] Vega Tutorial - How to Set Up Vega to Work with Browser, 2018. [https://rkhal101.github.io/\\_posts/WAVS/vega/vega\\_browser\\_setup](https://rkhal101.github.io/_posts/WAVS/vega/vega_browser_setup).
- [9] WIRED - a new Google+ blunder exposed data from 52.5 million users, 2018. <https://www.wired.com/story/google-plus-bug-52-million-users-data-exposed/>.
- [10] WIRED - the Facebook hack exposes an Internet-wide failure, 2018. <https://www.wired.com/story/facebook-hack-single-sign-on-data-exposed/>.
- [11] Ars Technica - DHS: Multiple US gov domains hit in serious DNS hijacking wave, 2019. <https://arstechnica.com/information-technology/2019/01/multiple-us-gov-domains-hit-in-serious-dns-hijacking-wave-dhs-warns/>.
- [12] Cisco Talos - DNS Hijacking Abuses Trust In Core Internet Service, 2019. <https://blog.talosintelligence.com/2019/04/seaturtle.html>.

- [13] Email addresses harvester, 2019. <https://github.com/maldevel/EmailHarvester>.
- [14] Fathom and Fathom 3.7.3 documentation, 2019. <https://mozilla.github.io/fathom/>.
- [15] Google / Harris Poll - Online Security Survey, 2019. [https://services.google.com/fh/files/blogs/google\\_security\\_infographic.pdf](https://services.google.com/fh/files/blogs/google_security_infographic.pdf).
- [16] Opera, Brave, Vivaldi to ignore Chrome's anti-ad-blocker changes, despite shared codebase — ZDNET, 2019. <https://www.zdnet.com/article/opera-brave-vivaldi-to-ignore-chromes-anti-ad-blocker-changes-despite-shared-codebase/>.
- [17] 2020. <https://securitytxt.org/>.
- [18] ChromeDriver - WebDriver for Chrome, 2020. <https://sites.google.com/a/chromium.org/chromedriver/downloads>.
- [19] Cookie Hunter - Notification Service, 2020. <https://cookiehunter.ics.forth.gr/results.php>.
- [20] GeckoDriver, 2020. <https://github.com/mozilla/geckodriver>.
- [21] html5lib - PyPI, 2020. <https://pypi.org/project/html5lib/>.
- [22] Puppeteer, 2020. <https://developers.google.com/web/tools/puppeteer>.
- [23] The Chromium Projects - HTTP Strict Transport Security, 2020. <https://www.chromium.org/hsts>.
- [24] Content Security Policy (CSP) — MDN, 2021. <https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP>.
- [25] estraverse - npm, 2021. <https://www.npmjs.com/package/estraverse>.
- [26] Introduction — Caja — Google for Developers, 2021. <https://developers.google.com/caja/>.
- [27] Service Worker API - Web APIs — MDN, 2021. [https://developer.mozilla.org/en-US/docs/Web/API/Service\\_Worker\\_API](https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API).
- [28] sugarcoat-paper-dataset/resources at master - SugarCoatJS/sugarcoat-paper-dataset - GitHub, 2021. <https://github.com/SugarCoatJS/sugarcoat-paper-dataset/tree/master/resources>.

- [29] sugarcoat-paper-dataset/rules.txt at master - SugarCoatJS/sugarcoat-paper-dataset - GitHub, 2021. <https://github.com/SugarCoatJS/sugarcoat-paper-dataset/blob/master/rules.txt>.
- [30] sugarcoat/mocks at master - SugarCoatJS/sugarcoat - GitHub, 2021. <https://github.com/SugarCoatJS/sugarcoat/tree/master/mocks>.
- [31] XDriver, 2021. <https://gitlab.com/kostasdrk/xdriver3-open/>.
- [32] LeakInspector: an add-on that warns and protects against personal data exfiltration, 2022. <https://github.com/leaky-forms/leak-inspector>.
- [33] Manifest v3 in Firefox: Recap and Next Steps — Mozilla Add-ons Community Blog, 2022. <https://blog.mozilla.org/addons/2022/05/18/manifest-v3-in-firefox-recap-next-steps/>.
- [34] node-forge - npm, 2022. <https://www.npmjs.com/package/node-forge>.
- [35] RFC 9113 - HTTP/2, 2022. <https://datatracker.ietf.org/doc/html/rfc9113>.
- [36] Web - Facebook Login - Documentation - Facebook for Developers, 2022. <https://developers.facebook.com/docs/facebook-login/web>.
- [37] A research-oriented top sites ranking hardened against manipulation - Tranco, 2023. <https://tranco-list.eu/>.
- [38] astring - npm, 2023. <https://www.npmjs.com/package/astring>.
- [39] Content Security Policy (CSP) - HTTP — MDN, 2023. <https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP>.
- [40] Content Security Policy Level 3, 2023. <https://www.w3.org/TR/CSP3/#security-inherit-csp>.
- [41] Content Security Policy Level 3, 2023. <https://w3c.github.io/webappsec-csp/#multiple-policies>.
- [42] CSP: default-src - HTTP — MDN, 2023. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/default-src>.
- [43] CSP: script-src - HTTP — MDN, 2023. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src>.
- [44] CSP: script-src-attr - HTTP — MDN, 2023. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src-attr>.

- [45] CSP: script-src-elem - HTTP — MDN, 2023. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src-elem>.
- [46] Error - JavaScript — MDN, 2023. [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Error](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error).
- [47] Error.prototype.stack - JavaScript — MDN, 2023. [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Error/Stack](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error/Stack).
- [48] Integrating Google Sign-In into your web app — Google Sign-In for Websites — Google Developers, 2023. <https://developers.google.com/identity/sign-in/web/sign-in>.
- [49] jQuery, 2023. <https://jquery.com/>.
- [50] Manifest - Web Accessible Resources — Extensions — Chrome for Developers, 2023. <https://developer.chrome.com/docs/extensions/reference/manifest/web-accessible-resources>.
- [51] Mr. Cooper hackers stole personal data on 14 million customers — TechCrunch, 2023. <https://techcrunch.com/2023/12/18/mr-cooper-hackers-stole-personal-data-on-14-million-customers/?guccounter=1>.
- [52] MutationObserver - Web APIs — MDN, 2023. <https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver>.
- [53] MutationObserver - Web APIs — MDN, 2023. <https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver>.
- [54] nonce - HTML: HyperText Markup Language — MDN, 2023. [https://developer.mozilla.org/en-US/docs/Web/HTML/Global\\_attributes/nonce](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/nonce).
- [55] Overview of the Chrome Extension Manifest V3 - Chrome for Developers, 2023. <https://developer.chrome.com/docs/extensions/mv3/intro/mv3-overview/>.
- [56] PageGraph - brave/brave-browser Wiki - GitHub, 2023. <https://github.com/brave/brave-browser/wiki/PageGraph>.
- [57] privacybadger/src/js/multiDomainFirstParties.js at master - EFForg/privacybadger, 2023. <https://github.com/EFForg/privacybadger/blob/master/src/js/multiDomainFirstParties.js>.
- [58] RequireJS, 2023. <https://requirejs.org/>.

