THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING

Spidering the Modern Web

Securing the Next Generation of Web Sites and Browser Extensions

ERIC OLSSON

CHALMERS

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2025

Spidering the Modern Web

Securing the Next Generation of Web Sites and Browser Extensions

Eric Olsson

ISSN 1652-876X

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden, 2025

Spidering the Modern Web
Securing the Next Generation of Web Sites and Browser Extensions
Eric Olsson
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Given the range of critical and sensitive services available on the Web, securing the web applications and browser extensions in this ecosystem is of paramount importance. However, this goal has not been achieved. Vulnerabilities in web applications remain undetected, and malicious browser extensions are still available in curated app stores.

While black-box scanning is a promising method for detecting vulnerabilities in diverse web applications, crawling these increasingly client-side and stateful applications is challenging. To discover vulnerabilities in modern web applications, we develop two new scanning methods that take into account these challenges.

We first propose a novel grey-box method, Spider-Scents, for detecting stored XSS vulnerabilities that avoids these challenges by relaxing the problem to finding unprotected outputs from the database. This method supplements an otherwise black-box scanner with the ability to directly inject payloads into the database. In our evaluation, we demonstrate that these code smells are highly related to complete vulnerabilities while showcasing the improved vulnerability detection and database coverage of our method.

We then propose a new black-box scanner, SpiderSapien, with the aim to test deep states in modern web applications, by generating valid client-side actions and form inputs that could unlock previously untested functionality. In our evaluation, we show that SpiderSapien improves vulnerability detection and code coverage, while the LLM-powered method solves more diverse forms.

Finally, we develop a framework to find fake reviews from the metadata of extensions on the Chrome Web Store. We identify how reviews can be faked, and propose five statistical methods to detect them. We demonstrate how these methods find fake reviews, and show how this can be used to find malicious extensions.

Keywords: Web application security, Web application scanning, Vulnerability detection, Browser extensions.

List of publications

This thesis is based on the following publications.

Papers A and C are published at peer-reviewed conferences, and Paper B is a manuscript. In this thesis, the full version of each paper is presented.

Paper A "Spider-Scents: Grey-box Database-aware Web Scanning for Stored XSS"
Eric Olsson, Benjamin Eriksson, Adam Doupé, and Andrei Sabelfeld
33rd USENIX Security Symposium (USENIX Security), August 2024..

Paper B "SpiderSapien: Client-Centric Crawler and Security Scanner"
Eric Olsson, Benjamin Eriksson, Adam Doupé, and Andrei Sabelfeld
Manuscript.

Paper C "FakeX: A Framework for Detecting Fake Reviews of Browser Extensions"
Eric Olsson, Benjamin Eriksson, Pablo Picazo-Sanchez, Lukas Andersson, and Andrei Sabelfeld
19th ACM ASIA Conference on Computer and Communications Security (ACM ASIACCS 2024).

Acknowledgments

My research has never been a solitary task. I am sure I would not be writing this thesis without the help and support I have received. First, I would like to thank my supervisor Andrei Sabelfeld for all of our work together, from first giving me a taste of interesting security problems in his course, to all the papers and projects we have now worked on together. Mohammad has also been a familiar presence throughout my time at Chalmers. While he has now left for Stockholm, he somehow continues to lend a helping hand, from first getting my Master's thesis on track, to almost anything else (really!). Similarly, my co-supervisor Benjamin has been invaluable in assisting me with both the broad and fine details of our projects. My coauthors, especially Adam and Pablo, have also given me an invaluable perspective on the larger field of security. This field only ever grows, and it seems like almost anything can, and will, become relevant again. Thanks to all my other colleagues and classmates for all the discussions we've had and the work we've done together. Finally, a big thanks to my family, and especially Anna, for their support.

Contents

Overview

I

Introduction

I.1 The modern Web

Much of the world relies on the Web to access sensitive services critical to their daily lives. These can range from filing taxes to voting, accessing financial services, communicating with messengers, and many more. In fact these Internet-accessible services are used by so many people that the Internet itself is seen as critical infrastructure, which entire nations depend upon [4]. While the utility of these services is obvious, the security of the ecosystem providing these services is less apparent. In particular, the security of the burgeoning Web ecosystem is more nuanced and sensitive to misbehaving code, either vulnerable or malicious, in any of its various components. Our research can be divided into securing two key aspects of the modern Web: web applications and browser extensions.

The *web applications* that underpin this modern Web are not fundamentally different from those that existed a decade ago. Web applications for these various services are still implemented in a dizzying variety of languages and frameworks. These applications still need to be secured, in part by *vulnerability scanners* which can identify vulnerabilities for developers during both their initial and continued development lifecycle. Key players in this ecosystem still pay increasing million-dollar bounties, even for classic vulnerabilities such as XSS [11, 17]. Despite the relatively simple solutions for these vulnerabilities, as web applications implement or extend their complex business logic, there persists a substantial risk of developers inadvertently including code escalating to a vulnerability. Furthermore, the greater diversity of possible implementation choices, in decisions spanning languages and frameworks both client- and server-side, has only made the task of vulnerability detection even more challenging. The prior challenges of handling increased client-side functionality [24] and statefulness [7], which distinguished an earlier version of the modern Web from its static predecessor, also remain. While there is no unique challenge to scan the latest web applications for vulnerabilities, there is instead an *increased scale* of all these prior client-side and stateful challenges.

Users can also install *browser extensions* to customize and extend their browsing experience. These small applications, built with HTML and JavaScript, can interact with and modify web pages, and are essential features for millions of users across the world [3]. While privileged access is necessary for crucial features of extensions, such as ad-blocking, the ability of extensions to observe, modify, and extract sensitive data from user's browsers has led to the proliferation of both *vulnerable* [10, 14, 31] and *malicious* [9, 15, 23, 25, 30] browser extensions. This is somewhat mitigated by the *app stores* offered by the various browser vendors, where they offer only extensions that are approved and validated to varying degrees. However, securing extensions and their users even in these curated environments has proved challenging [2]. Researchers and other security professionals continually uncover examples of both vulnerable and malicious browser extensions in extension app stores, despite their extensive security and privacy policies, and vetting through mechanisms such as code review [18].

I.2 Research efforts

In this thesis, we focus on securing web applications and browser extensions in the modern Web. To secure web applications, we develop vulnerability scanners that crawl these applications to discover Cross-Site Scripting (XSS) vulnerabilities. In the first paper [19], we investigate the relationship of stored XSS vulnerabilities to more easily discovered code smells, by integrating the database into a lightweight grey-box scanner. In the second paper [20], we return to a true black-box setting and develop a scanner capable of exploring more application functionality and its associated XSS vulnerabilities, by improving the underlying crawler's ability to interact with the rich client-side interface in modern web applications. To secure browser extensions, we examine the Web Store platform, as it is the primary means of distributing third-party extensions to users of the dominant Chrome browser. In the third paper [21], we identify how fake reviews can be used to manipulate the reputation of browser extensions on this platform, and further examine the relationship of clusters of extensions in fake review campaigns to malicious extensions.

I.3 Web applications

Discovering vulnerabilities in web applications has many similarities to the core approaches to vulnerability detection in other settings. XSS detection, as well as other detecting other injection vulnerabilities, can be seen as a taint-tracking problem. A vulnerability detection approach needs to be able to track this taint from attacker-controlled sources to vulnerable sinks in the application. Different approaches can achieve this by either dynamically exercising the application's run-time interface from a black-box perspective [8, 13, 20, 27], statically scanning the application's code in a white-box setting [5], driving a fuzzer with grey-box signals such as code coverage [12, 29], or combining two or more of these approaches for improved results. However, modern web applications pose fundamental challenges for all of these vulnerability detection approaches due to their dynamic workflows spanning multiple stateful client- and server-side components. Historically, black-box web application scanners have benefited from their design when compared to static methods, which need to both accurately model dynamic languages and track workflows across multiple component interfaces [26]. In contrast, dynamic analysis inherently aligns with the dynamic languages of the Web, and does not need to separately model and connect the interfaces across application components. Therefore, commercial black-box vulnerability scanners, such as Arachni [1], Burp [28], and ZAP [32], have entered the workflow of security professionals discovering vulnerabilities for tasks such as penetration testing. However, the promise of black-box scanning for Web vulnerabilities has been limited by the ability of these scanners to handle modern, increasingly stateful and client-side web applications. An example of a multi-step, stateful stored XSS unlikely to be discovered by prior scanners is illustrated in Figure I.1.

I.3.1 Challenge

Black-box scanners are built with crawlers that iteratively explore an application to discover more of its functionality, which can then be tested for security properties, such as XSS vulnerabilities. To be able to crawl an application and discover its entire surface, crawlers will need to be able to interact with the application, model its functionality, and prioritize different interactions. If any of these three core tasks is not sufficiently solved, a crawler will fail to discover some additional application functionality, and miss all of its associated vulnerabilities. The diversity of choices made while developing applications for the modern Web makes all three of these crawling tasks harder. These applications are implemented in myriad languages and frameworks at

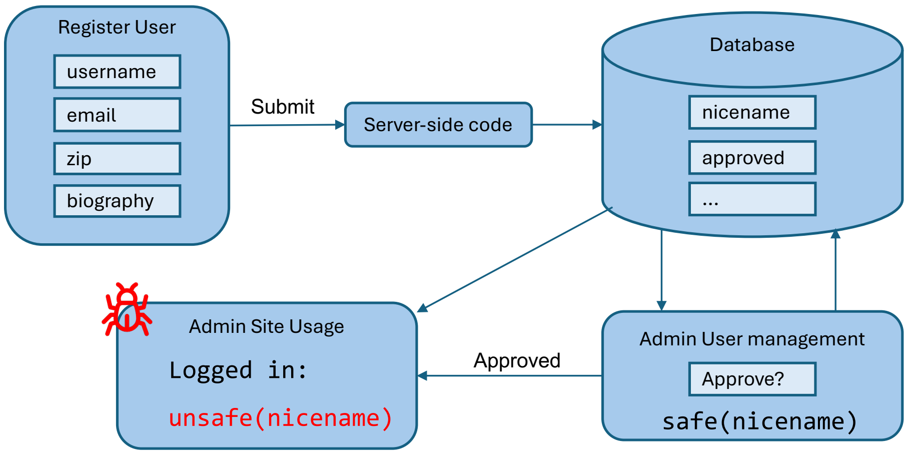


Figure I.1: State diagram of a stored XSS vulnerability. A user registers, and their username is stored as `nicename`. An admin can then approve this user. Finally, an admin will get XSS code execution from `nicename` when approved users are displayed on a site usage panel.

both the client and server. A crawler that only identifies elements by, for example, the standard HTML types, will fail to discover all actions present on a given page. Web applications are also increasingly stateful. These states are presented both client- and server-side. A crawler needs to be able to model this, in a black-box fashion that handles drastically different applications. Failing at this can lead to problems such as wasted time or infinite scans, as well as irreproducible vulnerability discovery. These application states also need to be considered when prioritizing actions, the strategy for which again needs to scale to multiple applications. Some actions only become possible in certain states, and can disappear if an 'incorrect' action is selected which leads to a necessary part of this state being removed.

I.3.2 Contributions

These challenges for black-box scanners result in the difficulty of discovering the XSS vulnerability in Figure I.1. They can be addressed in at least the two ways presented in this thesis, one specific to stored XSS, and another general to vulnerability scanning.

In the first paper in this thesis [19], inspired by work in binary fuzzing, we aim to improve the discovery of stored XSS vulnerabilities by relaxing the requirement that a scanner must find the whole stored XSS. Detecting the complete path from input source to output sink has inherent challenges to

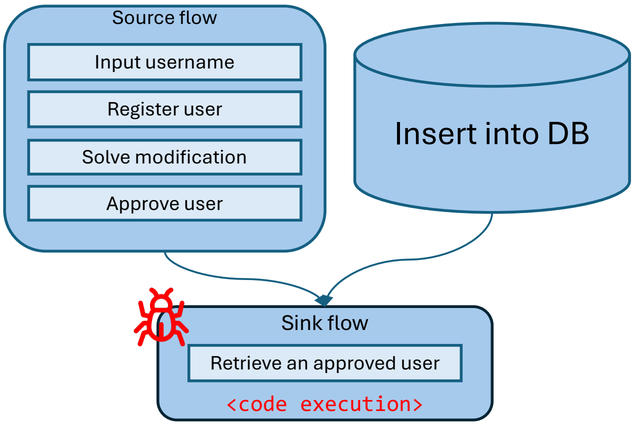


Figure I.2: Source and sink flows corresponding to Figure I.1. On the top left, we see how a successful black-box scanner would model the steps in the source flow. Rather than discovering and connecting each of these steps, Spider-Scents can use its grey-box access to the database to directly inject a payload, and only have to observe its sink flow execution.

both discover sources and connect them with sinks. Instead, by supplementing an otherwise black-box scanner with the ability to access the database to directly inject payloads, and crawling the application to discover the payload's reflections, the scanner can avoid these challenges as illustrated in Figure I.2. Thereby, the grey-box Spider-Scents scanner can discover *unprotected outputs* where database content is included in HTML without proper sanitization. These *code smells* are demonstrated to be highly related to complete stored XSS vulnerabilities in our evaluation. Overall, we find 85 stored XSS vulnerabilities across 7 web applications, a notable increase over other compared black-box scanners.

In the second paper in this thesis [20], we return to the problem of general black-box scanning, and aim to improve it without adding additional access, or overly targeting our method to a particular implementation language or framework. We develop a novel method for driving scanners to test *deep states* in modern web applications, by focusing the scanner on generating valid client-side actions and form inputs. This is driven by our intuition that these actions and inputs can unlock hidden states, which contain previously untested application functionality. Overall, the SpiderSapien crawler finds 36 XSS in 6 web applications, vastly more than all other compared black-box scanners, while also substantially improving the server-side code coverage of the applications. Furthermore, we also evaluate the capabilities of the LLM-powered form solving component in isolation and show that this aspect of the method also improves upon previous work.

I.4 Browser extensions

Browser extensions have become an essential feature in browsers, by enabling end-users to customize their browsing experience. While useful, even for various security tasks, browser extensions can also bring additional risk to the end-users that install these third-party code in their browsers. Besides the original browser and web application server, the code included in these extensions also becomes relevant to the security and privacy of a user browsing a website. By design, these extensions can be highly privileged as they need to perform sensitive actions for their core functionality. Therefore, from early in their inception, the pressing need to analyze browser extensions from a security perspective was recognized [16]. Prior research has found numerous security-relevant browser extensions, including both vulnerable and malicious examples [9, 10, 14, 15, 23, 25, 30, 31]. Vulnerable extensions, while not including any overtly malicious functionality, nevertheless introduce additional vulnerabilities for their users, possibly in particular combinations including browsers, web applications, and even other extensions. These can be exploited by malicious applications and extension developers for various goals. On the other hand, malicious extension developers directly take advantage of the end users who are led to install the extensions they control. In this thesis, we focus on these malicious extensions and the developers that publish them.

Browser extensions are also similar to many other software distributed today, in that they are primarily discovered and installed through app stores. These are run by the various browser vendors. Extension app stores offer the promise of reducing the risk of vulnerable or malicious extensions, by validating and vetting approved extensions to ensure their compliance with regards to security and privacy policies. Furthermore, these app stores can also offer users assistance in selecting the best extensions for them, as these platforms surface their most useful, popular, and overall well-liked examples of browser extensions across categories.

I.4.1 Challenge

Browser extensions are similar to general web applications, but smaller in their scale. Built with JavaScript, the analysis of these dynamic applications to discover vulnerable or malicious code can be similar. Dynamic analysis [15, 30], static analysis [10, 14, 23, 31], and combined approaches [9] have all been proposed. The small scale of these extensions can even make heavier forms of analysis, such as information-flow control [6], more suitable to this domain than general web applications. However, even with this smaller scale, these analyses can still fail to detect both vulnerable and malicious code in browser extensions. A further challenge is in the shift from discovering vulnerabilities, to detecting malicious behaviour. Besides the general challenges of analyzing Web software, a security analyst must also contend with malicious code that is both obfuscated against static analysis and evasive to dynamic analysis [22]. Even from the powerful vantage point of a platform extension reviewer, who might have additional information such as developer account details including insider information like source IPs, and more detailed code submissions and version-tracking, this task has proven challenging by the numerous examples of published malicious extensions evading detection. Malicious code can be easily obscured by the bulk of benign code present in an extension. Furthermore, benign extensions can be easily copied and modified to include malicious code.

I.4.2 Contributions

In the third paper in this thesis [21], we analyze the metadata of extensions, to discover the relationship of fake reviews to clusters of extensions sharing this activity. While the content of reviews can easily be faked or even directly copied, in metadata the activity of less-than-honest actors is harder, or at least more expensive to obscure. We first identify how reviews can be faked, and propose five statistical methods that can detect these techniques. Using these detection methods, we demonstrate how these methods find fake reviews on the Web Store. We also show that clusters of extensions with fake reviews can be used to find malicious extensions.

II

Statement of contributions

Below we list the abstracts of the appended papers and outline the personal contributions for each.

A Spider-Scents: Grey-box Database-aware Web Scanning for Stored XSS

Eric Olsson, *Benjamin Eriksson, Adam Doupé, and Andrei Sabelfeld*

As web applications play an ever more important role in society, so does ensuring their security. A large threat to web application security is XSS vulnerabilities, and in particular, stored XSS. Due to the complexity of web applications and the difficulty of properly injecting XSS payloads into a web application, many of these vulnerabilities still evade current state-of-the-art scanners. We approach this problem from a new direction—by injecting XSS payloads directly into the database we can completely bypass the difficulty of injecting XSS payloads into a web application. We thus propose Spider-Scents, a novel method for grey-box database-aware scanning for stored XSS, that maps database values to the web application and automatically finds unprotected outputs. Spider-Scents reveals *code smells* that expose stored XSS vulnerabilities. We evaluate our approach on a set of 12 web applications and compare with three state-of-the-art black-box scanners. We demonstrate improvement of database coverage, ranging from 79% to 100% database coverage across the applications compared to the range of 2% to 60% for the other scanners. We systematize the relationship between unprotected outputs, vulnerabilities, and exploits in the context of stored XSS. We manually analyze unprotected outputs reported by Spider-Scents to determine their vulnerabil-

ity and exploitability. In total, this method finds 85 stored XSS vulnerabilities, outperforming the union of state-of-the-art's 32.