- [59] ReScan-evaluated applications' Docker images, 2023. <https://gitlab.com/kostasdrk/rescanApps>.
- [60] ReScan repository, 2023. <https://gitlab.com/kostasdrk/rescan>.
- [61] Resuming the transition to Manifest V3 — Blog — Chrome for Developers, 2023. <https://developer.chrome.com/blog/resuming-the-transition-to-mv3>.
- [62] Same-origin policy - Web Security — MDN, 2023. [https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin\\_policy](https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy).
- [63] SecurityPolicyViolationEvent - Web APIs — MDN, 2023. <https://developer.mozilla.org/en-US/docs/Web/API/SecurityPolicyViolationEvent>.
- [64] TrustedHTML - Web APIs — MDN, 2023. <https://developer.mozilla.org/en-US/docs/Web/API/TrustedHTML>.
- [65] TrustedScript - Web APIs — MDN, 2023. <https://developer.mozilla.org/en-US/docs/Web/API/TrustedScript>.
- [66] TrustedScriptURL - Web APIs — MDN, 2023. <https://developer.mozilla.org/en-US/docs/Web/API/TrustedScriptURL>.
- [67] web accessible resources - Mozilla — MDN, 2023. [https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/web\\_accessible\\_resources](https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/web_accessible_resources).
- [68] webRequest - Mozilla — MDN, 2023. <https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest>.
- [69] acorn - npm, 2024. <https://www.npmjs.com/package/acorn>.
- [70] AdSense — Google for Developers, 2024. <https://developers.google.com/adsense/>.
- [71] beautifulsoup4 - PyPI, 2024. <https://pypi.org/project/beautifulsoup4/>.
- [72] Blind SQL Injection — OWASP Foundation, 2024. [https://owasp.org/www-community/attacks/Blind\\_SQL\\_Injection](https://owasp.org/www-community/attacks/Blind_SQL_Injection).
- [73] Blocking/Redirect request based on response headers, 2024. <https://issues.chromium.org/issues/40727004>.
- [74] BugMeNot: share logins, 2024. <https://bugmenot.com/>.

- [75] chrome.declarativeNetRequest — API — Chrome for Developers, 2024. <https://developer.chrome.com/docs/extensions/reference/api/declarativeNetRequest>.
- [76] chrome.userScripts — API — Chrome for Developers, 2024. <https://developer.chrome.com/docs/extensions/reference/api/userScripts>.
- [77] Desktop Browser Market Share Worldwide — Statcounter Global Stats, 2024. <https://gs.statcounter.com/browser-market-share/desktop/worldwide>.
- [78] EasyList, 2024. <https://easylist.to/easylist/easylist.txt>.
- [79] EasyPrivacy, 2024. <https://easylist.to/easylist/easyprivacy.txt>.
- [80] Extension using v3 to modify a part of response headers value in declarativeNetRequest, 2024. <https://issues.chromium.org/issues/40794461>.
- [81] [Feature request] manifest v3: blocking responses based on "content-type" header, 2024. <https://issues.chromium.org/issues/40657203>.
- [82] fingerprintjs: Browser fingerprinting library., 2024. <https://github.com/fingerprintjs/fingerprintjs>.
- [83] Get started with Google Analytics — Google Analytics for Firebase, 2024. <https://firebase.google.com/docs/analytics/get-started?platform=web>.
- [84] GitHub - brave/adblock-lists: Maintains adblock lists that Brave uses, 2024. <https://github.com/brave/adblock-lists>.
- [85] GitHub - cure53/DOMPurify: DOMPurify - a DOM-only, super-fast, uber-tolerant XSS sanitizer for HTML, MathML and SVG., 2024. <https://github.com/cure53/DOMPurify>.
- [86] Internet and social media users in the world 2023 — Statista, 2024. <https://www.statista.com/statistics/617136/digital-population-worldwide/>.
- [87] JavaScript libraries market share, websites and contacts - Wappalyzer, 2024. <https://www.wappalyzer.com/technologies/javascript-libraries/>.
- [88] McAfee - customer url ticketing system, 2024. <https://trustedsource.org/en/feedback/url>.
- [89] OWASP. Owasp zed attack proxy (zap), 2024. <https://www.zaproxy.org/>.
- [90] OWASP ZAP - Getting Started, 2024. <https://www.zaproxy.org/getting-started/>.



- [91] Secure, Fast and Private Web Browser with Adblocker — Brave Browser, 2024. <https://brave.com/>.
- [92] StyxJS: Robust and Real-time Third-party Script Attribution, 2024. <https://github.com/kostasDrk/StyxJS>.
- [93] Subresource Integrity - Web security — MDN, 2024. [https://developer.mozilla.org/en-US/docs/Web/Security/Subresource\\_Integrity](https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity).
- [94] The Privacy Sandbox: Technology for a More Private Web., 2024. <https://privacysandbox.com/>.
- [95] Trusted Types API - Web APIs — MDN, 2024. [https://developer.mozilla.org/en-US/docs/Web/API/Trusted.Types\\_API](https://developer.mozilla.org/en-US/docs/Web/API/Trusted.Types_API).
- [96] uAssets/filters at master - uBlockOrigin/uAssets - GitHub, 2024. <https://github.com/uBlockOrigin/uAssets/tree/master/filters>.
- [97] Unrestricted File Upload — OWASP Foundation, 2024. [https://owasp.org/www-community/vulnerabilities/Unrestricted\\_File\\_Upload](https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload).
- [98] Sahar Abdelnabi, Katharina Krombholz, and Mario Fritz. VisualPhishNet: Zero-Day Phishing Website Detection by Visual Similarity. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.
- [99] Ben Adida. Sessionlock: Securing Web Sessions Against Eavesdropping. In *Proceedings of the 17th International Conference on World Wide Web*, 2008.
- [100] Shubham Agarwal. Helping or Hindering? How Browser Extensions Undermine Security. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2022.
- [101] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. JSand: Complete Client-Side Sandboxing of Third-Party JavaScript without Browser Modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012.
- [102] Furkan Alaca and P. C. van Oorschot. Device Fingerprinting for Augmenting Web Authentication: Classification and Analysis of Methods. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, 2016.
- [103] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In *27th USENIX Security Symposium (USENIX Security '18)*, 2018.