Statement of contributions. I was responsible for developing our method beyond the initial idea of inserting payloads into a database and crawling for their reflection. This included database synthesis, reverting changes, choosing cells, structuring payloads, and identifying breakage. I also implemented this method in the published Python code, and designed and ran the majority of the evaluation of this scanner.

Appeared in: *Proceedings of the 33rd USENIX Security Symposium (USENIX Security), August 2024.*

B SpiderSapien: Client-Centric Crawler and Security Scanner

__Eric Olsson__, Benjamin Eriksson, Adam Doupé, and Andrei Sabelfeld

Black-box web application crawling and scanning plays an important role for security testing of web applications. Yet state-of-the-art scanners fall short of addressing key characteristics of a modern web application: its extreme dynamism and interactivity on the client side. This paper identifies immersive interaction as a key ingredient for scanners to deeply explore modern web applications. We propose SpiderSapien, a client-centric crawler and security scanner. Driven by immersive interaction, SpiderSapien incorporates novel methods to detect interactable elements, order UI interactions, and use LLMs to solve forms. In doing so, we demonstrate how to reliably discover and test deep states of modern web applications. The evaluation of our approach shows substantial improvements in both code coverage and vulnerability detection over previous work, with an average increase in code coverage of 21.5% compared to the *union* of the other scanners and a total of 36 XSS vulnerabilities, across 6 of the 8 web applications, compared to the 4 XSS others find. In addition, a separate empirical evaluation of SpiderSapien's LLM-powered form solving capabilities on diverse *real forms* on the open web demonstrates superiority over the previous approaches in generating desired input on the client side, solving at least 23.3% more of the non-trivial forms compared.

Statement of contributions. I was responsible for developing and implementing our LLM-based form solving method, and contributed to the client-side crawling aspect. I also designed and ran the novel evaluation setup of the form solving methods on the open Web.

Appeared in: *Manuscript*

C FakeX: A Framework for Detecting Fake Reviews of Browser Extensions

***Eric Olsson**, Benjamin Eriksson, Pablo Picazo-Sanchez, Lukas Andersson, and Andrei Sabelfeld*

Browser extensions boost user experience on the web. Similarly to smartphone app stores, browsers like Chrome distribute browser extensions via their Web Store, enabling a thriving market of third-party developed extensions. The Web Store incorporates a user review system to help users decide which extensions to install. Unfortunately, the open nature of the review system is subject to reputation manipulation. As browser vendors fight reputation manipulation, attackers employ more sophisticated methods to stay under the radar. Focusing on fake reviews, we identify several techniques attackers use: fake accounts, disjoint sets of fake accounts for different extensions, automation of generated reviews, and focusing on reviews rather than ratings. We present FakeX, a framework to detect fake reviews by focusing on inference from review metadata. FakeX employs five distinct methods, including temporal distribution analysis, relationship clustering, and ratio-based assessments, to unveil patterns indicative of fake reviews. Evaluation of over 1.7 million reviews reveals the effectiveness of FakeX in identifying hundreds of fake review campaigns. Furthermore, our investigation of these fake reviews uncovers 86 malicious extensions, mounting attacks that range from data-stealing to monetization, impacting over 64 million users. In addition, we collaborate with Adblock Plus and Avast to demonstrate FakeX in action, expanding a seed list of newly detected malicious extensions to discover a further 16 malicious extensions with millions of users, where, in some cases, attackers tried to improve malicious code.

Statement of contributions. I was responsible for developing details of the Horizontal Vertical Clustering (HVC) method, including tuning hyperparameters and selecting notable clusters. I also contributed to other data analysis throughout this paper, and was responsible for the comparison of the HVC to Aggregated Time Window (ATW) methods.

Appeared in: *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security (ASIA CCS '24), July 2024*

Bibliography

[1] Arachni. `https://www.arachni-scanner.com`.

[2] D. Bán and B. Livshits. Extension vetting: Haven't we solved this problem yet? In *Proceedings of the 1st Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*, San Diego, CA, USA, Feb. 2019. Internet Society. Co-located with NDSS 2019.

[3] Chrome Extensions Stats, 2023.

[4] CISA.gov. Critical infrastructure sectors. `https://www.cisa.gov/topics/critical-infrastructure-security-and-resilience/critical-infrastructure-sectors`.

[5] J. Dahse and T. Holz. Static detection of second-order vulnerabilities in web applications. In *USENIX Security Symposium*, pages 989–1003. USENIX Association, 2014.

[6] M. Dhawan and V. Ganapathy. Analyzing information flow in javascript-based browser extensions. In *ACSAC*, pages 382–391. IEEE Computer Society, 2009.

[7] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *USENIX Security Symposium*, pages 523–538. USENIX Association, 2012.

[8] B. Eriksson, G. Pellegrino, and A. Sabelfeld. Black widow: Blackbox data-driven web scanning. In *SP*, pages 1125–1142. IEEE, 2021.

[9] B. Eriksson, P. Picazo-Sanchez, and A. Sabelfeld. Hardening the security analysis of browser extensions. In *SAC*, pages 1694–1703. ACM, 2022.

[10] A. Fass, D. F. Somé, M. Backes, and B. Stock. Doublex: Statically detecting vulnerable data flows in browser extensions at scale. In *CCS*, pages 1789–1804. ACM, 2021.

[11] Google. `https://security.googleblog.com/2023/02/vulnerability-reward-program-2022-year.html`.

[12] E. Güler, S. Schumilo, M. Schloegel, N. Bars, P. Görz, X. Xu, C. Kaygusuz, and T. Holz. Atropos: Effective fuzzing of web applications for

server-side vulnerabilities. In *USENIX Security Symposium*. USENIX Association, 2024.

[13] X. Guo, A. Kawlay, E. Liu, and D. Lie. Evocrawl: Exploring web application code and state using evolutionary search. In *NDSS*. The Internet Society, 2025.

[14] S. Hsu, M. Tran, and A. Fass. What is in the chrome web store? In *AsiaCCS*. ACM, 2024.

[15] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *USENIX Security Symposium*, pages 641–654. USENIX Association, 2014.

[16] M. T. Louw, J. S. Lim, and V. N. Venkatakrishnan. Enhancing web browser security against malware extensions. *J. Comput. Virol.*, 4(3):179–195, 2008.

[17] Meta. `https://about.fb.com/news/2022/12/metas-bug-bounty-program-2022/`.

[18] J. M. Moreno, N. Vallina-Rodriguez, and J. Tapiador. Did I vet you before? assessing the chrome web store vetting process through browser extension similarity. *CoRR*, abs/2406.00374, 2024.

[19] E. Olsson, B. Eriksson, A. Doupé, and A. Sabelfeld. Spider-scents: Grey-box database-aware web scanning for stored XSS. In *USENIX Security Symposium*. USENIX Association, 2024.

[20] E. Olsson, B. Eriksson, A. Doupé, and A. Sabelfeld. Spidersapien: Client-centric crawler and security scanner. Manuscript, 2025.

[21] E. Olsson, B. Eriksson, P. Picazo-Sanchez, L. Andersson, and A. Sabelfeld. Fakex: A framework for detecting fake reviews of browser extensions. In *AsiaCCS*. ACM, 2024.

[22] N. Pantelaios and A. Kapravelos. FV8: A forced execution javascript engine for detecting evasive techniques. In *USENIX Security Symposium*. USENIX Association, 2024.

[23] N. Pantelaios, N. Nikiforakis, and A. Kapravelos. You've changed: Detecting malicious browser extensions through their update deltas. In *CCS*, pages 477–491. ACM, 2020.

[24] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow. jäk: Using dynamic analysis to crawl and test modern web applications. In *RAID*, volume 9404 of *Lecture Notes in Computer Science*, pages 295–316. Springer, 2015.

[25] P. Picazo-Sanchez, B. Eriksson, and A. Sabelfeld. No signal left to chance: Driving browser extension analysis by download patterns. In *ACSAC*, pages 896–910. ACM, 2022.

[26] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS*. The Internet Society, 2010.

[27] A. Stafeev, T. Recktenwald, G. D. Stefano, S. Khodayari, and G. Pellegrino. Yurascanner: Leveraging llms for task-driven web app scanning. In *NDSS*. The Internet Society, 2025.

[28] B. Suite. `https://portswigger.net/burp`.

[29] E. Trickel, F. Pagani, C. Zhu, L. Dresel, G. Vigna, C. Kruegel, R. Wang, T. Bao, Y. Shoshitaishvili, and A. Doupé. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect SQL and command injection vulnerabilities. In *SP*, pages 2658–2675. IEEE, 2023.

[30] M. Weissbacher, E. Mariconti, G. Suarez-Tangil, G. Stringhini, W. K. Robertson, and E. Kirda. Ex-ray: Detection of history-leaking browser extensions. In *ACSAC*, pages 590–602. ACM, 2017.

[31] J. Yu, S. Li, J. Zhu, and Y. Cao. Coco: Efficient browser extension vulnerability detection via coverage-guided, concurrent abstract interpretation. In *CCS*, pages 2441–2455. ACM, 2023.

[32] O. ZAP. `https://www.zaproxy.org`.

Web Applications

A

Spider-Scents: Grey-box Database-aware Web Scanning for Stored XSS

Eric Olsson, Benjamin Eriksson, Adam Doupé, and Andrei Sabelfeld

33rd USENIX Security Symposium (USENIX Security), August 2024.

Abstract

As web applications play an ever more important role in society, so does ensuring their security. A large threat to web application security is XSS vulnerabilities, and in particular, stored XSS. Due to the complexity of web applications and the difficulty of properly injecting XSS payloads into a web application, many of these vulnerabilities still evade current state-of-the-art scanners. We approach this problem from a new direction—by injecting XSS payloads directly into the database we can completely bypass the difficulty of injecting XSS payloads into a web application. We thus propose Spider-Scents, a novel method for grey-box database-aware scanning for stored XSS, that maps database values to the web application and automatically finds unprotected outputs. Spider-Scents reveals *code smells* that expose stored XSS vulnerabilities. We evaluate our approach on a set of 12 web applications and compare with three state-of-the-art black-box scanners. We demonstrate improvement of database coverage, ranging from 79% to 100% database coverage across the applications compared to the range of 2% to 60% for the other scanners. We systematize the relationship between unprotected outputs, vulnerabilities, and exploits in the context of stored XSS. We manually analyze unprotected outputs reported by Spider-Scents to determine their vulnerability and exploitability. In total, this method finds 85 stored XSS vulnerabilities, outperforming the union of state-of-the-art's 32.

A.1 Introduction

The web is a key enabler for today's ever-more digital world. Our society increasingly relies on web applications to support the financial, governmental, and military infrastructure. The dynamic functionality of web applications, coupled with the myriad of implementation technologies, makes developing a bug-free application challenging. Furthermore, these bugs can often manifest as security vulnerabilities. The complexity of modern systems and ever-powerful adversaries make securing web applications a *grand challenge*. Even the biggest web players such as Google and Meta still release vulnerable applications and services, which reflects in $12 and $2 million in bug bounties in 2022 respectively [15, 26].

Challenge of XSS. A particularly common class of web application security vulnerability is *Cross-Site Scripting (XSS)* [28], allowing attackers to inject JavaScript code into web pages. Astonishingly, XSS has persisted in the

OWASP Top 10's list of most critical security risks to web applications for the past 20 years [48]. This remarkable persistence is reflected in bug bounties, with HackerOne reported paying over $4.7 million for XSS vulnerabilities in 2022 alone [17]. Securing web applications against XSS is difficult because there is no single general solution that prevents all XSS vulnerabilities [20]. Indeed, XSS vulnerabilities are context-dependent [42, 44], requiring that the correct output sanitization be used depending on the output context.

Stored XSS. Stored XSS, where the injection is stored and only later executed [28], is particularly challenging due to the disconnect that storage brings between the *source flow*, where the payload is input and stored, and the *sink flow*, where the retrieved payload is executed. Current vulnerability detection approaches [14, 32] have fundamental difficulties finding stored XSS.

Insufficiency of ∗-box approaches. The difficulty of securing a web application against XSS motivates the development of vulnerability detection tools [2, 7, 11, 33, 49, 54]. Web application vulnerability detection approaches can be classified as white-box, black-box, or grey-box based on what information is available (cf. Section A.4):

White-box approach: White-box approaches [12, 19, 22, 24] usually statically analyze source code artifacts. Such static analysis is necessarily specific to the structure of the analyzed artifact, such as the server-side language or framework. Unfortunately, white-box vulnerability detection is fundamentally limited in its applicability to web applications because it is hard for white-box static analysis to precisely model the combined interplay of increasingly complex and dynamic client-side, database, and server-side behavior [43]. In addition, white-box analyses depend on the availability of artifacts, further limiting their usage.

Black-box approach: More advantageously from a usability perspective, black-box vulnerability detection for web applications does not typically require access to source code and instead analyzes a running web application from the perspective of a user. Black-box scanners have been developed with various methods to better cover the increased attack surface of modern web applications [8], such as modeling server-side state [7], tracking data flows and fuzzing payloads [9, 10], modeling client-side state [25, 33], and combining multiple approaches [11]. However, while coverage of the attack surface has improved for some XSS, black-box scanners are often still unable to find even simpler stored XSS [32].

Grey-box approach: A common solution [1, 14, 18, 49, 50] is to combine black-box dynamic interactions of a running application, with white-box access. How these two information sources are combined varies. Sometimes, artifacts

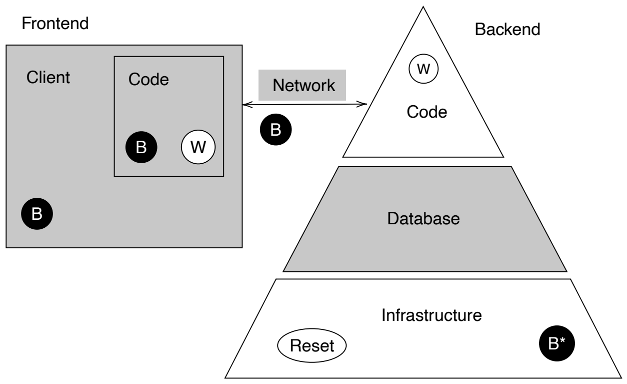


Figure 1: Access to components involved in a web application that different approaches need. Black-box approaches (*B*) have access to the front-end, and sometimes (*B) need infrastructure access to perform resets in the face of irreversible state changes. White-box approaches (*W*) have access to both front- and back-end code. Our approach, shaded in grey, accesses the front-end and database.**

or non-standard interfaces only available with white-box access can be used to guide the otherwise black-box scan of a web application [47] . More commonly, a fully white-box static analysis of the application source code is combined with a black-box scan for dynamic runtime information [1, 14]. Combining these sources of information can mitigate some challenges inherent to otherwise using one approach in isolation. However, this combination still has a white-box component, from which its usability suffers.

While the previously described approaches constitute an exciting and active area of research, we identify a key consideration in the design space of $*$-box approaches. A core problem with finding stored XSS is that a *black-box scanner must find both the source and sink flows, and also understand the relationship between the two*, without any access to the web application's source code.

Approach. Inspired by recent work in improving binary fuzzing [34], our insight is to make stored XSS easier to find by relaxing the requirement that the scanner must find the source of a stored XSS. To do so, we supplement an otherwise black-box scanner with access to the database and allow the scanner to inject payloads directly into the database while scanning the running web application for sensitive sinks that output the inserted payload. This yields the benefit that a scanner no longer needs to find both source and sink nor understand their relation. The elegance of our approach is that it requires no knowledge of the web application source, only the database. Figure 1 illustrates the unique point our approach occupies in the design space of ∗-box approaches.

This unique position represents a paradigm shift from the domain of *application input* to that of *application state*, represented in the database. Challenges (presented in Section A.3) stemming from fundamental problems with *-box approaches do not have to be solved by our method.

Based on these insights, we develop Spider-Scents, an approach to grey-box database-aware web scanning for stored XSS. Spider-Scents injects payloads directly into the database and reports where database content is used in the HTML output without proper sanitization, flagging what we call *unprotected outputs*. This does not mean that all reports are stored XSS vulnerabilities, as the web application might be sanitizing the data on input. However, relying on input sanitization is against best practices for XSS prevention, as it is impossible to sanitize user input for every possible HTML output context. Indeed, the OWASP guidelines [31] postulate: “Apply Input Validation (using "allow list" approach) combined with Output Sanitizing+Escaping on user input/output,” confirming that input validation alone is not enough. We term this result a *code smell*, an indication that something is wrong deeper within the application [13]. Even in the best case, where there is neither bug nor vulnerability, the application is fragile. Any new functionality added, such as creating a REST API, risks failing to properly sanitize user input. Therefore, even the unprotected outputs that are not currently exploitable stored XSS should be addressed by the web developer. Light manual analysis is required to verify a complete stored XSS vulnerability (see Section A.6.7)

Non-vulnerable unprotected outputs constitute what we call a *dormant XSS*—they would be vulnerable, except that the *current* web application does not allow an exploit payload. The web application’s evolution risks elevating a dormant XSS to a *complete XSS vulnerability*, even for security-wary applications. Our empirical study indeed confirms a dormant XSS vulnerability on WordPress, elevated to a complete vulnerability by a *real* published plugin (see Section A.4.2).