- [104] Davide Balzarotti, Marco Cova, Vika Felmetzger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *IEEE Symposium on Security and Privacy*, 2008.
- [105] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *2010 IEEE Symposium on Security and Privacy*, 2010.
- [106] Leyla Bilge, Thorsten Strufe, Davide Balzarotti, and Engin Kirda. All Your Contacts Are Belong To Us: Automated Identity Theft Attacks on Social Networks. In *Proceedings of the 18th international conference on World wide web*, 2009.
- [107] Kevin Bock, Daven Patel, George Hughey, and Dave Levin. unCaptcha: A Low-Resource Defeat of reCaptcha’s Audio Challenge. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.
- [108] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Wilayat Khan. CookiExt: Patching the Browser Against Session Hijacking Attacks. *Journal of Computer Security*, 2015.
- [109] Elie Bursztein, Borbala Benko, Daniel Margolis, Tadek Pietraszek, Andy Archer, Allan Aquino, Andreas Pitsillidis, and Stefan Savage. Handcrafted Fraud and Extortion: Manual Account Hijacking in the Wild. In *IMC ’14 Proceedings of the 2014 Conference on Internet Measurement Conference*, 2014.
- [110] Aaron Cahn, Scott Alfeld, Paul Barford, and S. Muthukrishnan. An Empirical Study of Web Cookies. In *Proceedings of the 25th International Conference on World Wide Web*, 2016.
- [111] Stefano Calzavara, Riccardo Focardi, Matteo Maffei, Clara Schneidewind, Marco Squarcina, and Mauro Tempesta. WPSE: Fortifying Web Protocols via Browser-Side Security Monitoring. In *27th USENIX Security Symposium (USENIX Security)*. USENIX Association, 2018.
- [112] Stefano Calzavara, Riccardo Focardi, Matúš Nemeč, Alvisė Rabitti, and Marco Squarcina. Postcards from the Post-HTTP World: Amplification of HTTPS Vulnerabilities in the Web Ecosystem. In *2019 IEEE Symposium on Security and Privacy*, 2019.
- [113] Stefano Calzavara, Riccardo Focardi, Marco Squarcina, and Mauro Tempesta. Surviving the Web: A Journey into Web Session Security. *ACM Computing Surveys*, 2017.
- [114] Stefano Calzavara, Alvisė Rabitti, and Michele Bugliesi. Sub-session hijacking on the web: Root causes and prevention. In *Journal of Computer Security*, 2018.

- [115] Stefano Calzavara, Alvisio Rabitti, Alessio Ragazzo, and Michele Bugliesi. Testing for Integrity Flaws in Web Sessions. In *Computer Security - th European Symposium on Research in Computer Security, ESORICS*, 2019.
- [116] Yinzhi Cao, Zhichun Li, Vaibhav Rastogi, Yan Chen, and Xitao Wen. Virtual Browser: A Virtualized Browser to Sandbox Third-Party JavaScripts with Enhanced Security. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, 2012.
- [117] Bertil Chapuis, Olamide Omolola, Mauro Cherubini, Mathias Humbert, and Kévin Huguenin. An Empirical Study of the Use of Integrity Verification Mechanisms for Web Subresources. In *Proceedings of The Web Conference*, 2020.
- [118] Jianjun Chen, Jian Jiang, Haixin Duan, Tao Wan, Shuo Chen, Vern Paxson, and Min Yang. We Still Don't Have Secure Cross-Domain Requests: an Empirical Study of CORS. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.
- [119] Quan Chen, Panagiotis Ilia, Michalis Polychronakis, and Alexandros Kapravelos. Cookie Swap Party: Abusing First-Party Cookies for Web Tracking. In *Proceedings of the Web Conference*, 2021.
- [120] Quan Chen, Peter Snyder, Ben Livshits, and Alexandros Kapravelos. Detecting Filter List Evasion with Event-Loop-Turn Granularity JavaScript Signatures. In *IEEE Symposium on Security and Privacy*, 2021.
- [121] Chen, Yuanchao and Li, Yuwei and Pan, Zulie and Lu, Yuliang and Chen, Juxing and Ji, Shouling. URadar: Discovering Unrestricted File Upload Vulnerabilities via Adaptive Dynamic Testing. *IEEE Transactions on Information Forensics and Security*, 19, 2024.
- [122] Sandy Clark, Stefan Frei, Matt Blaze, and Jonathan Smith. Familiarity breeds contempt: The honeymoon effect and the role of legacy code in zero-day vulnerabilities. In *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010.
- [123] Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. mitmproxy: A free and open source interactive HTTPS proxy, 2024. <https://mitmproxy.org/>.
- [124] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. ZOZ-ZLE: Fast and Precise In-Browser JavaScript Malware Detection. In *20th USENIX Security Symposium (USENIX Security 11)*, 2011.
- [125] Italo Dacosta, Saurabh Chakradeo, Mustaque Ahamad, and Patrick Traynor. One-time Cookies: Preventing Session Hijacking Attacks with Stateless Authentication Tokens. *ACM Trans. Internet Technol.*, 2012.

- [126] Firat Coskun Dalgic, Ahmet Selman Bozkir, and Murat Aydos. Phish-IRIS: A New Approach for Vision Based Brand Prediction of Phishing Web Pages via Compact Visual Descriptors. *2018 2nd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, 2018.
- [127] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications. In *Proceedings of the 18th Conference on USENIX Security Symposium*, 2009.
- [128] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. FlowFox: A Web Browser with Flexible and Precise Information Flow Control. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2012.
- [129] Joe DeBlasio, Stefan Savage, Geoffrey M Voelker, and Alex C Snoeren. Tripwire: Inferring Internet Site Compromise. In *Proceedings of the Internet Measurement Conference*, 2017.
- [130] Levent Demir, Amrit Kumar, Mathieu Cunche, and Cedric Lauradoux. The Pitfalls of Hashing for Privacy. *IEEE Communications Surveys & Tutorials*, 2017.
- [131] Lieven Desmet, Frank Piessens, Wouter Joosen, and Pierre Verbaeten. Bridging the Gap Between Web Application Firewalls and Web Applications. In *Proceedings of the fourth ACM workshop on Formal methods in security*, 2006.
- [132] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. In *21st USENIX Security Symposium*, 2012.
- [133] Adam Doupé, Marco Cova, and Giovanni Vigna. Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners. In *"Detection of Intrusions and Malware, and Vulnerability Assessment"*, 2010.
- [134] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. The Cookie Hunter: Automated Black-Box Auditing for Web Authentication and Authorization Flaws. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2020.
- [135] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. ReScan: A Middleware Framework for Realistic and Robust Black-box Web Application Scanning. In *30th Annual Network and Distributed System Security Symposium, NDSS*, 2023.
- [136] Fabien Duchéne, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. LigRE: Reverse-engineering of control and data flow models for black-box XSS detection. In *20th Working Conference on Reverse Engineering*, 2013.

- [137] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. KameleonFuzz: Evolutionary Fuzzing for Black-Box XSS Detection. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, 2014.
- [138] Siham el Idrissi, Naoual Berbiche, Fatima Guerouate, and Sbihi Mohamed. Performance Evaluation of Web Application Security Scanners for Prevention and Protection Against Vulnerabilities. *International Journal of Applied Engineering Research*, 2017.
- [139] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of ACM CCS 2016*, 2016.
- [140] Steven Englehardt, Dillon Reisman, Christian Eubank, Peter Zimmerman, Jonathan Mayer, Arvind Narayanan, and Edward W. Felten. Cookies That Give You Away: The Surveillance Implications of Web Tracking. In *Proceedings of the 24th International Conference on World Wide Web*, 2015.
- [141] B. Eriksson, G. Pellegrino, and A. Sabelfeld. Black Widow: Blackbox Data-driven Web Scanning. In *2020 IEEE Symposium on Security and Privacy*, 2021.
- [142] Aurore Fass, Dolière Francis Somé, Michael Backes, and Ben Stock. DoubleX: Statically Detecting Vulnerable Data Flows in Browser Extensions at Scale. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2021.
- [143] Viktoria Felmetzger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *Proceedings of the 19th USENIX Conference on Security*, 2010.
- [144] Daniel Fett, Ralf Küsters, and Guido Schmitz. A Comprehensive Formal Security Analysis of OAuth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.
- [145] Roy Fielding and Julian Reschke. RFC 7231 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, 2014. <https://tools.ietf.org/html/rfc7231#section-4.2.1>.
- [146] Gertjan Franken, Tom Van Goethem, and Wouter Joosen. Who Left Open the Cookie Jar? A Comprehensive Evaluation of Third-Party Cookie Policies. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.
- [147] Mohammad Ghasemisharif, Amrutha Ramesh, Stephen Checkoway, Chris Kanich, and Jason Polakis. O Single Sign-Off, Where Art Thou? An Empirical Analysis of Single Sign-On Account Hijacking and Session Management on the Web. In *27th USENIX Security Symposium (USENIX Security)*, 2018.

- [148] Ghasemisharif, Mohammad and Kanich, Chris and Polakis, Jason. Towards Automated Auditing for Account and Session Management Flaws in Single Sign-on Deployments. In *IEEE Symposium on Security and Privacy (SP)*, 2022.
- [149] Shashank Gupta and Brij Bhooshan Gupta. Cross-Site Scripting (XSS) attacks and defense mechanisms: Classification and State-of-the-art. *International Journal of System Assurance Engineering and Management*, 2017.
- [150] Emre Güler, Sergej Schumilo, Moritz Schloegel, Nils Bars, Philipp Görz, Xinyi Xu, Cemal Kaygusuz, and Thorsten Holz. Atropos: Effective Fuzzing of Web Applications for Server-Side Vulnerabilities. In *33rd USENIX Security Symposium*, 2024.
- [151] B. Krumnow H. Jonker, S. Karsch and M. Slegers. Shepherd: A Generic Approach to Automating Website Login. In *Proceedings of the 2020 Workshop on Measurements, Attacks, and Defenses for the Web*, 2020.
- [152] William G.J. Halfond, Shauvik Roy Choudhary, and Alessandro Orso. Penetration Testing with Improved Input Vector Identification. In *International Conference on Software Testing Verification and Validation*, 2009.
- [153] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, VN Venkatakrisnan, Runqing Yang, and Zhenrui Zhang. Vetting SSL usage in applications with SSLint. In *2015 IEEE Symposium on Security and Privacy*, 2015.
- [154] Mario Heiderich, Christopher Späth, and Jörg Schwenk. DOMPurify: Client-Side Protection Against XSS and Markup Injection. In *Proceedings of the 22nd European Symposium on Research in Computer Security*, 2017.
- [155] Jin Huang, Junjie Zhang, Jialun Liu, Chuang Li, and Rui Dai. UFuzzer: Lightweight Detection of PHP-Based Unrestricted File Upload Vulnerabilities Via Static-Fuzzing Co-Analysis. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*, 2021.
- [156] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the 13th International Conference on World Wide Web*, 2004.
- [157] Markus Huber, Martin Mulazzani, Edgar Weippl, Gerhard Kitzler, and Sigrun Goluch. Exploiting Social Networking Sites for Spam. In *Proceedings of the 17th ACM conference on Computer and communications security*, 2010.
- [158] Muhammad Ikram, Rahat Masood, Gareth Tyson, Mohamed Ali Kaafar, Noha Loizon, and Roya Ensafi. The Chain of Implicit Trust: An Analysis of the Web Third-Party Resources Loading. In *The World Wide Web Conference*, 2019.

- [159] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors. In *IEEE Symposium on Security and Privacy*, 2021.
- [160] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. AdGraph: A Graph-Based Approach to Ad and Tracker Blocking. In *IEEE Symposium on Security and Privacy*, 2020.
- [161] Umar Iqbal, Charlie Wolfe, Charles Nguyen, Steven Englehardt, and Zubair Shafiq. Khaleesi: Breaker of Advertising and Tracking Request Chains. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.
- [162] Collin Jackson and Adam Barth. ForceHTTPS: Protecting high-security web sites from network attacks. In *Proceedings of the 17th International World Wide Web Conference*, 2008.
- [163] Louis Jannett, Vladislav Mladenov, Christian Mainka, and Jörg Schwenk. DISTINCT: Identity Theft Using In-Browser Communications in Dual-Window Single Sign-On. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2022.
- [164] Hugo Jonker, Benjamin Krumnow, and Gabry Vlot. Fingerprint Surface-Based Detection of Web Bot Detectors. In *European Symposium on Research in Computer Security*, 2019.
- [165] Jordan Jueckstock and Alexandros Kapravelos. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2019.
- [166] Jordan Jueckstock, Peter Snyder, Shaown Sarker, Alexandros Kapravelos, and Benjamin Livshits. Measuring the Privacy vs. Compatibility Trade-off in Preventing Third-Party Stateful Tracking. In *Proceedings of The Web Conference*, 2022.
- [167] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. SecuBat: A Web Vulnerability Scanner. In *Proceedings of the 15th International Conference on World Wide Web*, 2006.
- [168] Soroush Karami, Panagiotis Ilia, and Jason Polakis. Awakening the Web's Sleeper Agents: Misusing Service Workers for Privacy Leakage. In *NDSS*, 2021.
- [169] Soroush Karami, Panagiotis Ilia, Konstantinos Solomos, and Jason Polakis. Carnus: Exploring the Privacy Threats of Browser Extension Fingerprinting. In *NDSS*, 2020.

- [170] Soroush Karami, Faezeh Kalantari, Mehrnoosh Zaeifi, Xavier J. Maso, Erik Trickel, Panagiotis Ilia, Yan Shoshitaishvili, Adam Doupé, and Jason Polakis. Unleash the Simulacrum: Shifting Browser Realities for Robust Extension-Fingerprinting Prevention. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.
- [171] Soheil Khodayari and Giancarlo Pellegrino. JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals. In *30th USENIX Security Symposium*, 2021.
- [172] Platon Kotzias, Abbas Razaghpanah, Johanna Amann, Kenneth G Paterson, Narseo Vallina-Rodriguez, and Juan Caballero. Coming of age: A longitudinal study of TLS deployment. In *Proceedings of the Internet Measurement Conference 2018*, 2018.
- [173] Michael Kranch and Joseph Bonneau. Upgrading HTTPS in mid-air: An empirical study of strict transport security and key pinning. In *22nd Annual Network and Distributed System Security Symposium, NDSS*, 2015.
- [174] Katharina Krombholz, Wilfried Mayer, Martin Schmiedecker, and Edgar Weippl. "I Have No Idea What I'm Doing"-On the Usability of Deploying HTTPS. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.
- [175] Tammo Krueger, Christian Gehl, Konrad Rieck, and Pavel Laskov. TokDoc: A Self-healing Web Application Firewall. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010.
- [176] Deepak Kumar, Zane Ma, Zakir Durumeric, Ariana Mirian, Joshua Mason, J Alex Halderman, and Michael Bailey. Security Challenges in an Increasingly Tangled Web. In *Proceedings of the 26th International Conference on World Wide Web*, 2017.
- [177] H. Kwon, H. Nam, S. Lee, C. Hahn, and J. Hur. (In-)Security of Cookies in HTTPS: Cookie Theft by Removing Cookie Flags. *IEEE Transactions on Information Forensics and Security*, 2019.
- [178] Pierre Laperdrix, Oleksii Starov, Quan Chen, Alexandros Kapravelos, and Nick Nikiforakis. Fingerprinting in Style: Detecting Browser Extensions via Injected Style Sheets. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
- [179] Tobias Lauinger, Chaabane Abdelberi, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou Shalt Not Depend on Me: Analysing the Use of Out-dated JavaScript Libraries on the Web. In *Proceedings of the Network and Distributed System Security Symposium*, 2017.