Evaluation. We evaluate our approach across 12 web applications and compare our results with three state-of-the-art black-box scanners. The applications range from reference applications used in prior work to latest versions of modern applications. Our results show that we cover (measured as how much of the database the scanner can change, described in Section A.5.2) between 79% to 100% across all applications. In comparison, the other scanners cover 2% to 60% on average. We also find vulnerabilities that the other scanners are unable to detect. In total, we find 85 XSS vulnerabilities compared to 32 unique XSS for the other scanners. To further classify the impact of our findings we manually analyze the input protections and permission models of the Spider-

Scents discovered vulnerabilities and determine that 59 are exploitable (and not self-XSS).

Contributions. We offer the following contributions:

- We present a novel approach for finding stored XSS vulnerabilities by injecting XSS payloads directly into the database, thus simplifying the detection of stored XSS. We present this in Section A.4.
- We implement our approach into a prototype Spider-Scents, a semi-automated grey-box database-aware stored XSS scanner.
- We evaluate Spider-Scents and three state-of-the-art black-box scanners on 12 web applications. We present the results in Section A.5 and analyze these in Section A.6. Spider-Scents finds 85 XSS vulnerabilities across 7 applications.
- We systematize the relationship between *unprotected outputs*, *stored XSS vulnerabilities*, and *exploits* based on input protections and permissions models. Following this systematization, we manually analyze unprotected outputs reported by Spider-Scents to determine their vulnerability and exploitability. We also present this in Section A.6.
- For the benefit of future research in this area, we share the source code of Spider-Scents[1].

Ethical considerations and coordinated disclosure. By actively scanning only our local clones of web applications in a controlled environment, we strictly avoid any harm caused by scanners on the web. We handle the discovered security vulnerabilities in accordance with the best practices of ethics in security [41]. We are in the process of reporting our findings to the affected vendors, following coordinated vulnerability disclosure for all discovered vulnerabilities. We report responses from vendors in Section A.6.9.

A.2 Terminology

Here we attempt to systematize the terminology around XSS vulnerability analysis. Spider-Scents finds places in the web application where database content is used in the HTML output without proper sanitization. We term these *unprotected outputs*—output sinks where the output is not protected sufficiently against XSS. In contrast, there are also *protected outputs*: output sinks that are properly sanitized against XSS.

[1]Our implementation is available online at `https://www.cse.chalmers.se/research/group/security/spider-scents/`

We call a *complete XSS vulnerability* where user input flows to the unprotected output. Furthermore, that input must itself be an *unprotected input*—an input source lacking sufficient XSS protection. In contrast, there are also *protected inputs*: inputs protected with some combination of *sanitization* such as validation, stripping, or escaping. An unprotected output can also fail to be a complete XSS due to having *no input*.

If an unprotected input flows to an unprotected output, the web application has an *XSS vulnerability*. However, an XSS vulnerability is not necessarily *exploitable*, as this depends on the access control policy of the web application. The core question is if the user (or role) that injects the XSS payload can get the output on either another user or a role with greater permissions. We call XSS vulnerabilities that are not exploitable *self-XSS*, which have significantly less severity than exploitable XSS vulnerabilities.

A.3 Roadblocks for current XSS scanners

Automatically finding vulnerabilities in web applications remains a challenge despite active research in improving vulnerability detection. Stored XSS is especially difficult to find, as this type of vulnerability involves correctly injecting input into the application, where it will be stored by the database, and subsequently used in the web application's output incorrectly sanitized.

Black-box approaches can explore entire web applications without access or reliance on the underlying web application source code. It is possible for black-box scanners to track an entire stored XSS vulnerability from initial payload injection to vulnerable output. However, finding all such vulnerabilities for a black-box scanner is difficult, due to several challenges inherent to stored XSS that they must solve:

Vulnerable input validation. Web applications perform input validation (in client-side JavaScript and also server-side code) to ensure that the input data conforms to certain requirements. Web applications can check vulnerable inputs for validation that the scanner must pass and also inject an XSS payload into. As most black-box scanners use a pre-configured list of XSS payloads, it is difficult for them to create a custom XSS payload that also bypasses the vulnerable input validation.

Interdependent vulnerable input validation. Web applications validate all types of inputs. Often, a web application requires the user to fill out a set of inputs together. Consider a user registration form that might require the desired username, email address, zip code, and biography. All forms are required, the server-side input validation requires that the username be unique, the email address has a specific form, the zip code is five digits, and the biography has no validation and is vulnerable to stored XSS. Due to the *interdependence* of the four inputs, to find the stored XSS a black-box scanner must be able to provide a unique username (which is difficult for repeated injection attempts), correctly-formatted email address and zip code. As the number of inputs interdependent to vulnerable input increases, and as the input validation is application-specific, it is more difficult for black-box scanners to generate the proper interdependent input to inject stored XSS payloads.

Vulnerable input modification. Web applications can also escape or modify user input. Consider a blog that accepts blog posts in markdown format that is transformed to HTML before storing them in the database. Because the black-box scanner has no knowledge of the server-side source code, it cannot know about this modification. If a vulnerable input is modified before being stored into the database, it is difficult for a scanner to create a custom XSS payload that can survive the modification, and, therefore, it is difficult to detect.

Multi-step vulnerable input. Web applications are stateful applications that can require a multi-step process before persisting user data. A classic example of this is the multi-step stored XSS in WackoPicko [8], where commenting on a picture requires first previewing the comment (which is output with sanitization) and then approving the comment (where it is output without sanitization). To our knowledge, the first black-box scanner that was able to automatically detect this vulnerability was Black Widow [11], published 11 years after WackoPicko was released. Therefore, vulnerable inputs that require multi-step interactions are difficult for black-box scanners to detect.

Vulnerable input identification. The core of stored XSS is that the XSS payload is stored in the database before being used as output. Even if a black-box scanner can correctly inject an XSS payload, it must be able to find where that input is output—otherwise, it will never detect the XSS.

A.4 Approach

Our goal is to overcome the challenges black-box scanners face (mentioned in Section A.3) in detecting stored XSS vulnerabilities. Rather than take a completely black-box approach, we use a novel grey-box approach that includes *knowledge of the database* to help an otherwise black-box scanner find stored XSS vulnerabilities. Our idea is that by injecting the XSS payload directly into the database, and then scanning the web application for the payload's output, we can *completely bypass* several of the roadblocks black-box scanners face in detecting stored XSS: vulnerable input validation, interdependent vulnerable input validation, vulnerable input modification, and multi-step vulnerable input.

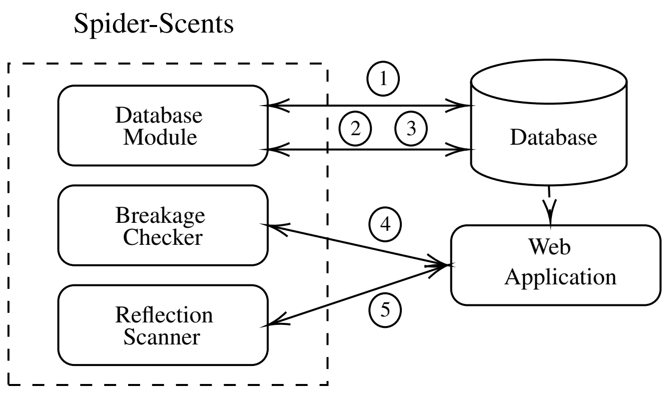


Figure 2: Overview of Spider-Scents' different components and their interactions with both the database and web application.

'Grey-box'. In web security, black-box has been synonymous with dynamic testing, as white-box is to static analysis. With this view, grey-box can appear to be defined as the combination of these two: dynamic testing and static analysis. Recent papers [14, 49] in grey-box web testing have supplied two definitions: the previous, based on *method*, or based on *access* or *visibility* to the application. Our approach is grey-box only by the second definition: we do not use static analysis, but we have access to the database.

Besides the application data contained, metadata such as table structure and associated relations are also assumed to be available, through the same database connection. In practical terms, SQL databases provide programmatic access to such metadata in the `INFORMATION_SCHEMA` tables [30].

A.4.1 Overview

In contrast to existing state-of-the-art scanners, our database-aware method requires a paradigm shift from application *input* to *state*. Instead of solving problems associated with the input domain (such as crawling, modelling, or payload selection), we address new challenges of preparing the application state, selecting which part to modify, and analyzing its impact on the application.

Spider-Scents is our implementation of our novel grey-box scanning approach. Figure 2 shows a diagram of our approach. First ①, we prepare the web application for scanning. Next ②, we choose a database cell to modify, and ③ insert an XSS payload into the cell. We then validate ④ that the modified database did not break the application. Finally ⑤, we crawl the web application looking for reflections of the injected payload. This approach iterates as long as there remain untested database cells to modify.

Reports

Due to Spider-Scents injecting XSS payloads directly into the database, what it reports *is not XSS vulnerabilities*, but rather *unprotected outputs* (defined in Section A.2). Therefore, the final step is to manually analyze the results of Spider-Scents. As our evaluation in Section A.5 shows, many unprotected outputs are also XSS vulnerabilities—Spider-Scents finds 133 unprotected outputs in evaluated applications and 85 XSS vulnerabilities.

An interesting side-effect of finding so many vulnerabilities with Spider-Scents is that we realized that the *impact* of the discovered XSS vulnerabilities is critically important to contextualize the results. Specifically, we found that some of the XSS vulnerabilities were *self-XSS*, defined in Section A.2 wherein the privilege required to store the XSS payload is the same as the user that views it. Of the 85 XSS vulnerabilities found by Spider-Scents, 26 are self-XSS while 59 are fully-exploitable.

A.4.2 Motivating Examples

Spider-Scents' database-aware scanning is particularly well suited to finding stored XSS vulnerabilities due to circumventing common challenges for black-box scanners. Here, we present examples of the types of issues that Spider-Scents is better at finding than previous work:

1. Fully-exploitable stored XSS that other scanners do not find.
2. Dormant vulnerabilities that become exploitable.
3. Self-XSS that other scanners do not find.

Fully-exploitable stored XSS. Even though they are capable of finding a fully-exploitable stored XSS vulnerability, other scanners often fail due to their inability to extensively explore the input surface of the web application, including satisfying vulnerable input and interdependent input validation. In Section A.5, we further analyze the precise reasons why other scanners miss fully-exploitable stored XSS that Spider-Scents finds.

In addition, beyond needing a full XSS from input to output, the XSS vulnerability also must be exploitable. Determining the exploitability of an XSS currently requires manual analysis, as the exploitability of a vulnerability depends on application context such as the levels of user permissions within the application. Other presentations of scanners have generally not provided such analysis of their XSS results.

An example of a fully-exploitable stored XSS that other scanners fail to find is in the CMS Made Simple bookmarks functionality, which is shown in

```
echo "<td><a␣href=\"editbookmark.php".$urlext."&amp;bookmark_id=".
    $onemark->bookmark_id."\">".$onemark->title."</a></td>\n";
echo "<td>".$onemark->url."</td>\n";
echo "<td><a␣href=\"editbookmark.php".$urlext."&amp;bookmark_id=".
    $onemark->bookmark_id."\">";
echo $themeObject->DisplayImage('icons/system/edit.gif', lang('edit'),
    '','','systemicon');
echo "</a></td>\n";
echo "<td><a␣href=\"deletebookmark.php".$urlext."&amp;bookmark_id=".
    $onemark->bookmark_id."\"␣onclick=\"return␣confirm('".
    cms_html_entity_decode(lang('deleteconfirm', $onemark->title) )."
    ');\">";
echo $themeObject->DisplayImage('icons/system/delete.gif', lang('
    delete'),'','','systemicon');
echo "</a></td>\n";
```

Figure 3: CMS Made Simple bookmarks functionality. A bookmark contains a title and a URL. Both the title and URL are correctly escaped in their respective first two cells in this snippet from CMSMS code, but the title is not protected in its inclusion in the delete button.

Figure 3. In this case, it is harder to find the input form, but easy to find the output from the database. Armed with this vulnerability, a user can perform XSS on an admin of CMSMS. Spider-Scents reports this as an unprotected output, and we manually confirm its exploitability.

Dormant XSS. Spider-Scents reports what it finds as unprotected outputs rather than stored XSS vulnerabilities, as it only finds the second half of a complete XSS workflow in unprotected outputs. Unprotected outputs do not always have an unprotected input flowing to them. However, we believe there is significant value in reporting unprotected outputs because they are a *code smell*—the web application has not followed the best practice of “always escape late” emphasized by both WordPress and WordPress VIP [51, 52]. While not a bug or vulnerability, an unprotected output is something that a developer should look at and fix.

In fact, we believe that an unprotected output in isolation can be considered a *dormant XSS*—it would be vulnerable, except that the current inputs for the web application do not allow an exploit payload due to either escaping, stripping, validation, no input possible, or some combination of these. The web application’s evolution, either through future development or integration with other code (plugins, for example), can elevate a dormant XSS to a full XSS.

For example, Spider-Scents found that WordPress has unprotected output of both `display_name` and `user_nicename` from the `users` table. In the base WordPress application, there are no unprotected inputs to these columns—in

```
$qnn = $wpdb->prepare( "UPDATE␣$wpdb->users␣SET␣user_nicename␣=␣%s␣
    WHERE␣user_login␣=␣%s␣AND␣user_nicename␣=␣%s", $new_username,
    $new_username, $old_username );
$wpdb->query( $qnn );

$qdn = $wpdb->prepare( "UPDATE␣$wpdb->users␣SET␣display_name␣=␣%s␣
    WHERE␣user_login␣=␣%s␣AND␣display_name␣=␣%s", $new_username,
    $new_username, $old_username );
$wpdb->query( $qdn );
```

Figure 4: Username-Changer WordPress plugin vulnerable code: `display_name` and `user_nicename` are not sanitized.

fact, these are not modifiable after the creation of a user. However, *both* are exposed in a vulnerable version of the `username-changer` plugin [5] (code shown in Listing 4), and, therefore, when this vulnerable plugin is installed this unprotected output becomes a fully-exploitable XSS. [2] If WordPress followed their own best practices of "always escape late", this dormant XSS would not be possible (and Spider-Scents would not report it as an unprotected output).

It is also possible that some inputs to the database are kept constant. For example, in Hostel Management System, an application we evaluate in Section A.5, a list of US states is hard-coded in the database. Spider-Scents finds unprotected outputs in this list. While not directly exploitable, these unprotected outputs could become a problem through extension of code, either in future versions of the application or with plugins. In our manual analysis, we quantify how many reports fall into this category and present them in the *NI* column in Table 2.

Self-XSS. Finally, an XSS vulnerability is not necessarily exploitable. An XSS can be unexploitable due to user permissions, such that the admin can only perform an XSS on themself. This self-XSS is the counterpoint to fully-exploitable XSS. An example of this is templates in MyBB. A template for the calendar functionality in this application can be modified to include executable JavaScript. However, only an admin user has the necessary permissions to add or modify this template. Spider-Scents finds this vulnerability, and we manually confirm that it is a self-XSS.

[2]Vulnerable version of WordPress `username-changer` plugin https://github.com/evertiro/Username-Changer/blob/dd1976b05213d9895886da7f9a91515c52188344/includes/functions.php#L84

Algorithm 1 Synthesizing data in the database.

```
$rows \leftarrow 3$
$i \leftarrow 1$
while $i \leq rows$ do
    $row \leftarrow []$
    $j \leftarrow 0$
    while $j \leq columns$ do
        $append(row, increment(i))$
        $j \leftarrow j + 1$
    end while
    $insert(row)$
    $error \leftarrow breakage()$
    if $error$ then
        $delete(row)$
    end if
    $i \leftarrow i + 1$
end while
```

A.4.3 Preparing the web application

The initial step in our approach is to prepare the web application's database for scanning, as shown in step ① in Figure 2.

Database synthesis

Ideally, our method should scan an application with a *full* database. Steinhauser and Tůma note the importance of this as well [47]. However, they do not attempt to solve this and instead rely on somewhat complete configurations provided by the applications themselves, or other publicly-available manually-assembled data. For an e-commerce website, this means that the database already contains products, customers, and other data. To be able to discover an XSS-vulnerable workflow that spans multiple tables, data must exist in each table.

To address cases where data is lacking, more specifically empty database tables, we insert a constant number of rows of benign data matching the schema of the empty tables. We perform this simple algorithm shown in Section A.4.3 on each empty table in the database.

columns is the number of columns in the schema for the empty table and $increment(i)$ modifies a base value by the increment i. For example, integers have a base value of 0, dates have '1970-01-01 00:00:00', and strings have `a`. By modifying a constant value, we ensure that there are records correlated between tables by these deterministic values, to satisfy constraints common in web applications using normalized databases. We always insert 3 initial rows due to different auto-increments in the applications' database setups—we have observed empirically that these often start at 1, but not always.

While this naive solution is implemented and works well for our evaluation, the more general case of database synthesis is orthogonal to this work. Related work in this specific direction can be found in Section A.7.5.

Reverting changes

We also periodically revert modifications done by Spider-Scents to the database, the rules for which are described in Appendix A.1.1. This is to add independence between our payload insertions, and also reduces our reliance on detecting application breakage, if we automatically revert based on other rules. Some rules we implement enforce independence across boundaries in the database; such as tables and columns.

To revert a database edit, from *newData* to *oldData*, perform:

```
$error \leftarrow updateRow(oldData)$
if $error$ then
    $removeRow(newData)$
    $insertRow(oldData)$
end if
```

removeRow and *updateRow* can identify a row based on either keys or values. It is necessary to handle reverting changes in Spider-Scents, as built-in database functionality such as transactions cannot be open-ended, which is necessary to allow the simultaneous manipulation of both Spider-Scents and the web application backend of the database.

Reverting changes also happens when breakage is detected, conditions for which are covered in Section A.4.6.

Logging in

Finally, we also must make sure that the web application itself is in a proper state to be scanned. Among other things, this means making sure that Spider-Scents is logged in. We automatically grab relevant details such as cookies, user agent, and the user URL (dashboard) at the start of each scan. Spider-Scents uses Selenium to interact with a headless Chrome instance with a custom extension to record this information, after having been supplied with the necessary credentials (username, password, and login page). These client details, which web applications often use to identify users, are then re-used throughout the rest of the scan.