- [180] Jehyun Lee, Pingxiao Ye, Ruofan Liu, Dinil Mon Divakaran, and Mun Chan. Building Robust Phishing Detection System: An Empirical Analysis. In *Workshop on Measurements, Attacks, and Defenses for the Web*, 2020.
- [181] Taek-Jin Lee, SeongIl Wi, S. Lee, and Sooel Son. FUSE: Finding File Upload Bugs via Penetration Testing. In *NDSS*, 2020.
- [182] Sebastian Lekies, Krzysztof Kotowicz, Samuel Groß, Eduardo A. Vela Nava, and Martin Johns. Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2017.
- [183] Sebastian Lekies, Ben Stock, and Martin Johns. 25 Million Flows Later: Large-Scale Detection of DOM-Based XSS. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2013.
- [184] Frank Li, Zakir Durumeric, Jakub Czyz, Mohammad Karami, Michael Bailey, Damon McCoy, Stefan Savage, and Vern Paxson. You’ve Got Vulnerability: Exploring Effective Vulnerability Notifications. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.
- [185] Xiaowei Li, Wei Yan, and Yuan Xue. SENTINEL: Securing Database from Logic Flaws in Web Applications. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, 2012.
- [186] Yukun Li, Zhenguo Yang, Xu Chen, Huaping Yuan, and Wenyin Liu. A Stacking Model Using URL and HTML Features for Phishing Webpage Detection. *Future Generation Computer Systems*, 2019.
- [187] Xu Lin, Panagiotis Ilia, and Jason Polakis. Fill in the Blanks: Empirical Analysis of the Privacy Threats of Browser Form Autofill. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2020.
- [188] Yun Lin, Ruofan Liu, Dinil Mon Divakaran, Jun Yang Ng, Qing Zhou Chan, Yiwen Lu, Yuxuan Si, Fan Zhang, and Jin Song Dong. Phishpedia: A Hybrid Deep Learning Based Approach to Visually Identify Phishing Webpages. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
- [189] Ruofan Liu, Yun Lin, Xianglin Yang, Siang Hwee Ng, Dinil Mon Divakaran, and Jin Song Dong. Inferring Phishing Intention via Webpage Appearance and Dynamics: A Deep Vision Based Approach. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

- [190] Wu Luo, Xuhua Ding, Pengfei Wu, Xiaolei Zhang, Qingni Shen, and Zhonghai Wu. ScriptChecker: To tame third-party script execution with task capabilities. In *Proceedings of the Network and Distributed System Security Symposium*, 2022.
- [191] Francesco Marcantoni, Michalis Diamantaris, Sotiris Ioannidis, and Jason Polakis. A large-scale study on the risks of the html5 webapi for mobile sensor-based attacks. In *The World Wide Web Conference*, 2019.
- [192] Moxie Marlinspike. New Tricks For Defeating SSL In Practice. *BlackHat DC*, 2009.
- [193] Matthias Marx, Ephraim Zimmer, Tobias Mueller, Maximilian Blochberger, and Hannes Federrath. Hashing of personally identifiable information is not sufficient. *SICHERHEIT*, 2018.
- [194] Arunesh Mathur, Nathan Malkin, Marian Harbach, Eyal Peer, and Serge Egelman. Quantifying Users’ Beliefs about Software Updates. *CoRR*, 2018.
- [195] Jonathan R. Mayer and John C. Mitchell. Third-Party Web Tracking: Policy and Technology. In *IEEE Symposium on Security and Privacy*, 2012.
- [196] Abner Mendoza, Phakpoom Chinprutthiwong, and Guofei Gu. Uncovering HTTP Header Inconsistencies and the Impact on Desktop/Mobile Websites. In *Proceedings of the 2018 World Wide Web Conference*, 2018.
- [197] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling AJAX by Inferring User Interface State Changes. In *Eighth International Conference on Web Engineering*, 2008.
- [198] Leo A. Meyerovich and Benjamin Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *IEEE Symposium on Security and Privacy*, 2010.
- [199] Michal Zalewski. *The Tangled Web*. No Starch Press, 2011.
- [200] Yogesh Mundada, Nick Feamster, and Balachander Krishnamurthy. Half-Baked Cookies: Hardening Cookie-Based Authentication for the Modern Web. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS ’16. ACM, 2016.
- [201] Marius Musch, Marius Steffens, Sebastian Roth, Ben Stock, and Martin Johns. ScriptProtect: Mitigating Unsafe Third-Party JavaScript Practices. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, 2019.
- [202] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You Are What You

- Include: Large-Scale Evaluation of Remote Javascript Inclusions. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2012.
- [203] Nick Nikiforakis, Wannes Meert, Yves Younan, Martin Johns, and Wouter Joosen. SessionShield: Lightweight Protection against Session Hijacking. In *Engineering Secure Software and Systems*, 2011.
- [204] Hrvoje Nikšić. Wget - gnu project - free software foundation, 2020. <https://www.gnu.org/software/wget/>.
- [205] Jeremiah Onaolapo, Enrico Mariconti, and Gianluca Stringhini. What happens after you are pwned: Understanding the use of leaked webmail credentials in the wild. In *Proceedings of the Internet Measurement Conference*, 2016.
- [206] Panagiotis Papadopoulos, Panagiotis Iliia, Michalis Polychronakis, Evangelos Markatos, Sotiris Ioannidis, and Giorgos Vasiliadis. Master of Web Puppets: Abusing Web Browsers for Persistent and Stealthy Computation. In *NDSS*, 2019.
- [207] Sunnyeo Park, Daejun Kim, Suman Jana, and Sooel Son. FUGIO: Automatic Exploit Generation for PHP Object Injection Vulnerabilities. In *31st USENIX Security Symposium*, 2022.
- [208] Muhammad Parvez, Pavol Zavarisky, and Nidal Khoury. Analysis of Effectiveness of Black-box Web Application Scanners in Detection of Stored SQL Injection and Stored XSS Vulnerabilities. In *10th International Conference for Internet Technology and Secured Transactions*, 2015.
- [209] Avanish Pathak. An Analysis of Various Tools, Methods and Systems to Generate Fake Accounts for Social Media. *Northeastern University Boston, Massachusetts*, 2014.
- [210] Mateusz Pawlik and Nikolaus Augsten. Efficient Computation of the Tree Edit Distance. *ACM Trans. Database Syst.*, 40, 2015.
- [211] Mateusz Pawlik and Nikolaus Augsten. Tree Edit Distance: Robust and Memory-efficient. *Information Systems*, 56, 2016.
- [212] Sarah Pearman, Jeremy Thomas, Pardis Emami Naeini, Hana Habib, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Serge Egelman, and Alain Forget. Let's Go in for a Closer Look: Observing Passwords in Their Natural Habitat. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2017.
- [213] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications. In *Research in Attacks, Intrusions, and Defenses*, 2015.

- [214] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.
- [215] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight Self-Protecting JavaScript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, 2009.
- [216] Pablo Picazo-Sanchez, Juan Tapiador, and Gerardo Schneider. After you, please: Browser Extensions Order Attacks and Countermeasures. *International Journal of Information Security*, 2020.
- [217] Jason Polakis, Marco Lancini, Georgios Kontaxis, Federico Maggi, Sotiris Ioannidis, Angelos D. Keromytis, and Stefano Zanero. All your face are belong to us: breaking Facebook’s social authentication. In *Annual Computer Security Applications Conference (ACSAC)*, 2012.
- [218] Jeremy Rack and Cristian-Alexandru Staicu. Jack-in-the-box: An Empirical Study of JavaScript Bundling on the Web and its Security Implications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2023.
- [219] N. Ramasubbu, M. Cataldo, R. K. Balan, and J. D. Herbsleb. Configuring Global Software Teams: A Multi-company Analysis of Project Productivity, Quality, and Profits. In *33rd International Conference on Software Engineering (ICSE)*, 2011.
- [220] Rautenstrauch, Jannis and Mitkov, Metodi and Helbrecht, Thomas and Hetterich, Lorenz and Stock, Ben. To Auth or Not To Auth? A Comparative Analysis of the Pre- and Post-Login Security Landscape. In *IEEE Symposium on Security and Privacy (SP)*, 2024.
- [221] Derick Rethans. Xdebug - Debugger and Profiler Tool for PHP, 2023. <https://xdebug.org/>.
- [222] Andres Riancho. w3af - open source web application security scanner, 2013. <http://w3af.org/>.
- [223] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and Defending Against Third-Party Tracking on the Web. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012.
- [224] Orpheas van Rooij, Marcos Antonios Charalambous, Demetris Kaizer, Michalis Papaevripides, and Elias Athanasopoulos. webFuzz: Grey-Box Fuzzing for Web Applications. In *European Symposium on Research in Computer Security*, 2021.

- [225] Suood Al Roomi and Frank Li. A Large-Scale Measurement of Website Login Policies. In *32nd USENIX Security Symposium (USENIX Security)*, 2023.
- [226] 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*, 2020.
- [227] 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*, 2021.
- [228] Roth, Sebastian and Calzavara, Stefano and Wilhelm, Moritz and Rabitti, Alvise and Stock, Ben. The Security Lottery: Measuring {Client-Side} Web Security Inconsistencies. In *31st USENIX Security Symposium (USENIX Security)*, 2022.
- [229] Iskander Sanchez-Rola, Matteo Dell'Amico, Davide Balzarotti, Pierre-Antoine Vervier, and Leyla Bilge. Journey to the Center of the Cookie Ecosystem: Unraveling Actors' Roles and Relationships. In *2021 IEEE Symposium on Security and Privacy*, 2021.
- [230] Shaown Sarker, Jordan Jueckstock, and Alexandros Kapravelos. Hiding in Plain Site: Detecting JavaScript Obfuscation through Concealed Browser API Usage. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2020.
- [231] Quirin Scheitle, Oliver Hohlfeld and Julien Gamba, Jonas Jelten, Torsten Zimmermann, Stephen D. Strowes, and Narseo Vallina-Rodriguez. A Long Way to the Top: Significance, Structure, and Stability of Internet Top Lists. In *IMC*, 2018.
- [232] Michael Schwarz, Moritz Lipp, and Daniel Gruss. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In *Proceedings of the Network and Distributed System Security Symposium*, 2018.
- [233] Asuman Senol, Gunes Acar, Mathias Humbert, and Frederik Zuiderveen Borgesius. Leaky Forms: a study of email and password exfiltration before form submission. In *31st USENIX Security Symposium (USENIX Security)*, 2022.
- [234] Mikhail Shcherbakov and Musard Balliu. SerialDetector: Principled and Practical Exploration of Object Injection Vulnerabilities for the Web. In *28th Annual Network and Distributed System Security Symposium, NDSS*, 2021.
- [235] Sandra Siby, Umar Iqbal, Steven Englehardt, Zubair Shafiq, and Carmela Troncoso. WebGraph: Capturing Advertising and Tracking Information Flows for Robust Blocking. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

- [236] Kapil Singh, Alexander Moshchuk, Helen J Wang, and Wenke Lee. On the Incoherencies in Web Browser Access Control Policies. In *IEEE Symposium on Security and Privacy*, 2010.
- [237] Suphannee Sivakorn, Angelos D. Keromytis, and Jason Polakis. That’s the Way the Cookie Crumbles: Evaluating HTTPS Enforcing Mechanisms. In *Proceedings of the ACM on Workshop on Privacy in the Electronic Society*, 2016.
- [238] Suphannee Sivakorn, Jason Polakis, and Angelos D. Keromytis. The Cracked Cookie Jar: HTTP Cookie Hijacking and the Exposure of Private Information. In *In Proceedings of the 37th IEEE Symposium on Security and Privacy*, 2016.
- [239] Alexander Sjösten, Peter Snyder, Antonio Pastor, Panagiotis Papadopoulos, and Benjamin Livshits. Filter List Generation for Underserved Regions. In *Proceedings of The Web Conference*, 2020.
- [240] Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld. Discovering Browser Extensions via Web Accessible Resources. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017.
- [241] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. Anything to Hide? Studying Minified and Obfuscated Code in the Web. In *The World Wide Web Conference*, 2019.
- [242] 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 SIGSAC Conference on Computer and Communications Security*, 2021.
- [243] Michael Smith, Peter Snyder, Moritz Haller, Benjamin Livshits, Deian Stefan, and Hamed Haddadi. Blocked or Broken? Automatically Detecting When Privacy Interventions Break Websites. *Proceedings on Privacy Enhancing Technologies*, 2022.
- [244] Peter Snyder, Lara Ansari, Cynthia Taylor, and Chris Kanich. Browser Feature Usage on the Modern Web. In *Proceedings of the 2016 Internet Measurement Conference*, 2016.
- [245] Peter Snyder, Cynthia Taylor, and Chris Kanich. Most Websites Don’t Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2017.
- [246] Johnny So, Michael Ferdman, and Nick Nikiforakis. The More Things Change, the More They Stay the Same: Integrity of Modern JavaScript. In *Proceedings of The Web Conference*, 2023.