Algorithm 2 Discovering sensitive rows.

```
$i \leftarrow 1$
**while** $i \leq rows$ **do**
    $row \leftarrow retrieve(i)$
    $error \leftarrow delete(i)$
    **if** $error$ **then**
        $sensitive(row)$
    **end if**
    $crawl()$
    $error \leftarrow insert(row)$
    **if** $error$ **then**
        $sensitive(row)$
    **end if**
    $i \leftarrow i+1$
**end while**
```

A.4.4 Choosing a database cell

For each cell in the database, Spider-Scents checks if the cell is suitable for our XSS payload. We ensure that the schema for the cell's column specifies that it is a text field with a length that can accommodate our payload, described in Section A.4.5.

Avoiding sensitive rows

We also must consider if the web application is particularly *sensitive* to changing a specific cell's value. If the application tries to revert an injected database value, reset a table or row, or even insert a conflicting row, various problems can occur. This can either be due to the web application not having a robust recovery method that can handle Spider-Scents's admittedly unexpected meddling in their database tables, or due to Spider-Scents losing track of the state the database is in. In either case, we avoid such sensitive cells by first probing the database for their existence.

For each table in the database, we discover sensitive rows as shown in Algorithm 2, where *rows* is the number of rows in the table, $retrieve(i)$ and $delete(i)$ perform appropriate actions on the i^{th} row in the table, $insert(row)$ inserts the row into the table, $sensitive(row)$ marks that this row in the table is sensitive to changes, and $crawl()$ crawls the application to induce reverts, conflicts, or resets in the database.

Choosing a cell

How a table in a database is used by an application also matters, and, unfortunately, this usage is not always fully specified in the database schema.

If a table has uniform types of rows, then it might be sufficient to modify cells in a single row. However, this assumption only holds if, for instance, the access control policies of the application specify that all users can see the data in all rows.

This also assumes that each entry in the table is handled identically by the web application code, which might not be a correct assumption. Indeed, not all tables have only such uniformly typed entries; a counterpoint is tables that store key-value data, where the interpretation of a value cell depends on a key cell identifying it. Given the existence of such key-value tables and the fact that reflection code for entries can change based on the contents of a particular entry, we must iterate across all rows in a table.

In addition, the order of cell changes does impact which XSS-vulnerable workflows are discovered. We implement different traversal orders, primarily based on iterating tables in alphabetical order. From there, the scanner either chooses cells (that satisfy payload requirements and are not sensitive) based on the iteration of rows or columns.

A.4.5 Payload insertion

In contrast to traditional black-box approaches, Spider-Scents does not interact with the web application in an attempt to insert data from user input to a particular database cell. Instead, Spider-Scents inserts the payload directly into a database cell (step ③).

Once a suitable cell has been found, Spider-Scents generates a unique ID to use for the payload. The payload we use is inspired by payloads used in prior work [11]. This structure helps reduce false positives with dynamic XSS detection. We use the following template, where `ID` is replaced with a unique generated ID:

```
"'><script>xss(ID)</script>
```

While other payloads, such as a general-purpose `img` payload, can be more or less suitable depending on the context of the reflection, we consider the problem of creating smaller or context-specific XSS payloads to be orthogonal to this work. Using this payload can lead to false negatives, for example, if the reflection happens in a `textarea` or `title` tag.

Structured data

This is with the caveat that we *do consider* the presence of *structured data* in a cell. Spider-Scents' iterative modification of database cells, and searching for these individual reflections in the web page, has the implicit assumption that it is *individual* data stored in database cells in a one-to-one correspondence from web input to database cell. This might not be true—data might be split up into multiple cells or combined within a single cell. A payload would need to be either split across cells in the first case or multiple payloads combined in the second. Correlating changes across cells is considered out of scope for this work.

However, when data is combined within a single cell, we consider this as structured data. Spider-Scents modifies payloads to fit some common types of structured data. If a known structure is detected by parsing cell data, the payload will be delivered to each valid location for it. Currently, Spider-Scents implements a customized payload for PHP serialized objects and image paths. The support for these formats is chosen to demonstrate this approach and can lead to false negatives due to not handling more widely-used structured data formats, such as JSON.

A.4.6 Application breakage

After inserting a payload, the changing database values can have catastrophic effects on the web application's functionality, which we call *breakage*.

While breakage can happen when using an application through its standard functionality, it is even more likely when Spider-Scents directly modifies the database and possibly inserts values that the application has not defensively coded against.

From a black-box scanner's perspective, such internal application-specification breaking changes are not available. However, similar behavior, when reachable on standard interfaces, is regarded as *irrecoverable state changes* [7]. While it may be intended functionality, such an irrecoverable state change overlaps with our notion of application breakage.

Guarding against breakage is inherently a tradeoff. On the one hand, changing the website's domain to an XSS payload in a CMS such as WordPress will rewrite all links, including admin ones, making the application unusable. On the other hand, new vulnerable behavior of the application might have been discovered instead.

Entirely black-box approaches will not be able to recover from such a breaking, irrecoverable state change. Some scanners add infrastructure access (shown in Figure 1) to be able to reset the application when this happens [7,10].

Algorithm 3 Determining breakage.

```
$i \leftarrow 1$
$broken \leftarrow 0$
while $i \leq length(urls)$ do
    $current \leftarrow request(urls[i])$
    $broken\_url \leftarrow request(urls[i])$
    if $broken\_url$ then
        $broken \leftarrow broken + 1$
    end if
    $i \leftarrow i + 1$
end while
if $broken/length(urls) \geq threshold$ then return $broken$
end if
```

With our position in the database, while we are more prone to cause breakage, we can also *identify and fix it*, without any additional access to the application.

Therefore, as noted by step ④ in Figure 2, Spider-Scents will dynamically scan the application looking for signs of breakage. If any are found we can reset the exact cell we changed that caused the breakage, even in the case when the web application is unusable.

Web application breakage across a web application is inferred by Algorithm 3, where *urls* is a list of URLs for distinct web pages in the application, $compare(baseline, current)$ compares the current status of a web page to a baseline measurement, and *threshold* is a threshold specified for how many pages can be acceptably broken in a web application.

$compare(baseline, current)$ can be implemented to compare measurements of a web page response based on HTTP status codes, linked content, length, or other heuristics. The approach we take is a combination of status codes and linked content.

A.4.7 Reflection scanning

Finally, we dynamically exercise the application with a reflection scanner (step ⑤). This scanner will crawl the application and report back on all the IDs that it finds. Here we differentiate between reflected JavaScript payloads that are executed, i.e. unescaped, and cases where we find the IDs in text. We do not try to find mangled or encoded payloads.

Spider-Scents uses *Black Widow* [11], the source code of which is available[3], with minor modifications to facilitate communication between modules,

[3]https://github.com/SecuringWeb/BlackWidow

as its reflection scanner.

A.4.8 Manual analysis

From the Spider-Scents reports we manually analyze the unprotected outputs to determine if the payloads could be added from the web application. Once an input element is found we supply valid data and ensure the database is updated accordingly. Next, we add our payload and record if it is (1) rejected due to validation, (2) escaped, or (3) sanitized. We then repeat this for all inputs relating to the column.

A.5 Evaluation

We evaluate our approach by analyzing 12 different web applications and report on the number of stored XSS vulnerabilities found.

We compare Spider-Scents with a combination of up-to-date academic and open-source scanners that find stored XSS in Arachni [2], Black Widow [11], and OWASP ZAP [54].

A.5.1 Web applications

Similar to previous works [7, 11, 33, 49] we test both old applications and new modern ones. We divide the target applications into two sets. The five reference applications (that have known CVEs) and are used in prior work are: SCARF [45], Hospital Management System [35], User Registration & Login and User Management System [36], Doctor Appointment Management System [37], and Hostel Management System [38].

For modern, complex, applications we use these seven: CMS Made Simple [46], Joomla [21], MyBB [27], OpenCart [29], Piwigo [39], PrestaShop [40], and WordPress [53], Statistics that describe the applications chosen, their version numbers, and their usage in evaluation by prior work is provided in the Appendix Table 4.

Applications are largely chosen based on those evaluated by the authors of jäk [33], Black Widow [11], and Witcher [49][4], as these represent the current state-of-the-art in academic black-box web scanners. We restrict the evaluation to those that are database-backed.

[4]Witcher only supports detecting SQL Injection and Command Injection, therefore we do not compare against it.

Note that while we have selected applications based on the prior criteria, we choose the *latest version* of each application. Therefore, the unique vulnerabilities found by Spider-Scents are also *new*.

A.5.2 Experimental setup

In this section, we present the experimental setup used for the evaluation of Spider-Scents and comparison with other scanners.

Performance metrics

We focus our evaluation on three metrics: database coverage, vulnerabilities, and exploitability.

Database coverage. To successfully execute a stored XSS payload the scanner must first write the payload to the database [5]. By comparing a snapshot of the database before and after each scan we can approximately measure what effect each scanner has on the database.

These snapshots allow us to more precisely determine where the scanner fails in storing a payload, and where Spider-Scents can benefit from directly adding the payload to the database. In addition, we also classify the changes as either benign or XSS payloads. If an XSS payload is added, we investigate if the scanner can find it.

Vulnerabilities. To evaluate our method's capability to find vulnerabilities we also record the number of vulnerabilities reported by each scanner.

There is no clear method, neither in literature nor suggested by the scanners' implementations, of how to differentiate between two different XSS vulnerabilities. This means that for a given web application functionality, different scanners can generate different amounts of reported XSS vulnerabilities. For example, a vulnerable search bar included on every URL could generate a reported XSS vulnerability for each URL. To level the playing field and allow for a fair comparison we manually inspect each vulnerability and cluster them based on their related functionality. This clustering is justified as an application of *root cause analysis*, a process already well-established and valued in software bug reports [16].

Furthermore, we only compare stored XSS results from other scanners. Reports from compared scanners are manually confirmed to either be stored XSS or non-stored XSS (reflected or DOM). During evaluation, Arachni finds

[5]Assuming the payload is stored in the database; see Section A.6.3 for an example of a stored XSS in the filesystem.

4, Black Widow finds 2, and ZAP finds 4 [6] non-stored XSS. Our method is unable to find these, as it is specific to stored XSS.

Exploitability. Scanners search for XSS vulnerabilities by injecting data with XSS payloads into the application. Afterwards, they search for this data, either statically, or dynamically. However, due to permissions, this does not guarantee that an attacker can abuse the discovered XSS. For example, if only the super user can inject the payload, the vulnerability is not exploitable.

As modern applications can have complex user and group permissions with different associated application views, it is difficult for a scanner to automatically reason about the *exploitability* of these possible injections. In this paper, we manually verify and report on the exploitability of each reported vulnerability. We divide this step into two parts, i) input protections and ii) permissions.

For input protection, we ensure that it is possible to add the XSS payload from the application to the database. If it is not possible (protected), we further categorize the input protection for the reported vulnerability.

No Input, when there is no usable input field allowing for writing to the database. For example, input fields that are only available during installation or constants, such as US states.

Escaping, when the application changes the user input, to prevent it from being interpreted in some context, before adding it to the database. E.g. transforming < to `&lt;` makes the symbol safe to be included in HTML context.

Stripping, when some data is removed ("stripped") from the input. E.g. removing `<script>` from the user input.

Validation, when the application refuses to add the user input to the database if it does not satisfy some format, such as containing illegal characters.

For permissions, we consider the vulnerability exploitable if a less privileged user can add a payload that is executed on a page that a user with more privileges can access. For example, if a normal user can book an appointment whose XSS payload is executed in the admin dashboard, then we would consider this exploitable. However, if only the admin could add the payload to such an appointment then it would be equivalent to self-XSS.

[6] In general, non-stored-XSS found by scanners is the difference between columns *R* and *S* in Table 1. However, due to the false positive reports by ZAP in the Hospital Management System (see Section A.6.2), ZAP only finds 4 non-stored-XSS in the Doctor Apt. and Hostel applications.

Scanner configuration

We configure other scanners to make as fair a comparison as possible. While we focus on stored XSS, web scanners can scan for a plethora of other vulnerabilities, including SQL injection, command injection, and local file inclusion.

For this evaluation, we configure each scanner to only focus on finding XSS vulnerabilities. Furthermore, to allow scanners a better chance to authenticate and stay authenticated, we make slight modifications to the *web application*. First, we ensure the index page has a link to the admin login. Secondly, we rewrite the POST parameters server-side to match the correct user. This will level the playing field, as scanners prefer different authentication methods. Other scanners are configured to limit their runtime to 8 hours, similar to prior work [11,49]

Configuration of parameters specific to Spider-Scents can be found in Appendix A.1.1.

A.5.3 Comparison results

In addition to the most direct comparison statistic—vulnerabilities found—we also collect a new statistic for this problem: *database coverage*. This is motivated by the different approach taken by Spider-Scents.

Database coverage. Modifying data in the database is required to detect stored XSS in database-backed applications. As such, we record the number of unique columns in the database each scanner modifies. In the case where an entire row is added, we give the scanner credit for all columns in the table. In our analysis in Section A.6.4, we look more closely at the data inserted by the scanners.

In Appendix Table 5, we present the database coverage of each scanner and compare them to Spider-Scents. We further visualize this in Figure 5. As is evident, Spider-Scents can affect a much greater portion of the database compared to the other scanners. We cover between 79% to 100% while the other scanners cover between 2% and 60% on average. This shows that black-box scanners are still limited in how much they can affect the database, and subsequently, how well they can detect stored XSS.

There are cases where other scanners affect columns that Spider-Scents does not modify: For example, on CMSMS, both Arachni and ZAP affect columns that Spider-Scents does not. In this particular case, it is the `cms_adminlog.username` and the `cms_users.username` columns. Both these have a max length of 25 while our payload is 30 characters. We discuss these cases in more detail in Section A.6.1.

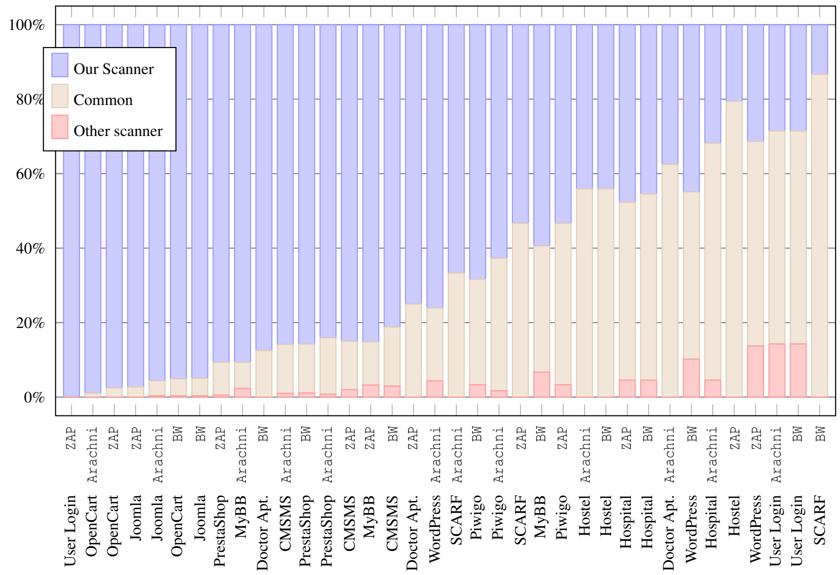


Figure 5: For every bar we present the fraction of database columns affected. First, on top, the fraction of columns only Spider-Scents finds, the middle shows the fraction of columns both scanners find, and finally, on the bottom, the fraction of columns only the other scanner finds.

Database application mappings. Our approach generates a mapping from database tables and columns where a payload is inserted, to the URLs where the payload is found. In Figure 6 we show such a mapping for the application Piwigo, where the red lines indicate unprotected output and the black lines indicate protected output.

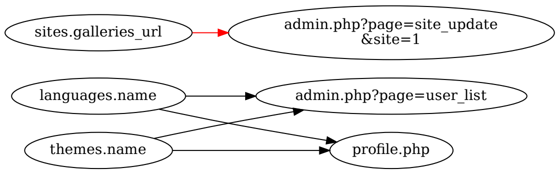


Figure 6: Subset of results from scanning the Piwigo web application. Black lines indicate protected outputs while red lines indicate unprotected outputs.

XSS results. In this section, we compare the reported XSS vulnerabilities by each scanner. In Table 1, we present reports by each scanner in column *R*, manually confirmed stored XSS in column *S*, and manually verified and de-duplicated reported vulnerabilities in column *V*. Note that anything we find in column *V*, all black-box scanners should report as well.

Reports by other scanners are of XSS vulnerabilities. However, the Spider-Scents scanner reports unprotected reflections, not XSS vulnerabilities (see Section A.4.1). Therefore, column *S* is inapplicable, and undefined for Spider-Scents.

Overall, our approach finds 85 stored XSS vulnerabilities that other scanners should be able to find, compared to the 15, on average, that they do find. 53 of these vulnerabilities found by our approach are unique and new.

With the exception of the Piwigo vulnerability Black Widow finds, which we discuss in Section A.6.3, we find all stored XSS the other scanners find.

Notably, classic black-box scanners still struggle to find stored XSS, as indicated by the relatively low numbers in Table 1. There are some notable outliers, such as ZAP reporting 26 XSS on the Hospital Management System. However, as later clarified by manual analysis, this is a single stored payload being mislabeled as multiple XSS vulnerabilities.

Table 1: XSS vulnerabilities reported (and manually verified) by each scanner. *R* - All XSS or unprotected outputs reported by the scanner, *S* - Confirmed stored XSS, *V* - Verified and de-duplicated with our unique finds in parentheses.