- [247] Saumya Solanki, Gautam Krishnan, Varshini Sampath, and Jason Polakis. In (Cyber)Space Bots Can Hear You Speak: Breaking Audio CAPTCHAs Using OTS Speech Recognition. In *Proceedings 10th ACM Workshop on Artificial Intelligence and Security, AISEC '17*, 2017.
- [248] Konstantinos Solomos, Panagiotis Ilia, Soroush Karami, Nick Nikiforakis, and Jason Polakis. The Dangers of Human Touch: Fingerprinting Browser Extensions through User Actions. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.
- [249] Konstantinos Solomos, Panagiotis Ilia, Nick Nikiforakis, and Jason Polakis. Escaping the Confines of Time: Continuous Browser Extension Fingerprinting Through Ephemeral Modifications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2022.
- [250] Sooel Son, Kathryn S. Mckinley, and Vitaly Shmatikov. Fix Me Up: Repairing access-control bugs in web applications. In *In Network and Distributed System Security Symposium (NDSS)*, 2013.
- [251] Stefano Calzavara and Hugo Jonker and Benjamin Krumnow and Alvis Rabitti. Measuring Web Session Security at Scale. *Computers and Security*, 111, 2021.
- [252] 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*, 2021.
- [253] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. In *NDSS*, 2019.
- [254] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.
- [255] Ben Stock, Giancarlo Pellegrino, Christian Rossow, Martin Johns, and Michael Backes. Hey, you have a problem: On the feasibility of large-scale web vulnerability notification. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.
- [256] Ben Stock, Stephan Pfister, Bernd Kaiser, Sebastian Lekies, and Martin Johns. From facepalm to brain bender: Exploring client-side cross-site scripting. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, 2015.
- [257] Junhua Su and Alexandros Kapravelos. Automatic Discovery of Emerging Browser Fingerprinting Techniques. In *Proceedings of The Web Conference*, 2023.

- [258] Chris Sullo. Nikto2 — CIRT.net, 2024. <https://cirt.net/Nikto2>.
- [259] N. Surribas. Wapiti : a Free and Open-Source web-application vulnerability scanner, 2023. <https://wapiti.sourceforge.io/>.
- [260] Larry Suto and Consultant San. Analyzing the accuracy and time costs of web application security scanners. 2010.
- [261] Jeff Terrace, Stephen R. Beard, and Naga Praveen Kumar Katta. JavaScript in JavaScript (js.js): Sandboxing Third-Party Scripts. In *3rd USENIX Conference on Web Application Development*, 2012.
- [262] Kurt Thomas, Dmytro Iatskiv, Elie Bursztein, Tadek Pietraszek, Chris Grier, and Damon McCoy. Dialing Back Abuse on Phone Verified Accounts. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.
- [263] Tung Tran, Riccardo Pelizzi, and R. C. Sekar. JaTE: Transparent and Efficient JavaScript Confinement. In *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015.
- [264] Trickel, Erik and Pagani, Fabio and Zhu, Chang and Dresel, Lukas and Vigna, Giovanni and Kruegel, Christopher and Wang, Ruoyu and Bao, Tiffany and Shoshitaishvili, Yan and Doupé, Adam. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities. In *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.
- [265] T. Unger, M. Mulazzani, D. Frühwirt, M. Huber, S. Schrittwieser, and E. Weippl. SHPF: Enhancing HTTP(S) Session Security with Browser Fingerprinting. In *International Conference on Availability, Reliability and Security*, 2013.
- [266] Tobias Urban, Martin Degeling, Thorsten Holz, and Norbert Pohlmann. Beyond the Front Page: Measuring Third Party Dynamics in the Field. In *Proceedings of The Web Conference*, 2020.
- [267] Kami Vaniea and Yasmeen Rashidi. Tales of Software Updates: The Process of Updating Software. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 2016.
- [268] Rakesh Verma and Keith Dyer. On the Character of Phishing URLs: Accurate and Robust Statistical Learning Classifiers. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, 2015.
- [269] Alexandre Vernotte, Franck Lebeau, Frédéric Dadeau, Bruno Legeard, Fabien Peureux, and Francois Piat. Efficient Detection of Multi-step Cross-Site Scripting Vulnerabilities. In *10th International Conference on Information Systems Security*, 2014.



- [270] Marco Vieira, Nuno Antunes, and Henrique Madeira. Using Web Security Scanners to Detect Vulnerabilities in Web Services. In *IEEE/IFIP International Conference on Dependable Systems Networks*, 2009.
- [271] Thomas Vissers, Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. Maneuvering Around Clouds: Bypassing Cloud-Based Security Providers. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.
- [272] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing Me Onto Your Accounts Through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In *2012 IEEE Symposium on Security and Privacy*, 2012.
- [273] Zilun Wang, Wei Meng, and Michael R. Lyu. Fine-Grained Data-Centric Content Protection Policy for Web Applications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2023.
- [274] Rick Wash, Emilee Rader, Kami Vaniea, and Michelle Rizor. Out of the Loop: How Automated Software Updates Cause Unintended Security Consequences. In *10th Symposium On Usable Privacy and Security (SOUPS 2014)*, 2014.
- [275] 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*, 2016.
- [276] Ronghai Yang, Wing Cheong Lau, Jiongyi Chen, and Kehuan Zhang. Vetting Single Sign-On SDK Implementations via Symbolic Reasoning. In *27th USENIX Security Symposium (USENIX Security)*, 2018.
- [277] Kaizhong Zhang and Dennis Shasha. Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM J. Comput.*, 18, 1989.
- [278] Mingxue Zhang and Wei Meng. JSISOLATE: Lightweight In-browser JavaScript Isolation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.
- [279] Xiaofeng Zheng, Jian Jiang, Jinjin Liang, Haixin Duan, Shuo Chen, Tao Wan, and Nicholas Weaver. Cookies Lack Integrity: Real-World Implications. In *24th USENIX Security Symposium (USENIX Security)*, 2015.
- [280] Yuchen Zhou and David Evans. Why aren't HTTP-only cookies more widely deployed. *Proceedings of 4th Web 2.0 Security and Privacy*, 2010.

- 
- [281] Yuchen Zhou and David Evans. SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities. In *23rd USENIX Security Symposium (USENIX Security)*, 2014.
- [282] Yuchen Zhou and David Evans. Understanding and Monitoring Embedded Web Scripts. In *IEEE Symposium on Security and Privacy*, 2015.
- [283] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. Why Does Your Data Leak? Uncovering the Data Leakage in Cloud From Mobile Apps. In *2019 IEEE Symposium on Security and Privacy*, 2019.
- [284] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. AUTHSCOPE: Towards Automatic Discovery of Vulnerable Authorizations in Online Services. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

# Appendix A

## A.1 Static file extensions

Requests towards the following file extensions are directly proxied to the web application:

js, json, css, crt, mp3, wav, wma, ogg, mkv, zip, gz, tar, xz, rar, z, deb, iso, csv, tsv, dat, txt, log, sql, xml, mdb, apk, bat, bin, exe, jar, wsf, fnt, fon, otf, ttf, ai, bmp, gif, ico, jpeg, png, ps, psd, svg, tif, tiff, cer, rss, key, odp, pps, ppt, pptx, c, class, cpp, cs, h, java, sh, swift, vb, odf, xlr, xls, xlsx, bak, cab, cfg, cpl, cur, dll, dmp, drv, icns, ini, lnk, msi, sys, tmp, 3g2, 3gp, avi, flv, h264, m4v, mov, mp4, mpeg, rm, swf, vob, wmv, doc, docx, odt, pdf, rtf, tex, wks, wps, wpd, woff, eot, xap.