Scanner	Arachni			Black Widow			ZAP			Spider-Scents		
	R	S	V	R	S	V	R	S	V	R	S	V
CMSMS	0	0	0	0	0	0	0	0	0	18	-	8 (8)
Doctor Apt.	5	2	2	1	0	0	4	1	1	8	-	4 (2)
Hospital	5	4	4	4	4	4	26	1	1	33	-	30 (22)
Hostel	13	13	13	3	3	3	0	0	0	23	-	19 (6)
Joomla	0	0	0	0	0	0	1	0	0	9	-	0
MyBB	0	0	0	0	0	0	0	0	0	6	-	6 (6)
OpenCart	0	0	0	0	0	0	0	0	0	6	-	0
Piwigo	0	0	0	1	1	1	0	0	0	5	-	1 (1)
PrestaShop	0	0	0	0	0	0	0	0	0	3	-	0
SCARF	0	0	0	10	9	9	0	0	0	12	-	11 (2)
User Login	0	0	0	0	0	0	0	0	0	3	-	3 (3)
WordPress	0	0	0	0	0	0	0	0	0	7	-	3 (3)
Total	23	19	19	19	17	17	31	2	2	133	-	85 (53)

Exploitability. While scanners report on user input being executed as JavaScript, they fall short of understanding the *exploitability* of the vulnerability. In this section, we break down the vulnerabilities we find with Spider-Scents into unprotected output, unprotected inputs, and unprotected permissions, defined in Section A.2. We define the unprotected input as an input field where it is possible to add a payload without it being escaped, stripped, or subject to validation, as described in Section A.5.2.

In Table 2 we present the results from our approach and exploitability analysis. Interestingly, we note that there is a diverse mix of input protection methods, even within one application. For example, in CMS Made Simple, escaping is used for the user's first and last name, while stripping is used for the email, and validation is used for the content alias. Nevertheless, the application still failed to properly sanitize its output.

Moreover, complex and dynamic user roles in modern applications make it difficult to automatically reason about the impact of XSS. For example, in CMSMS there is a binary option for permission to modify bookmarks, that can be assigned to any user group. Any user with this permission can abuse an XSS to gain privileges. In contrast, MyBB has a strict separation of admin configurations and forum moderation configurations. This means that any XSS a scanner finds, including ours, while authenticated as an admin in MyBB, could be regarded as self-XSS as only trusted parties control the input. Similarly in Piwigo, the page title is vulnerable to XSS, however, only the admin can change it. As we see in Table 2, while both MyBB and CMSMS fail to escape

Table 2: Exploitability of reported vulnerabilities. *T* - Total reflections, *NI* - No Input, *VA* - Validation, *ST* - Stripping, *ESC* - Escaping, *P* - Permission, *EXP* - Exploitable. * A CSRF vulnerability could be abused to exploit it. ** Poor authentication validation allows privilege escalation.

	T	NI	VA	ST	ESC	P	EXP
CMSMS	18	3	1	3	3	0	8
Doctor Apt.	8	4	0	0	0	2*	2
Hospital	33	3	0	0	0	6*	24
Hostel	23	4	0	0	0	4**	15
Joomla	9	5	0	4	0	0	0
MyBB	6	0	0	0	0	6	0
OpenCart	6	0	0	0	6	0	0
Piwigo	5	1	3	0	0	1	0
PrestaShop	3	0	3	0	0	0	0
SCARF	12	1	0	0	0	7*	4
User Login	3	0	0	0	0	0	3
WordPress	7	0	0	1	3	0	3

all outputs, MyBB is less exploitable due to its stricter permissions.

In the Doctor Appointment Management System, neither output protection nor input protection is used. Despite this, two vulnerabilities are not exploitable because of permissions. Specifically, while users can change their own email address, only doctors (super users in this context) can change a doctor's name. Still, these are not fully protected, as the application with its default settings is also vulnerable to CSRF attacks. These vulnerabilities can be combined to exploit the XSS. Therefore, we believe it is useful for developers to learn where unprotected outputs are so that they can be fixed, even if they are protected by permissions.

A.6 Analysis

In this section, we investigate our results and highlight limitations of both black-box scanners and our approach.

A.6.1 Database coverage

While our method generally achieves higher database coverage, there are some interesting cases where other scanners still perform better in this metric.

In our evaluation, the portion of the database we miss and other scanners can reach is a result of our payload's length. Our payload length is always at least 30 characters long, making it too big for some cells. For example, on Piwigo Black Widow can affect the `oc_customer_ip.country` column, which only holds two characters. ZAP also modifies `mybb_templates.version`, with a size of 20 characters. This could be enough for some XSS payloads.

In theory, we can miss finding possible vulnerabilities if changing a non-text (numeric or date) value is necessary to trigger a vulnerability. Foreign key constraints can also cause problems but are less common in the text fields we focus on.

A.6.2 False positives

Black-box scanners use a variety of methods to detect injected XSS payloads, which can result in false positives. ZAP, for example, incorrectly identified XSS in WordPress. It statically found the injected token `;alert(1);` in a JavaScript context. However, the token was inside a string, which in this case it is not possible to break out of.

Confusing multiple payloads is another problem many scanners face. In the Hospital Management System, ZAP can successfully inject an XSS payload into the database. However, it does not detect this as a stored XSS, and is confused when it later scans for DOM-based XSS with the same payload, `alert(5397)`, which ZAP does find. This is caused in part by the number 5397 not being random but a constant, defined in the code as `UNLIKELY_INT`. Therefore, for a single stored XSS vulnerability, ZAP instead reports 26 DOM-based XSS. In this case, mistaking DOM XSS for stored XSS can impact developers who are unable to reproduce the results when the database is reset.

Our approach can avoid many of these problems by using unique IDs for each cell in the database and dynamically testing that each payload is executed. As such, similarly to Black Widow, we have a low rate of this type of false positive.

In contrast to black-box scanners, our approach does not automatically verify that an unprotected input exists. Therefore, we might report a database cell that cannot be changed by the web application. For example, in the Hostel Management System, the list of US states were not escaped on output, but were all hard-coded. We argue that when more functionality is added, either through software updates or third-party code such as plugins, it can introduce a vulnerability, and as such these reports are important.

A.6.3 What we miss

In the complex setting of web applications, there might always be more unknown vulnerabilities. In the absence of ground truth, in line with previous work [9–11, 33, 47, 49], our false negative comparison baseline is the stored XSS results of other scanners.

In Piwigo our method can find a reflected value, but not XSS, for a value in the configuration table. However, upon further manual analysis, we note that the reason we did not find the XSS vulnerability was because of the payload chosen. In this case, the value was reflected inside a textarea, meaning a `</textarea>` tag was needed to break out and execute JavaScript.

Black Widow found one XSS on Piwigo that we missed. While we were able to add the payload and find the correct URL, the URL was too late in the reflection scanner's queue and was therefore never visited. Increasing the reflection scanner's timeout would solve this, at the cost of runtime performance.

We have only implemented a prototype that demonstrates the utility of our approach. As with all scanners, the choices taken in that implementation can lead to false negatives. False negatives can stem from missing SQL analysis that can limit our interaction with the database, such as foreign keys and other constraints, and triggers. Better authentication, and re-authentication, mechanisms would also improve our approach. Replaceable components to our method, such as the reflection-crawling and payload-selection modules, are shown to be the cause for some of the false negatives in this section.

In addition to our evaluated comparison with the other scanners, we also survey vulnerabilities from CVEs and previous academic papers to construct a dataset of known vulnerabilities in our choice of evaluated web applications. In this dataset, 33 previous reports covered 29 unique stored XSS vulnerabilities, of which we find 26 (details in Table 6). In WordPress, we miss a vulnerability that relies on an attacker-controlled server that returns a crafted message to a link embedded in a post. For all evaluated methods, including ours, this type of attack is out-of-scope. However, it should be noted that we do correctly mark the database column as unprotected output. In CMSMS, we miss a stored XSS vulnerability in the filename of uploaded files. Here the payload is not stored in the database but rather in the filesystem. Extending our method from databases to other storage mechanisms could be an avenue for future work. Finally, in OpenCart, we miss a stored XSS in the category description because we do not spend enough time crawling the application after database modifications, to scan for reflections of the inserted payload. After extending the reflection scanner's timeout, Spider-Scents was able to find this vulnerability as well.

A.6.4 What others miss

From the scanning results of the Doctor Appointment Management System, we can see that ZAP fails to detect multiple XSS vulnerabilities. By analyzing database snapshots before and after execution we note that ZAP was only able to insert data into the `tbldoctor` table. While it was able to add the string “ZAP” to the `FullName` column, it could not add an XSS payload. Furthermore, ZAP misses other tables, such as `tblpage` and `tblappointment`, that our method modifies and detects as unprotected output.

We also note cases where the compared scanners fail to find XSS due to a lack of database coverage. For example, in MyBB, no other scanner affects the vulnerable `mybb_usergroups.namestyle` column.

A.6.5 Exploitability

Both black-box scanners and our method will report on injected JavaScript being reflected and executed. However, as we see in Table 2, not all these executions could, in their current form, be exploitable. Interestingly, we note that web applications are relatively equally split on using sanitization and escaping on user input. Validation, on the other hand, is less common. Moreover, permissions also play an important role in protecting these *XSS vulnerabilities* from becoming exploitable.

A.6.6 Drop-in testing with Spider-Scents

In our evaluation, we assisted other scanners by modifying the web applications under test, so they could evade typical login checks. While beneficial for increasing their performance for the sake of comparison, such modifications are not ideal.

To demonstrate the applicability of Spider-Scents, we *do not modify* web applications when we run Spider-Scents. Therefore, we rely entirely on our breakage heuristics, automatic reverting, and automatic use of captured log-in details.

A.6.7 Manual analysis with Spider-Scents

Spider-Scents requires more manual effort to verify a vulnerability compared to a black-box solution. However, as Spider-Scents reports the corresponding database table and column, e.g. `users.email`, it is usually relatively easy to manually find relevant input fields and test for a working XSS payload, as described in Section A.4.8. In our evaluation, it took an author approximately

Table 3: Runtime performance of Spider-Scents. Runtime is proportional to the size of the database of the application, reported in both the raw number of database cells and those that Spider-Scents can scan (satisfy payload requirements).

	Scan time	Database cells	Scannable
CMSMS	7:14	11844	4339
Doctor Apt.	0:08	195	87
Hospital	0:22	282	106
Hostel	0:13	205	90
Joomla	12:59	11584	4813
MyBB	4:21	15701	5321
OpenCart	1:39	31490	11553
Piwigo	1:07	1826	520
PrestaShop	32:29	44529	10745
SCARF	0:06	36	15
User Login	0:01	10	6
WordPress	1:55	385	868

15 minutes per report, on average. Preparing Spider-Scents to scan takes a similar time to other scanners, with the small addition of database credentials.

Further automation to ease analysis is possible, such as mapping input fields to the database, although this will require addressing general challenges of crawling, such as exploration and input validation [11].

A.6.8 Runtime performance of Spider-Scents

In contrast to other black-box scanners, which can run indefinitely [11], our method runs for a time proportional to the database of the application. In our evaluation, other black-box scanners are limited to a runtime of 8 hours. Spider-Scents almost always completed its scans within this time window, with the exception of modern applications Joomla and PrestaShop. Reference applications are scanned within minutes, while modern applications are scanned in hours.

We report the scan time of Spider-Scents in evaluation in Table 3. These times are collected on a laptop from 2021 with 8 cores and 16 gigabytes of RAM, running both the application's web server and the Spider-Scents scanner.

A.6.9 Coordinated disclosure

We have reported all new vulnerabilities to the affected vendors and will summarize their responses here. MyBB is planning to fix the vulnerabilities we reported in the upcoming 1.9 version. The CMSMS developers argue that any authenticated XSS (regardless of the specific user/group permissions) is not considered a vulnerability. Instead, they will revise their documentation to no longer motivate their permission model as a “security mechanism”. WordPress, on the other hand, does consider some authenticated XSS as vulnerabilities, depending on permission. However, their security model differs from that evaluated. In our model, we considered any privilege escalation as a vulnerability, while WordPress developers consider *editor* and *admin* to be equivalent. As such, there does not seem to be a consensus among web developers as to how application permissions should be modelled. We are still waiting for a response from PHPGurukul for vulnerabilities in their multiple applications. However, these vulnerabilities have a clearer precedence with similar vulnerabilities to ours, e.g. *CVE-2023-27225*.

A.6.10 Summary

As the results show, Spider-Scents performs both better in database coverage and stored XSS vulnerability detection when compared to state-of-the-art scanners. Based on what vulnerabilities the other scanners miss and what we uniquely find, we believe the reason for this improved performance is because we bypass the majority of the roadblocks that current XSS scanners face (as defined in Section A.3). Solving these challenges directly is a substantially harder problem [8], and will require solving fundamental challenges with crawling [11]. In many instances, the other scanners fail to get any data into the vulnerable database column for vulnerabilities only Spider-Scents finds, and in other cases when they do, only benign data is added. In general, the main problem current scanners face, which we bypass, is getting the payload into the database.

A.7 Related Work

A.7.1 Black-box scanners

Enemy of the State [7] models server-side state in different links and requests are identified that drive such state changes. Notably, this work recognizes the necessity of a solution to resetting a web application. In this case, the application is run in a VM, and the machine is reset to counteract irreversible state changes. Spider-Scents does not need a VM, and resets can be done in a granular and inexpensive fashion. Furthermore, Enemy of the State's access to the VM subsumes this paper's access to the database.

LigRE [9] and KameleonFuzz [10] also focus on server-side state. LigRE improves XSS detection with taint flow inference, and KameleonFuzz adds genetic algorithms for payload generation and modification. Similar to Enemy of the State, these approaches require the ability to reset the web application.

jäk [33] instead focuses on modelling client-side state. JavaScript APIs are hooked to be able to model dynamic behaviour. The crawler generates a navigation graph including this information.

CrawlJax [25] also models client-side state. Interactable candidate elements, such as clickable ones, are interacted with to extend the crawler's reach. A state-flow graph models the user interface.

Black Widow [11] identifies key fundamental challenges for black-box scanning. They mitigate them by combining navigation graphs, workflows, and inter-state dependencies in one XSS scanner. In contrast to prior work, Black Widow does not assume the ability to reset the web application.

A.7.2 White-box scanners

Saner [4] focuses on identifying improper sanitization to find vulnerabilities such as XSS and SQLi. Saner is limited to analyzing PHP, and even more to custom sanitization routines. To reduce complexities with application state – such as the database – Saner does not interact with a live instance of the web application, instead choosing to build a model of the sanitization process from static analysis results.

Restler [3] does not use the entire application's codebase, but instead only the REST API specification. Static analysis of this specification identifies inter-request dependencies to generate tests, which generate dynamic feedback execution to guide further testing. Similar to Spider-Scents in both analyzing a different artifact/interface than typical static or 'grey-box' analyses, Restler also focuses on bugs. Indeed, they note that vulnerabilities in a REST specification are unclear.

Sentinel [24] seeks to limit access to sensitive data in the database to SQLi attacks. The authors model web applications to identify invariants for the 'normal' functionality, which they use to examine queries and responses to block malicious SQL usage.

A.7.3 Grey-box scanners

Most prior grey-box approaches inform a white-box scan with some runtime information from a black-box scan, to reduce the false positive rate and generate a full exploit proof. We argue that only needing database access is more general than source code [7]. It is easier to apply our approach to a different web application. Being almost black-box, we are agnostic to the coding language and framework for the application's implementation. White-box approaches might not be able to handle obfuscated code. Obfuscated code can also be present due to extensions, such as plugins. We also do not replace the database or insert some proxy between the database and the application. This makes it easy to adapt our approach to other storage mechanisms.

webfuzz [50] instruments code for coverage, and uses this feedback to fuzz requests for detecting reflected and stored XSS. This approach is expensive - WordPress reaches 27% coverage in 2000 minutes.

Witcher [49] identifies issues with using grey-box coverage to guide a web application fuzzer for vulnerability discovery. A Fault Escalator is defined to detect when the application is in some vulnerable state, and guide the fuzzer to escalate that to a vulnerability. Together with a refined notion of coverage, Witcher can fuzz URLs to find command and SQL injection. This approach is limited to first-order reflected vulnerabilities and does not model application state.

Gelato [18] detects reflected and DOM-based XSS. Taint analysis is used to target exploration of the large state space of modern JavaScript.

Backrest [14] statically infers a model REST API, then uses coverage and taint feedback to drive fuzzing of requests for detecting SQLi, XSS, and command injection. While the motivations are to both improve coverage and runtime, the runtime improvements are more evident. Notably, XSS detection, especially stored XSS, is reduced when provided with feedback. The authors point out the problem of following taint across the interface with storage in a database.

Chainsaw [1] implements automated exploit generation, where potential vulnerabilities derived from static analysis are dynamically tested, with successful executions being concrete exploits. Symbolic execution of PHP is used to find sources and sinks, as well as sanitizations/transformations along paths. The database is regarded as an additional input to the application, with the database schema consumed. Workflow-based vulnerabilities, such as stored XSS, are found within a comparable 600 minutes.

[7] Static analysis can also be performed on compiled binaries or intermediate representations. However, the same arguments against generality apply to those other artifacts.

A.7.4 Database-aware grey-box web scanning

Similarly to Spider-Scents, Steinhauser and Tůma utilize a grey-box approach for detecting context-sensitive XSS using the database alongside a normal black-box scanner [47]. They deal with context-sensitivity in line with Context-Auditor [23].

Steinhauser and Tůma’ s grey-box approach intercepts database and web application communication, injecting non-XSS payloads into the application by replacing data coming from the database.

XSS flaws are detected by black-box parsing of HTML responses from the application matching portions of payloads in responses, to detect the payload if the application applies some common encodings. The parser continues with the possible XSS flaw, and payloads are iteratively modified to avoid context encoding. Some automatic reports of XSS flaws must then be manually analyzed to identify vulnerabilities

This approach is substantially different from that of Spider-Scents. We achieve a different, more complete form of coverage, by iterating through the contents of the database, instead of only modifying values as they are retrieved by the requests from a black-box scanner. This approach ① skips missing database entries, with the database only populated by pre-provided configs or manually sampled data from public demo instances. We also scale differently; with the database modelled as *additional inputs*, this approach of extending HTTP request fields can become ② several *orders of magnitude slower* than the base black-box scanner. For efficiency, *all database injections* are combined per request, ③ which authors note increases breakage, without proposing a solution. The applications under test are also ④ substantially modified to aid the scanner. Finally, Steinhauser and Tůma’s approach is implemented by ⑤ extending MariaDB, which replaces the database in the tested web application.

In contrast, Spider-Scents ① augments the database, ② is lightweight in our evaluation, ③ identifies and fixes application breakage, ④ works without modifying web applications, and ⑤ is implemented without a heavy-weight database replacement or proxy.

In terms of results, we share Joomla and PrestaShop. Steinhauser and Tůma verified 5 and 12 vulnerabilities in Joomla and PrestaShop, with their reports resulting in all Joomla and some PrestaShop flaws being fixed. As all Joomla flaws reported were fixed, the 9 we identify either are missed by their approach or come from further development of Joomla. Unfortunately, source code artifacts for this paper are unavailable, so we cannot do a direct comparison.

A.7.5 Database synthesis

SynthDB [6] is a recent work addressing the tangential problem of preparing database-backed web applications for security testing, such as vulnerability scanning, by synthesizing a database. In contrast to our simplistic approach, with the singular goal of having some data in every table while correlating inserted fields across tables, SynthDB uses concolic execution of PHP source to collect database constraints. These constraints are solved to uncover more program paths, while not violating 'database integrity'. Similar to Spider-Scents, the performance of scanners such as Burp is improved with this white-box preprocessing step.

Our approach also uncovers a separate problem - finding a *minimal* database. In the osCommerce application, there are over 3 million cells in the base application.Our approach must have its parameters tuned to handle this volume of cells. However, we have found this application's scale abnormal by several orders of magnitude; other web applications typically only have hundreds to tens of thousands of cells.

A.8 Conclusion

Black-box vulnerability scanners are the best tools currently available for democratizing security testing—allowing web developers with no security background or knowledge to proactively find vulnerabilities in their web applications. However, the twisted designs and logic of web applications make it difficult for black-box vulnerability scanners to even inject XSS payloads into the web application. Our approach cuts this Gordian Knot of properly supplying inputs to a web application—by injecting the XSS payloads directly into the database. We believe that this approach represents a step forward in automatic stored XSS detection, and the evaluation results show that our Spider-Scents prototype surpasses state-of-the-art black-box vulnerability scanners, while our manual systematization provides the necessary contextualization of vulnerability and exploitability to these results.

Acknowledgements This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, the Swedish Foundation for Strategic Research (SSF), and the Swedish Research Council (VR).

Bibliography

[1] Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. Chainsaw: Chained automated workflow-based exploit generation. In *CCS*, 2016.

[2] Arachni. https://www.arachni-scanner.com.

[3] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *ICSE*, 2019.

[4] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *S&P*, 2008.

[5] Username Changer. https://wordpress.org/plugins/username-changer/#description.

[6] An Chen, JiHo Lee, Basanta Chaulagain, Yonghwi Kwon, and Kyu Hyung Lee. Synthdb: Synthesizing database via program analysis for security testing of web applications. *NDSS*, 2023.

[7] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *USENIX Security*, 2012.

[8] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *DIMVA*, 2010.

[9] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. Ligre: Reverse-engineering of control and data flow models for black-box xss detection. In *WCRE*, 2013.

[10] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. Kameleonfuzz: evolutionary fuzzing for black-box xss detection. In *CODASPY*, 2014.

[11] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. Black widow: Blackbox data-driven web scanning. In *S&P*, 2021.

[12] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security*, 2010.

[13] Martin Fowler. Codesmell. https://martinfowler.com/bliki/CodeSmell.html.

[14] François Gauthier, Behnaz Hassanshahi, Benjamin Selwyn-Smith, Trong Nhan Mai, Max Schlüter, and Micah Williams. Backrest: A model-based feedback-driven greybox fuzzer for web applications. *arXiv preprint arXiv:2108.08455*, 2021.

[15] Google. https://security.googleblog.com/2023/02/vulnerability-reward-program-2022-year.html.

[16] Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. "not my bug!" and other reasons for software bug report reassignments. In *CSCW*, 2011.

[17] HackerOne. https://www.hackerone.com/reports/6th-annual-hacker-powered-security-report.

[18] Behnaz Hassanshahi, Hyunjun Lee, and Paddy Krishnan. Gelato: Feedback-driven and guided security analysis of client-side web applications. In *SANER*, 2022.

[19] 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 *WWW*, 2004.

[20] A03:2021 Injection. https://owasp.org/Top10/A03_2021-Injection/.

[21] Joomla. https://www.joomla.org.

[22] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security*, 2010.

[23] Faezeh Kalantari, Mehrnoosh Zaeifi, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, and Adam Doupé. Context-auditor: Context-sensitive content injection mitigation. In *RAID*, 2022.

[24] Xiaowei Li, Wei Yan, and Yuan Xue. Sentinel: securing database from logic flaws in web applications. In *CODASPY*, 2012.

[25] Ali Mesbah, Engin Bozdag, and Arie Van Deursen. Crawling ajax by inferring user interface state changes. In *ICWE*, 2008.

[26] Meta. https://about.fb.com/news/2022/12/metas-bug-bounty-program-2022/.

[27] MyBB. https://mybb.com.

[28] CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting'). https://cwe.mitre.org/data/definitions/79.html.

[29] OpenCart. https://www.opencart.com.

[30] Oracle. https://dev.mysql.com/doc/refman/8.0/en/information-schema.html.

[31] OWASP. https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html.

[32] Muhammad Parvez, Pavol Zavarsky, and Nidal Khoury. Analysis of effectiveness of black-box web application scanners in detection of stored sql injection and stored xss vulnerabilities. In *ICITST*, 2015.

[33] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. jäk: Using dynamic analysis to crawl and test modern web applications. In *RAID*, 2015.

[34] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: fuzzing by program transformation. In *S&P*, 2018.

[35] PHPGurukul. https://phpgurukul.com/hospital-management-system-in-php/.

[36] PHPGurukul. https://phpgurukul.com/sdm_downloads/login-system/.

[37] PHPGurukul. https://phpgurukul.com/doctor-appointment-management-system-using-php-and-mysql/.

[38] PHPGurukul. https://phpgurukul.com/hostel-management-system/.

[39] Piwigo. https://piwigo.org.

[40] Prestashop. https://prestashop.com.

[41] The Menlo Report. https://www.dhs.gov/sites/default/files/publications/CSD-MenloPrinciplesCORE-20120803_1.pdf.

[42] Mike Samuel, Prateek Saxena, and Dawn Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *CCS*, 2011.

[43] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS*, 2010.

[44] Prateek Saxena, David Molnar, and Benjamin Livshits. Scriptgard: Automatic context-sensitive sanitization for large-scale legacy web applications. In *CCS*, 2011.

[45] SCARF. https://scarf.sourceforge.net.

[46] CMS Made Simple. http://www.cmsmadesimple.org.

[47] Antoní n Steinhauser and Petr Tůma. Database traffic interception for graybox detection of stored and context-sensitive XSS. *Digital Threats: Research and Practice*, 2020.

[48] OWASP Top Ten. https://owasp.org/www-project-top-ten/.

[49] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities. In *S&P*, 2022.

[50] Orpheas van Rooij, Marcos Antonios Charalambous, Demetris Kaizer, Michalis Papaevripides, and Elias Athanasopoulos. webfuzz: Grey-box fuzzing for web applications. In *ESORICS*, 2021.

[51] WordPress VIP. https://docs.wpvip.com/technical-references/security/validating-sanitizing-and-escaping/.

[52] WordPress. https://developer.wordpress.org/apis/security/escaping/.

[53] WordPress. https://wordpress.com.

[54] OWASP ZAP. https://www.zaproxy.org.

Table 4: Web applications used in the evaluation

Application	Date	Version	GitHub Stars	Lines of Code	Prior Research
CMSMS	2022	2.2.16	∼	144944	
Doctor Apt.	2023	2023/1/11	∼	65603	[49]
Hospital	2022	2022/11/8	∼	67667	[49]
Hostel	2021	2021/9/30	∼	9377	
Joomla	2023	4.2.8	4.5k	747197	[11,33,47,50]
MyBB	2023	1.8.33	932	153055	[33]
OpenCart	2023	4.0.1.1	6.8k	186101	
Piwigo	2023	13.6.0	2.6k	280906	[33]
PrestaShop	2022	1.7.8.8	7.3k	1175530	[11,47]
SCARF	2007	2007/2/27	∼	1318	[7,11,24]
User Login	2021	V3	∼	7036	[49]
WordPress	2023	6.1.1	17.6k	651599	[9–11,33,49,50]

A.1 Appendix

A.1.1 Spider-Scents configuration

We have implemented a variety of tunable parameters for configuring Spider-Scents' choice of heuristics while scanning. Some notable parameters are:

- Avoid sensitive rows or not
- Insert rows into empty tables or not
- Configure the traversal through the database (order by table, row, column, random, reverse)
- Breakage threshold and detection type (based on status codes, response length, link content)
- Enforce independence across boundaries (across table, row, column)

We evaluate our approach avoiding sensitive rows, inserting into empty tables, traversing the database by tables and then columns, with breakage sensitive to status codes and allowing up to 50% of link content to be missing, and independence enforced across bounds.

Table 5: The number of unique database columns affected by each scanner. For each column in the table, we present: database columns only covered by Spider-Scents ($A \setminus B$), columns covered by both scanners ($A \cap B$), and columns covered by the other scanner ($B \setminus A$). The very last column presents the maximum number of columns that allow arbitrary text values.

Crawler	Arachni			Black Widow			ZAP			**MAX**
	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$	
CMSMS	85	13	1	82	16	3	85	13	2	111
Doctor Apt.	6	10	0	14	2	0	12	4	0	16
Hospital	14	28	2	20	22	2	21	21	2	44
Hostel	15	19	0	15	19	0	7	27	0	36
Joomla	283	12	1	281	14	1	287	8	0	325
MyBB	194	15	5	133	76	15	184	25	7	264
OpenCart	282	3	0	272	13	1	278	7	0	326
Piwigo	37	21	1	41	17	2	32	26	2	63
PrestaShop	306	55	3	313	48	4	329	32	2	410
SCARF	10	5	0	2	13	0	8	7	0	15
User Login	2	4	1	2	4	1	6	0	0	7
WordPress	35	9	2	22	22	5	16	28	7	53

Table 6: Known stored XSS vulnerabilities from CVEs and other publications.

Application	Source	Description	We Find
CMSMS	CVE-2023-36970	File upload stored XSS	✗
Hospital	`https://github.com/Ko-kn3t/CVE-2020-25271`	`username`	✓
Hospital	`https://sisl.lab.uic.edu/projects/chess/cross-site-scripting-in-hms3`	`weight`	✓
Hospital	`https://sisl.lab.uic.edu/projects/chess/cross-site-scripting-in-hms3`	`temperature`	✓
Hospital	`https://sisl.lab.uic.edu/projects/chess/cross-site-scripting-in-hms3`	`medicalpres`	✓
Hospital	`https://sisl.lab.uic.edu/projects/chess/cross-site-scripting-in-hms3`	`BloodPressure`	✓
Hospital	`https://sisl.lab.uic.edu/projects/chess/cross-site-scripting-in-hms3`	`BloodSugar`	✓
Hospital	`https://sisl.lab.uic.edu/projects/chess/cross-site-scripting-in-hms2`	`PatientName`	✓
Hospital	`https://sisl.lab.uic.edu/projects/chess/cross-site-scripting-in-hms2`	`PatientEmail`	✓
Hospital	`https://sisl.lab.uic.edu/projects/chess/cross-site-scripting-in-hms2`	`PatientGender`	✓
Hospital	`https://sisl.lab.uic.edu/projects/chess/cross-site-scripting-in-hms2`	`PatientAdd`	✓
Hospital	`https://sisl.lab.uic.edu/projects/chess/cross-site-scripting-in-hms2`	`PatientMedhis`	✓
Hospital	`https://www.exploit-db.com/exploits/47841`	`doctorspecilization`	✓
Hostel	CVE-2020-25270	`guardianName`	✓
Hostel	CVE-2020-25270	`guardianRelation`	✓
Hostel	CVE-2020-25270	`corresAddress`	✓
Hostel	CVE-2020-25270	`corresCIty`	✓
OpenCart	`https://github.com/nipunsomani/Opencart-3.x.x-Authenticated-Stored-XSS/blob/master/README.md`	Category description	✗
SCARF	[11]	Add session	✓
SCARF	[11]	Comment	✓
SCARF	[11]	Conference name	✓
SCARF	[11]	Edit paper	✓
SCARF	[11]	Edit session	✓
SCARF	[11]	Delete comment	✓
SCARF	[11]	General options	✓
SCARF	[11]	User options	✓
User Login	CVE-2022-43097, CVE-2020-23051, CVE-2020-24723	`fname`	✓
User Login	CVE-2022-43097, CVE-2020-23051, CVE-2020-24723	`lname`	✓
WordPress	`https://research.securitum.com/xss-in-wordpress-via-open-embed-auto-discovery`	Embed in post content	✗

B

SpiderSapien: Client-Centric Crawler and Security Scanner

Eric Olsson, Benjamin Eriksson, Adam Doupé, and Andrei Sabelfeld

Manuscript

Abstract

Black-box web application crawling and scanning plays an important role for security testing of web applications. Yet state-of-the-art scanners fall short of addressing key characteristics of a modern web application: its extreme dynamism and interactivity on the client side. This paper identifies immersive interaction as a key ingredient for scanners to deeply explore modern web applications. We propose SpiderSapien, a client-centric crawler and security scanner. Driven by immersive interaction, SpiderSapien incorporates novel methods to detect interactable elements, order UI interactions, and use LLMs to solve forms. In doing so, we demonstrate how to reliably discover and test deep states of modern web applications. The evaluation of our approach shows substantial improvements in both code coverage and vulnerability detection over previous work, with an average increase in code coverage of 21.5% compared to the *union* of the other scanners and a total of 36 XSS vulnerabilities, across 6 of the 8 web applications, compared to the 4 XSS others find. In addition, a separate empirical evaluation of SpiderSapien's LLM-powered form solving capabilities on diverse *real forms* on the open web demonstrates superiority over the previous approaches in generating desired input on the client side, solving at least 23.3% more of the non-trivial forms compared.

B.1 Introduction

Black-box security scanning is an excellent fit for detecting vulnerabilities in web applications. These scanners use a black-box crawler to interact with a web application, iteratively discovering and security testing endpoints. Vulnerabilities are discovered with nothing more than a running web application and scanner, regardless of the frameworks or languages underlying the web application.

Black-box Scanners and the Web. Historically, as web applications have incorporated more functionality along with its associated complexity, security scanning techniques have been developed in short order to handle these new features. In the 2000s, "sitemap" style black-box crawlers such as Skipfish [39] and w3af [36] could reliably scan the *Web 1.0*—websites composed of primarily statically linked content. While supporting some basic interactions with inputs in request parameters and forms, these scanners mainly focused on following static links by parsing HTML.

However, by the early 2010s *Web 2.0* had evolved to include dynamic *stateful web applications* with multi-step flows that could not be modelled well by that first generation of scanners. New black-box scanning methods such as the Enemy of the State [12] responded by incorporating a more complex internal state model. The additional *client-side functionality* of the Web 2.0 applications also motivated the concurrent development of yet more black-box crawlers such as CrawlJax [26] and jÄk [29], which incorporated support for either AJAX or broader JavaScript applications.

New Scale of Complexity. In the decade since this second generation of black-box crawlers, the characterization of Web 2.0 as consisting of stateful web applications with rich client-side functionality has remained the same. However, the *scale of this complexity*, in the dynamic statefulness and interactive client-side functionality of these applications, has exceeded the ability of these prior black-box scanners to handle.

While "software is eating the world" [33], we are now in an era where *web applications eat software*. Software services that were once offered as desktop applications can now often be accessed through web interfaces, with the same functionality. For example, Microsoft and Google deliver full productivity suites online through the widely available Microsoft 365 and Google Docs services.

These web interfaces have the same core challenges for crawling, statefulness and client-side interactivity, but at a new scale due to the inherent complexity of the functionality they now include. We refer to web applications with this scale of dynamism and client-side interactivity as *modern web applications*. Scanning these modern web applications requires that the scanner either be able to reason about multi-step stateful flows, or compose complex structured inputs to backend services that might result from the composition of multiple user steps in the actual application. Without this, the *deep* behavior of these web applications stays hidden and untested for vulnerabilities.

Insufficient Current Approaches. In practice, black-box scanners do not explore a large portion of the functionality or endpoints of these web applications. Fundamentally, this is because discovering and executing a correct sequence of client-side actions is unlikely. Scanners that avoid client-side actions and instead work directly with server-side requests are also unlikely to succeed, as the problem then transforms to either handling multiple stateful requests, or generating highly structured network inputs. Therefore, these applications are unlikely to be solved by current black-box approaches.

Security scanners that could handle these modern web applications have primarily been focused on identifying error flows with either static analysis or grey-box fuzzing. Static analysis [21, 3, 16, 20] can tailor their program analysis to specific languages and frameworks such that some of these multi-step stateful flows can be identified. Grey-box fuzzing [17, 34, 18] can also incorporate additional feedback to generate complex structured inputs. Both of these approaches generally focus on the *errors* present in the application, either in a bad flow or with error signals. While this is well-motivated, as vulnerabilities do necessarily involve some error component, we argue that security scanning also needs to incorporate the *valid* flow or input, where the *intended code path* is explored so as to generate *deeper* states in the application.