## A.2 ReScan's API

ReScan's API endpoints, which we detail next, can be split in two categories: *passive*, which aim to provide valuable insights and information sharing between ReScan and the scanner, and *active*, which alter ReScan's behavior at runtime, according to the scanner's needs. For instance, if the scanner tests for stored XSS it can enable the ISD module, but disable it later when testing for a vulnerability that does not affect other pages, e.g., open redirects. Similarly, it might disable the authentication helper when it is not required to maintain the session and avoid running the oracle in every request, e.g., when brute-forcing for sensitive files and directories. Finally, disabling the URL clustering module, which will no longer create any new clusters nor redirect already clustered pages, can be useful if the scanner wants to perform thorough checks in all pages, e.g., harvest e-mails or registered users.

To utilize the API, the scanner simply sends its API requests to ReScan through the same port it uses to proxy the requests targeting the web application, i.e., via the same HTTP channel. ReScan then identifies these API requests and performs the requested operation. It is worth noting that different components of the system are responsible for handling different API calls. For example, retrieving the entire navigation model or a specific workflow requires communicating with the *graph worker*, checking the authentication state requires invoking a *browser worker*, while enabling or disabling modules is handled by the *orchestrator*.

**Passive endpoints:**

- `/graph`: Return the entire navigation model for the target application.
- `/workflow`: Given a request, return its workflow from the model.
- `/isd`: Given a request, return the detected ISD sinks.
- `/isdAll`: Return all detected ISD sinks and sources.
- `/auth`: Return if we are currently authenticated.
- `/xss`: Return all successful XSS injections so far.
- `/isClustered`: Given a URL, return if clustered and what cluster it belongs to.

**Active endpoints:**

- `/[enable|disable]ISD`: Enable/disable ISD detection and sink collection.
- `/[enable|disable]Auth`: Enable/disable authentication helper.
- `/[enable|disable]Events`: Enable/disable event discovery.
- `/[enable|disable]Clustering`: Enable/disable URL clustering.

### A.3 Scanners' Configuration

- **w3af**. We enabled the `web_spider` plugin for crawling, the `auth.generic` plugin for authentication and the `xss` plugin for auditing with the `persistent_xss` parameter set to true for vanilla runs and false for ReScan.
- **wapiti**. We used the `wapiti-getcookie` utility for authentication, and enabled the `xss` and `permanentxss` auditing modules, disabling the latter for ReScan.
- **Enemy of the State**. The following command was issued:

```
$ jython crawler2.py <target URL>
```

- **ZAP**. The spider and `ajaxSpider` plugins were used for crawling (`ajaxSpider` was disabled for ReScan) and ZAP's standard authentication module. For auditing, we enabled the `xss` module, which incorporates both reflected and stored XSS detection, while for ReScan we enabled only the `xss_reflected` module.
- **Black widow**. The following command was issued:

```
$ ./crawl.py --url <target URL> --username <username>  
--passwd <password>
```

- **ReScan**. The following command was issued:

```
$ ./rescan.py --headless --workers 4 --isd --events  
--clustering --auth --port <proxy port>
```

# Appendix B

## B.1 JS inclusions

In Table B.1 we outline the JS inclusion methods StyxJS captures and compare them with the two original systems we retrofitted on top of it, namely SugarCoat’s underlying PageGraph and ScriptProtect. ScriptProtect has missed several inclusion methods and several others are *partially* captured. For instance, the system captures the injection of new code via `innerHTML`, but *only* if it is utilized to inject markup. However, we found that the same API can be used to set a script’s contents which leads to direct code execution; StyxJS captures such cases as well. As extensively outlined in §4.4.1, while PageGraph is able to capture all inclusion methods, it cannot *attribute* all of them back to the calling script (e.g., `setTimeout` for string-to-code evaluation), leaving room for trivial bypasses.

## B.2 PageGraph Validation

As mentioned in §4.3.1, we tried to validate StyxJS against PageGraph in real websites by comparing the scripts each system captures. However, when setting up our experiment, we found that PageGraph can *sometimes* incorrectly capture scripts that *never* execute. For instance, if an `onclick` event handler is set on an element and it does not execute, i.e., the click event is never fired on that element, PageGraph marks it as executed. On the other hand, StyxJS includes it in the JIG but does *not* mark it as executed since its entry code never runs. In a different case, if a text node is appended to a script element which is not yet attached to the DOM and therefore does not execute, PageGraph will *not* capture it, contrary to the previous case. StyxJS behaves consistently, since it also includes the script in the JIG, but does not mark it as executed. Unfortunately, this inconsistent behavior effectively prevents us from performing a correct validation experiment. Moreover, identifying and accounting for *all* cases PageGraph might exhibit such behaviors is out of the scope of this work. Nonetheless, in our test HTML page, where we know precisely what JS code executes and can account for such behaviors, we found 11 distinct dynamic scripts that PageGraph failed to attribute to a 3P, while StyxJS correctly captured all of them. All cases are attributed to

Table B.1: JavaScript inclusion methods covered by different systems. Exposing properties are marked with \*.

JS API / property	Condition	PageGraph	ScriptProtect	StyxJS
<b>HTMLScriptElement</b>				
src	-	✓	✓	✓
*text	-	✓	✓	✓
*innerText	-	✓	✓	✓
*textContent	-	✓	✓	✓
append()	Text arg(s)	✓	✗	✓
appendChild()	Text arg	✓	✗	✓
prepend()	Text arg(s)	✓	✗	✓
insertBefore()	Text arg	✓	✗	✓
replaceWith()	Text arg(s)	✓	✗	✓
replaceChild()	Text arg	✓	✗	✓
replaceChildren()	Text arg(s)	✓	✗	✓
<b>Text</b> Script's child				
*textContent		✓	✗	✓
*outerText		✓	✗	✓
*nodeValue		✓	✗	✓
*data		✓	✗	✓
<b>Element</b>				
*innerHTML	-	✓	☆	✓
*outerHTML	-	✓	☆	✓
insertAdjacentHTML()	-	✓	☆	✓
setAttribute()	Event	✓	☆	✓
<b>Attr</b>				
*value	Event	✓	✗	✓
<b>HTMLAnchorElement</b>				
*href	javascript: URL	*	✗	✓
<b>HTMLIFrameElement</b>				
*src	javascript: URL	✓	✓	✓
*srcdoc	-	✓	☆	✓
<b>Range</b>				
createContextualFragment()	-	✓	✗	✓
<b>document</b>				
write()	-	✓	☆	✓
writeln()	-	✓	☆	✓
<b>window</b>				
eval()	-	✓	✗	✓
Function()	-	✓	✓	✓
setTimeout()	-	*	✓	✓
setInterval()	-	*	✓	✓

✓: captured, ✗: not captured, ☆: partially captured, \*: Captured, but not attributed.

PageGraph's shortcomings, as detailed in §4.4.1.