Approach. We complement prior approaches by instead focusing on generating primarily valid application inputs and flows with the overarching goal of finding more XSS vulnerabilities. Our approach has the aim of generating actions in a fashion *like a human would*. Therefore, we focus on interacting with the application's client-side interface by reliably detecting interactable elements. Rather than mainly interacting with the application through low-level URLs, network requests, and event listeners, we develop a scanner focused on valid client-side *actions*.

This scanner, *SpiderSapien*, implements a black-box crawling approach that can more profitably interact with complex client-side applications by integrating *three pillars of interactivity* throughout the scanner: detecting interactable elements, ordering UI interactions, and using LLMs to solve forms.

In doing so, this approach can address the key challenge of *interaction strategy*, and navigate the client-side interface of modern web applications to explore the *deep* states that have been the elusive target of prior works [13, 14].

We note that this goal is shared with the recent black-box evolutionary EvoCrawl and LLM task-driven YuraScanner, and discuss differences in Section B.6. However, the motivating example of Section B.3, drawn from *real* evaluated applications with their non-standard elements and form validation, is a challenge best met by our approach.

Evaluation. First, we evaluate crawling performance on 8 open-source web applications compared to 4 black-box scanners. These applications represent a set of complex modern web applications that demonstrate the new functionality found by this approach. Compared to the union of all compared scanners, our new approach finds 21.5% more server-side code in the measured applications. Our approach also finds a total of 36 XSS vulnerabilities across 6 of the 8 applications, a substantial increase over the 4 found by the other scanners.

We also evaluate only the form solving ability of LLMs in an open web evaluation. Interacting with thousands of forms served on real websites from the Tranco [30] list of top-ranked web domains, we provide evidence that LLMs can reasonably generate valid input to forms. This novel evaluation setting is detailed in Section B.4.3. We evaluate using our new method with both Gemini and GPT LLMs. Among *non-trivial forms*, defined as those that *cannot be solved with no input*, we observe that our new LLM method solves between 47.1-54.2% of forms, while a baseline heuristic method the ZAP scanner uses can only solve 40.2% of forms. After accounting for form instability in this open web setting, we see a further improvement from 34.0-38.1% to 47.6-54.2%.

Contributions. We offer the following contributions:

- We develop a novel method that can drive black-box scanners to test *deeper* program states in modern web applications with valid client actions and better form inputs, which we present in Section B.3.
- We implement our method into a prototype SpiderSapien, a fully black-box crawler and XSS scanner, aided by LLMs for defined tasks.
- We evaluate our complete black-box scanning method as well as 4 others on 8 web applications in Section B.4. SpiderSapien finds 36 XSS vulnerabilities in these applications.
- We design a new open web setting to evaluate form solving methods on a diverse dataset while avoiding harm. We apply this evaluation to the LLM form solving component in isolation in Section B.4. We show a significant increase in this new method's performance solving non-trivial forms.
- We analyze our findings and share insights for future works in Section B.5.

Ethics and Open Science. We discuss our ethical considerations and compliance with open science policy in Section B.8.

We will open source our implementation upon publication.

B.2 Challenges

Exploring modern web applications from a black-box perspective remains an intricate task. We identify *interaction strategy* as a key overarching challenge and decompose it into three main subchallenges.

Interaction Strategy. Black-box crawlers must first interact with a web application to discover the endpoints or functionality through an iterative process. Stafeev and Pellegrino distinguish black-box crawlers by their algorithms for navigation strategies, used to decide what page to visit and in what order, and page similarity methods, used to identify either different or duplicate application states [31].

Rather than modifying only those components, we change *the whole crawling loop*, as our design interacts primarily with valid client-side actions, rather than underlying URLs, network requests, or JavaScript event listeners. We thereby improve a scanner's ability to interact with and find vulnerabilities deep in modern web applications. Modern web applications are both stateful server-side and client-side, and have many client-side interactions. Furthermore, these rich client-side interactions are often multi-step, with different options presented depending on a prior client-side interaction. While prior black-box scanners can theoretically discover new vulnerable application behavior despite it being locked behind many stateful interactions, this is unlikely.

We focus on improving the interaction strategy of our black-box crawler to handle modern web applications. An insight gained in this paper is that the resources typically used to determine a navigation strategy, i.e. CSS, HTML, and JS, were not being used in such a way that the behavior of the rendered DOM can be inferred and matched by the scanner. We identify three main challenges in adapting a scanner's interaction strategy—*detecting interactable elements*, *ordering interactions*, and *generating inputs for forms*.

Detecting Interactable Elements. A major source of complexity for scanners comes from the client-side JavaScript code in web applications. While the client-side code produces links, buttons, text fields, and other interactable elements that are designed for users to easily identify and interact with, this task challenges automated scanners. Part of the challenge is the lack of HTML semantic meaning applied to interactable elements. Instead of using the traditional `a` element or `button` element, which convey clickable HTML semantics, modern web applications have the ability to (and often do) use semantically meaningless (in terms of interactability) `label` elements or custom tag names, often by their usage of frameworks.

Apart from the semantic meaning of tags, JavaScript event listeners (e.g., *onclick*), can also be used to infer if an element is interactable. In the simplest case, a static element attribute sets the event listener. However, they can also be set dynamically in JavaScript, making it more difficult for a scanner to infer interactability. Furthermore, many JavaScript frameworks (e.g., jQuery) have custom event handling and can capture all events on parent elements or even the DOM body, which divorces the event listener from the relevant element. From the scanner's perspective, the individual elements will not have any event listeners, once again hiding their interactability.

Order of Interactions. In addition to correctly modeling which elements can be interacted with, as well as the type of interaction, e.g. click, type text, drag-and-drop, etc., the order of interactions also matters. For example, a page might have a button that opens a form in a modal popup. For the user it is clear that the form can now be interacted with, and not the covered elements behind the form. However, a scanner simply looking at the HTML and DOM will not recognize the difference between the now interactable form and non-interactable elements behind it, and perform actions in the wrong order. This will close the form prematurely, losing a chance to interact with it in its correct state. Alternatively, trying to interact with the elements behind the form will have no effect in their non-interactable state.

Another aspect that complicates this challenge is that a scanner could even click on hidden buttons, as JavaScript allows such behavior. However, the hidden button receiving the click will likely not have the intended effect in the application. As such, scanners not only have to find all interactable elements, but also *need to determine which are currently interactable* given the client-side state. This challenge means that scanners which support client-side actions, such as jÄk [29] and Black Widow [14], which might interact with elements in the correct order, still usually fail as they pick elements at random without prioritizing *currently* interactable elements.

Form Solving. The final challenge in achieving immersive interaction is that even after a scanner can detect all interactable elements and understand the proper order of interactions, a scanner will still encounter complex forms in a modern web application. A scanner must be able to provide valid inputs in order to reach deep program states.

Some examples of the inputs for form validation (which can be either performed partially or completely by the browser, client-side JavaScript code, or server-side code) include valid emails, numbers, URLs, and addresses. While HTML can provide semantic attributes for the browser to enforce–that could help the scanner determine input requirements—such annotations are optional. Examples include the `type` attribute on the input element and the more general regex-validation attribute `pattern`. When validation is performed client-side and these annotations supplied, black-box crawlers such as Black Ostrich [15], can often solve their validation constraints and provide a matching input.

However, forms are diverse in their structure and validation. Therefore, many forms only explain the expected type of data in descriptive text for the intended user, making it hard for scanners to infer it. A more fundamental problem is that server-side code, invisible to the scanners, might perform data validation on the input. This web application logic can enforce validations that HTML semantics does not support. For example, relations between inputs, or relations between inputs and values in a database. For user registration forms the “password” and “confirm password” combination is a common example. Moreover, there are also inputs that need to be unique across the application, such as “username” and “email”. Rather than presenting this validation as a constraining attribute of the element or in client-side JavaScript code, the only hint of such validation is in error messages.

B.3 Method

We propose SpiderSapien, a black-box crawling approach driven by immersive interaction. SpiderSapien contributes novel methods for detecting interactable elements, ordering UI interactions, and using LLMs to solve forms. In doing so, we aim to more reliably discover and test deep states of modern web applications.

B.3.1 Motivating Example

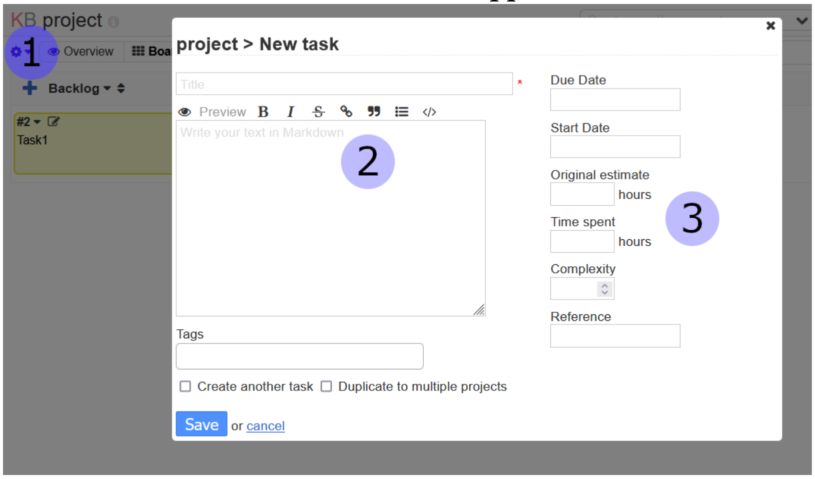
To exemplify the evolution in complexity of modern web applications we present the following example application workflow. This particular example is based on a combination of challenges seen in the popular applications Piwigo and Kanboard. In a project-management web application, users can create new tasks by clicking on a button ① to generate a modal pop-up including a form. The user can then interact with the form ②, allowing them to fill in the task details ③ and submit. From the perspective of a scanner, there are multiple challenges (as detailed in Section B.2).

Clicking on the *button* (Step ① in Figure B.1) is challenging because it is not a `button` element but instead a `label` (as is the case in Piwigo, for example), and the styling makes it visually appear to be a button. Furthermore, this `label` does not have any event listeners directly on the element, as the application relies on a global event handler instead.

After clicking, the form appears and the next challenge for the scanner is to prioritize these now visible and interactive elements in step ②. If this is not taken into account, the scanner might try to interact with the blocked elements behind the modal, which will not trigger their event listeners. Clicking outside the modal might also cause the form to disappear, resulting in future interactions with this form failing as well.

Lastly, the form itself contains further data validation challenges as the user must enter a numeric “number of hours” in step ③. This validation requirement is not in the standard HTML semantics, such as `type="number"`

Figure B.1: Web application flow with scanning challenges. The scenario is based on a combination of two applications.



or regex patterns, that would otherwise indicate that the values must be numeric.

B.3.2 Client-side Crawling

In this work we build our open-source tool SpiderSapien, implementing immersive interaction to drive the web application into deep program states by reliably discovering and handling modern web application interactions. SpiderSapien is built on top of the open-source Black Widow [14] scanner, with a radical overhaul of the client-side crawling to enable immersive interaction. This is accomplished by identifying interactable elements and prioritizing the element interactions in the scanner. We re-use the core navigation graph and XSS detection modules from Black Widow.

Interactable Element Discovery

As discussed in Section B.2, a scanner's ability to discover new web application behavior (and test that behavior) depends on it being able to identify the interactable elements on any given page. Prior scanners generally approach this task by either: (1) using predefined lists of interactable elements, usually based on the semantic HTML [2], or (2) identifying elements with associated event listeners and, possibly, hooking these event listeners. While not commonly used in security scanners, Firefox [8] uses a third option by applying static analysis of JavaScript to identify custom event listeners from libraries.

However, these approaches do not generalize, either by: (1) not including custom element types, (2) failing to properly associate an event listener to a specific element, or (3) the web application's implementation not being supported by a particular framework-specific static analysis.

In this work, we use an initial list of semantic HTML input types (`a`, `button`, `form`, and `input`), and extend this with elements that indicate their interactability at runtime. This allows our method to handle step ① of the motivating example by correctly identifying the `label` acting as a button.

We rely on the observation that application developers, JavaScript frameworks, and web browsers all aim to provide guidance to their human users, with various hints of how they can use the web applications. Providing these hints is also useful for other web application accessibility clients or extensions, such as screen readers. However, some of these interactability hints cannot be generalized. For instance, a user might infer interactability based on the color scheme. Therefore, we identify further interactable elements, regardless of HTML type or framework, by inspecting the cursor behavior of elements.

By dynamically checking the `cursor` property of an element we can draw insight into the developer's intended element semantic. Specifically, we check for the `pointer` value to determine if an element is clickable. According to the CSS specification [9], this should indicate a link. However many single-page applications and frameworks, including React, Vue, and Angular [28], use the HTML `a` tag (link) to perform client-side actions by using event listeners on the tag and using fragment URLs or the `javascript:` pseudo protocol. Similarly, we use the `text` value of the cursor to identify an element that supports text input. This is particularly useful for complex rich-text editor that rely on `div`s for user inputs. A benefit of this abstraction is that we do not need to consider individual JavaScript events like `onKeyDown`.

We also prune the list of interactable elements to prioritize active elements by inspecting the visibility of an element at runtime. Elements hidden underneath others cannot be clicked or otherwise interacted with, either from the perspective of a user's browser or Selenium automation. An alternative approach could use JavaScript to force interactions with hidden elements. However, we argue that this can lead to unintended client-side states, which we try to avoid in this work. Therefore, only those elements that are *currently visible* and *active* are included in the interactable element set.

As an optimization, we also prioritize elements that are in the current DOM (i.e., have a valid Selenium reference). This allows the scanner to achieve a deeper crawl of the client-side, as opposed to a shallow crawl across the entire application where elements from different URLs are se-

Algorithm 1 Algorithm to discover interactable elements.

```
$semanticElements \leftarrow$ <a>, <button>, <input>,...
$cursorElements \leftarrow$ pointer, text $\in element.cursor$
$elements \leftarrow semanticElements \cup cursorElements$
$interactables \leftarrow \{\}$
for $element \in elements$ do
    $\{x, y\} \leftarrow element.coordinates()$
    $topElement \leftarrow getElementBy(x, y)$
    if $element == topElement$ then
        $interactables.add(element)$
    end if
end for
```

lected. When picking an element that is not currently active, it will retrace the previous steps. Overall interactable element discovery is described by Algorithm 1.

Crawling Strategy

After discovering the valid interactable elements on a page, our scanner must order its interactions with these elements. We attempt to prioritize client-side interactions for exploration, aiming to reach deep modern web application states. However, we also need to balance client-side exploration interactions with possible exploit interactions, such as form submission and input elements. Finally, the crawler must handle classic exploration with normal static links.

Randomization is also important for coverage [31]. To accommodate this our method can, with low probability, randomly pick edges. While this can seem counterintuitive, there are cases where the scanner can get stuck continuously prioritizing a similar element without randomness.

Therefore, we design a two-phase approach to crawling. The crawler tracks actions possible from states in the web application in a directed graph, where edges represent actions, and nodes represent states. As we crawl to perform a vulnerability scan, we also prioritize inputs where a payload can be inserted. This crawling strategy allows our scanner to handle step ② of the motivating example by prioritizing the correct visible and interactive elements in the form.

In the first phase, the scanner performs a short shallow scan, prioritizing finding pages through links until a threshold of discovered links is met, or the scanner cannot find any new links. After that phase, the scanner enters the main phase where client-side interactions and payload injections are prioritized. The general prioritization follows three categories (in this order): (1) *new* actions, (2) *active* elements on the current page, and (3) *input* elements.

Because we prioritize active elements, the algorithm will act similarly to a depth-first search strategy. For example, a form (“input”) that is on the current page (“active”) and not taken before (“new”) will have a high priority.

To avoid getting stuck in one part of an application (such as an infinite calendar module [12]), we have two mechanisms to divert the scanner. First, if enough iterations have passed since new edges have appeared, an active link is chosen randomly instead. With low probability (5% in the evaluation) and if there are no new forms, we pick a random link.

Unlike scanners that separate exploration and attack, like ZAP, our method aims to always pick the best option when presented and does not need to terminate. A high-level description of the crawling strategy is presented in Appendix .1.5.

Other Client-side Adaptations

While improved interactable element discovery and crawling strategy are major components of immersive interaction, we also add a few more minor adaptations for client-side crawling. First, before submitting a form we prompt an LLM to check if the form could change the current username or password, to avoid breaking the application. We also use an LLM to determine the correct submit button in a form, as buttons can have wildly different semantics including canceling the submission. We also implement new handling for drag-and-drop. If a form supports file upload based on the `enctype`, we simulate a drag-and-drop to all elements in the form.

B.3.3 LLM-based Input Generation

As forms are one of the main ways to provide user input to a web application, determining a set of valid inputs that allow a form to be submitted is critical to a web scanner. The constraint validating the content of the numeric “number of hours” in step ③ of the motivating example is in custom JavaScript code rather than a regex pattern. Previous methods cannot expect to reliably detect a pattern while only supporting a single (or small set of) possible implementations.

Large Language Models

Therefore, we leverage the single- or few-shot ability of Large Language Models (LLMs) to provide reasonable form inputs. An LLM can reasonably infer that an input described by "number of hours", from step ③ of the motivating example, should be numeric, and in an appropriate range. We use few-shot examples [6], and error or completion feedback in our prompting of the LLM. We also use the function-calling ability of LLMs to parse its response and automate interactions.

These prompting techniques are widely used to improve the performance of LLMs across application domains. In fact, while we were inspired by the application of LLMs to vulnerability detection, the concurrently developed YuraScanner [32] has a form solving prompt resembling this method, including few-shot examples and summarizing some important elements of the form. However, among other differences (see Section B.6), SpiderSapien's recurrence is new for the form solving problem. Compared to the single-shot approach of YuraScanner, this solves forms that only present validation hints after attempted submission. This multi-shot solving approach allows our method to solve a step ⓪ *prior* to the motivating example. To create a project in Kanboard, an alphanumeric project name must be submitted. However, the name is checked server-side to ensure that it is *unique*. Otherwise, an error message presents this uniqueness requirement. Reliably solving such validation flows *requires* multiple attempts with recurrence.

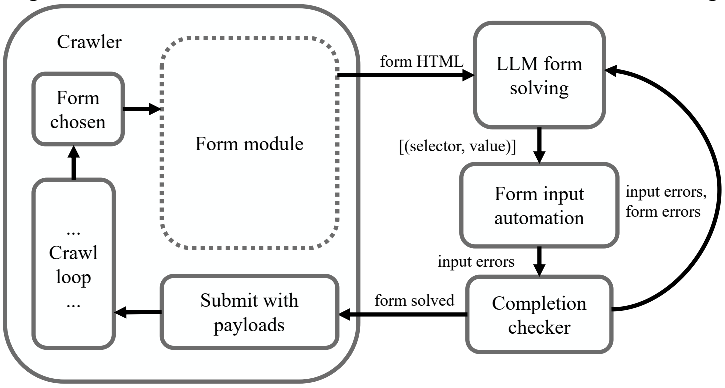
Implementation

The particular prompts chosen can be thought of as an initial prototype. Not all choices have been extensively evaluated or optimized, see discussion in Section B.5.7. Regardless, the performance of this method is already equivalent—if not exceeding that—of competing methods, as seen in Section B.4.3.

When the scanner finds a form it provides form HTML to the LLM form solving module. The LLM is prompted to solve this form, producing a list of (selector, value) pairs to indicate what values should be input to which elements. The scanner then submits the form with these values. Any errors from these input values are provided to the completion checker. If the form is judged to be incomplete, the input and overall form errors are introduced into the LLM prompt for it to provide new (selector, value) pairs. Once solved, the scanner re-submits with XSS payloads and continues scanning as depicted in Figure B.2.

The prompting of the LLM to solve a form is described by the template in Figure 6. Besides the form HTML in the prompt, we also present all input elements, those marked required, and those with regex patterns.

Figure B.2: Crawler interaction with LLM form solving



Function Calling. To describe the task of form solving and easily parse model output, we provide an API that the LLM is instructed to call. This function is supposed to be called by the LLM to input the result of a form solve, instructing the LLM to provide the structured output as a list of (CSS locator, user input) pairs. This API is fully described in Appendix .1.1.

Input Automation. The (CSS locator, user input) pairs are read from LLM output, and input is automatically entered into the form elements. This input automation includes support for clicking on checkbox and radio types, choosing appropriate values from select elements, uploading files, and typing into text fields. All automation is performed based on input values chosen by the LLM, rather than any heuristics.

Few-shot Examples. The LLM is also provided additional context about the problem domain in the form of six simple examples of forms and the type of function calls the LLM should generate to solve them.

Error and Completion Feedback

The form solving method also takes into account error feedback. This is necessary to solve validation performed server-side, either due to requirements such as uniqueness constraints, or developer choice. Therefore, the LLM is repeatedly prompted to solve a form.

While inputting generated input locator/value pairs, the crawler will log errors that occur, such as invalid element locators or invalid input values. A new message is constructed with a simple template describing these errors and prompting a retry. Additionally, the form HTML after attempted submission is provided in a separate prompt, asking “What error messages are present on this form?". Any such errors found by this sub-prompt are also included in the retry template.

This error feedback loop also requires a way to determine when not to repeat solving—i.e., when the input passes validation. If the crawler has navigated to a new page, this indicates a successful form solve and submission. Otherwise, an additional sub-prompt asks "Has this form been successfully filled? Answer 'Yes' or 'No'.". If `yes` is not included in the LLM's response, the loop will continue until a retry threshold is reached. In our evaluation, we set this threshold to 3 retries.

B.3.4 Implementation

We implement SpiderSapien as Python code extending the Black Widow scanner [14]. We implement the LLM code in LangChain to be able to easily swap the underlying model. While we have tested this LLM form solving code with versions of Google's Gemini, OpenAI's GPT, and local Ollama models, we primarily evaluate with Gemini.

B.4 Evaluation

We primarily evaluate the performance of the entire scanner, including both improved client-side crawling and LLM form solving, in the task of exploring a web application and discovering XSS vulnerabilities. This scanner evaluation is performed on a set of local open-source web applications.

We also evaluate the effectiveness of LLM form solving in isolation on the open web.

B.4.1 Web Application Crawl Evaluation Setup

The overall goal of this work is to better explore web applications from a black-box perspective, by successfully navigating client-side interactions and user input in forms. By being able to explore more states in these web applications, we also hope to find new vulnerabilities. To be able to gather relevant performance metrics and analyze vulnerability, our primary evaluation is on the crawling performance of our crawler on a set of local open-source web applications. We compare our crawler's performance to other relevant black-box methods.

Open-source Web Applications

We choose 8 open-source web applications for this evaluation: DokuWiki, TinyFileManager, Kanboard, WordPress, osCommerce, HotCrp, Leantime, and Piwigo. We describe these applications and include their versions in Appendix .1.4. We limit the evaluation to applications written in PHP to allow for uniform collection of server-side coverage. We strive to include the latest version of modern applications. For the sake of comparison, we select some applications used in previous works, for example, Kanboard in EvoCrawl [19], and both osCommerce and Leantime used in YuraScanner [32].

We make slight modifications to these applications to create a level playing field for all scanners, mainly with respect to authentication; details are provided in Appendix .1.4.

Compared Black-box Methods

We evaluate our approach against both state-of-the-art black-box academic and open-source scanners. This set of 4 scanners consists of: Arachni [1], Black Ostrich [15], Wapiti [37], and ZAP [40]. When possible, we configure the scanners to only detect XSS vulnerabilities. Since we modify the applications to make authentication easier we do not configure any authentication parameters in the scanners. While there are other academic scanners focusing on client-side code, such as jÄk, they are no longer maintained. However, as Black Ostrich uses the same method of hooking event registration as jÄk, the performance of Black Ostrich could shine a light on how an updated version of jÄk might perform.

Metrics

Our main metrics are the number of XSS vulnerabilities found and code coverage. For vulnerabilities, we present both the reported number by each scanner and also the number after we manually verify reports to filter out false positives. Some scanner's verifications of injection, including statically searching for `script` tags or not using unique IDs for payloads can result in these false positives.

In this study, we define coverage in terms of unique lines of code executed on the server-side collected using xDebug in PHP, similar to previous work [31, 19, 14]. However, web application scanner coverage could be measured in multiple different ways. Previous research has used metrics ranging from links discovered [29], to unique JavaScript code retrieved [31], or even client-side JavaScript code coverage [22]. In Section B.5.4 we discuss these metrics in more detail and argue why we use server-side code coverage in this evaluation.

B.4.2 Web Application Crawl Evaluation Results

While we qualitatively observe that SpiderSapien is able to reach new program states in the evaluated applications, the empirical measurements of coverage still show that other scanners can find things we miss. Despite this disconnect between the chosen metric and scanning quality, we still see that SpiderSapien can achieve better server-side coverage in the majority of cases. Furthermore, SpiderSapien finds far more XSS vulnerabilities than all other scanners. In the following sections, we overview the results of the local web application crawling evaluation in terms of coverage and vulnerabilities.

Coverage

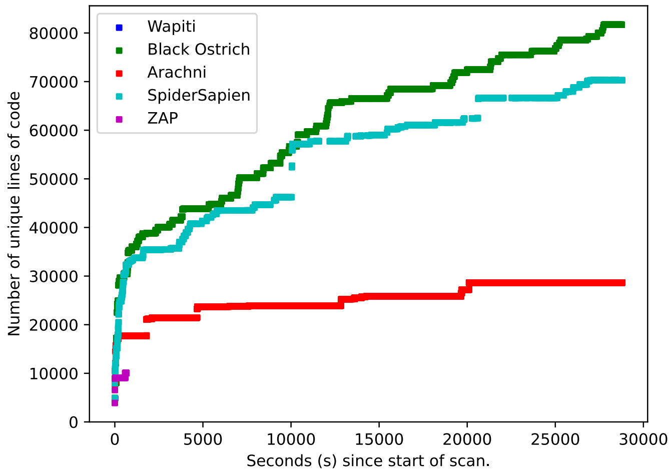
In this section, we present the server-side code coverage from running all the scanners on our set of open-source web applications. In Figure B.4 we plot the comparison between our scanner and the others, with the exact numbers presented in Appendix .1.6. The figure shows that our method outperforms the other scanners in the majority of cases. This serves as a good indicator that a focus on client-side crawling does achieve deeper scanning of applications, even for server-side code. However, there are some notable cases where other scanners perform well. Two of these we look closer at are ZAP on TinyFileManager and ZAP on DokuWiki. In the first case, ZAP triggers more errors, like invalid CSRF tokens, which we avoid by correctly retracing the form submissions. We discuss the differences between better coverage of *intended code* versus *error code* in Section B.5.1. In the case of DokuWiki, our scanner is able to break the application by correctly submitting a dangerous form that the other scanners struggle with. We explore this problem more in Section B.5.2

In Figure B.3, we highlight a small case study on the coverage over time on osCommerce. Note here that the commercial scanners either quit early, in this case ZAP and Wapiti, or quickly plateau like Arachni. While the slower academic scanners show potential of further increasing their coverage even after the 8-hour limit. In Section B.5.3 we discuss the potential of using web scanners for deeper and longer scans than commonly used.

Vulnerabilities

Here we present the XSS vulnerabilities found by each of the scanners. We include both *reported* vulnerabilities and *verified* vulnerabilities, which we

Figure B.3: The figure presents the coverage for each scanner over the eight hour evaluation on osCommerce.



have manually checked. Verifying the vulnerabilities is an important step as scanners can mistakenly report safe parts of the application as vulnerable. Across almost all applications, our method is able to find more XSS vulnerabilities than the other scanners. In total, we find 36 verified XSS vulnerabilities across all applications. Compared to 4 for the other scanners. We see that most state-of-the-art scanners do not find many XSS vulnerabilities in these modern applications. For the vulnerabilities other scanners do find, our method does also find most of them.

The exception is Arachni finding two more vulnerabilities in TinyFileManager. It should be noted that Arachni has a high rate of false positives in this case, because of their imprecise method that confirms all XSS vulnerabilities that reuse a shared payload. After finding a stored vulnerability, all subsequently attacked parameters will appear vulnerable. During our manual verification, 3 of these were confirmed to valid. However, we could not determine if Arachni actually found them or happened to mark all parameters after the first successful stored injection.

We note that while the other scanners do cover thousands of lines of code that we miss (see Section B.4.2), they do not find as many vulnerabilities, in relative terms. This might indicate that scanners should focus more on *intended* or high quality code coverage, which we discuss in Section B.5.1.

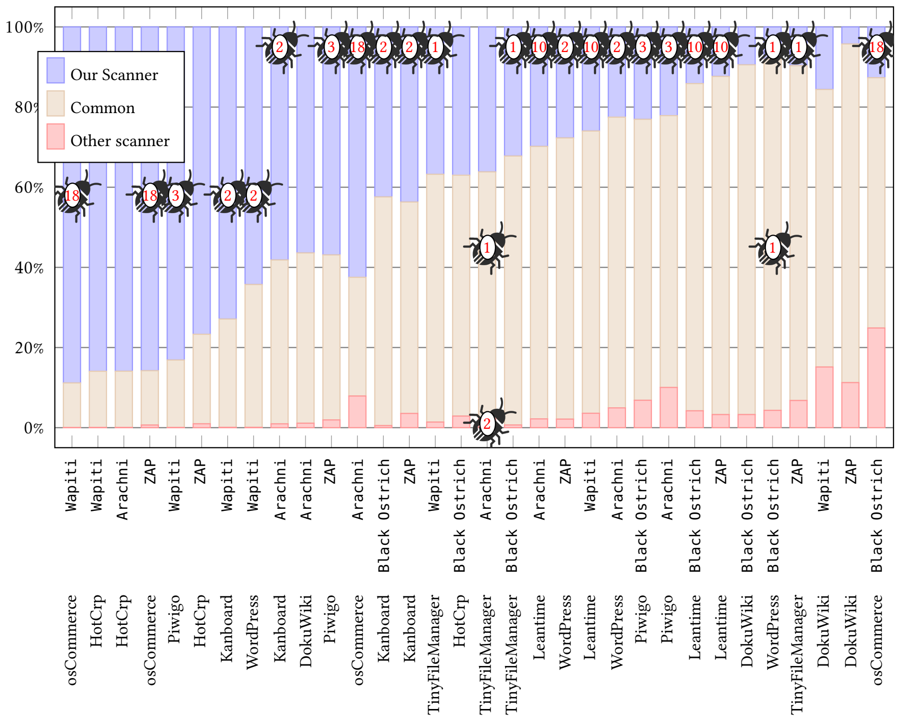


Figure B.4: The X-axis denotes the application and scanner we compare against while the Y-axis contains the fraction of unique lines of code the scanners find. Starting at the top, each column shows the fraction only we find, followed by code found by both scanners, and finally what only the other scanner finds. Similarly, the number in each bug denotes the XSS vulnerabilities found.

Table B.1: This table presents both *reported* and *verified* XSS injections from the scanners. As scanners might incorrectly report vulnerabilities we manually verify each report. We discuss the false positives (*) in more detail in Section B.4.2.

Scanner	Arachni		Black Ostrich		SpiderSapien		Wapiti		ZAP	
Type	R	V	R	V	R	V	R	V	R	V
DokuWiki	0	0	0	0	0	0	0	0	0	0
TinyFileManager	10	3*	0	0	1	1	0	0	0	0
Kanboard	0	0	0	0	2	2	0	0	0	0
WordPress	0	0	1	1	2	2	0	0	0	0
osCommerce	0	0	0	0	18	18	0	0	0	0
HotCrp	0	0	0	0	0	0	0	0	0	0
Leantime	0	0	0	0	10	10	0	0	0	0
Piwigo	0	0	0	0	3	3	0	0	0	0

B.4.3 Open Web Form Solving Evaluation

We also evaluate the performance of the LLM form solving component outside of the crawler. Rather than construct an artificial testbed, we use real forms gathered from the open web. Care is taken to not interfere with these live external websites—we discuss our ethical considerations in Section B.8.1.

Setup

We evaluate the ability of the form solver in isolation. Given an initial URL that contained a form, we load this URL in a separate Selenium script that runs only the form solver.

(1) First, we force visibility so that the script can interact with the form. (2) The LLM form solving method is provided with the form. (3) Solved values are entered with the same input automation as in the full crawler. (4) Before submitting, we turn off the network in Chrome with a custom browser extension. This is to reduce the impact of this evaluation on the open web. (5) We attempt to submit the form. (6) If the browser tries to navigate to our extension's local landing page, the form was successfully submitted. Otherwise, the attempt loops up to 3 times according to the method described in Section B.3.3. The network is re-enabled after every submission.

This setup can test solving diverse live forms on the open web, while reducing possible harm done to these websites by not submitting data. See Section B.5.6 for setup limitations.

Compared Methods

We compare to two methods. In each of these methods, we primarily alter the second step in the form solving setup. Instead of generating input values by prompting an LLM, we generate values by either a "No Input" method, or a "ZAP" method. We also modify the final retry step, in that no error feedback is provided, and the method is simply re-run.

In the "No Input" method, we provide no input. This serves as a baseline method, in that forms that *cannot* be successfully submitted by simply providing no input are *non-trivial*.

In the "ZAP" method, we recreate the form solving inputs generated by the ZAP scanner [11]. This provides a baseline heuristic method to solve forms in our evaluation. This method provides constants for most types of inputs, but selects a value from an HTML-specified range for numbers, and values based on the current date/time for relevant input types.

Form Choice

We make a new dataset of URLs with forms from domains on the Tranco [30] list, as detailed in Appendix .1.2. A random selection of 2000 of these URLs was used in this evaluation. For each method, the tested URLs and forms varies. The amount of tested URLs and forms differs between methods due to testing resource availability, network conditions, and generally variable content in the websites tested.

Results

All Forms. We include the raw results of this open web form solving evaluation, in terms of all successfully solved forms for each method, in Table 3. While the "No Input" and "ZAP" rows do not use the method of SpiderSapien, all others indicate the LLM used in our method and its temperature.

However, the relative ability of each method to solve forms, and especially non-trivial forms, is unclear in this unfiltered presentation. The primarily evaluated methods that produce inputs ("ZAP", Gemini, and Gemini Pro) only vary in effectiveness from 79.1-81.8%. In fact, the simple heuristics of "ZAP" excel by this metric. Furthermore, the "No Input" already can solve 77.8%.

The GPT models, while having the best solving performance, could not be extensively tested due to resource availability issues. This method's prompt was developed with GPT, so the performance disparity from Gemini could be due to model change without prompt adaptation (see Section B.5.7).

Table B.2: Success rate of select form solving methods on non-trivial forms. All methods are provided in Appendix .1.3.

All non-trivial forms				
Evaluated method	**URLs**	**Forms**	**Forms solved**	**Solve %**
ZAP	606	1029	414	40.2%
Gemini (t=1.5)	604	1024	483	47.1%
Gemini Pro (t=1.0)	535	873	402	46.1%
GPT4o-Mini (t=0.5)	244	568	308	54.2%
Non-trivial forms available to GPT4o-Mini				
ZAP	234	356	121	34.0%
Gemini (t=0.5)	199	315	153	48.6%
Gemini Pro (t=1.0)	198	283	134	47.4%
GPT4o-Mini (t=0.5)	244	568	308	54.2%

Non-trivial Forms. If we instead focus on non-trivial forms, we can see that LLM form solving starts showing its potential. Non-trivial forms are defined as forms that could not be solved with “No Input”. We can further normalize the results to examine only those forms available to the main compared methods. Both of these datasets are presented in Table B.2.

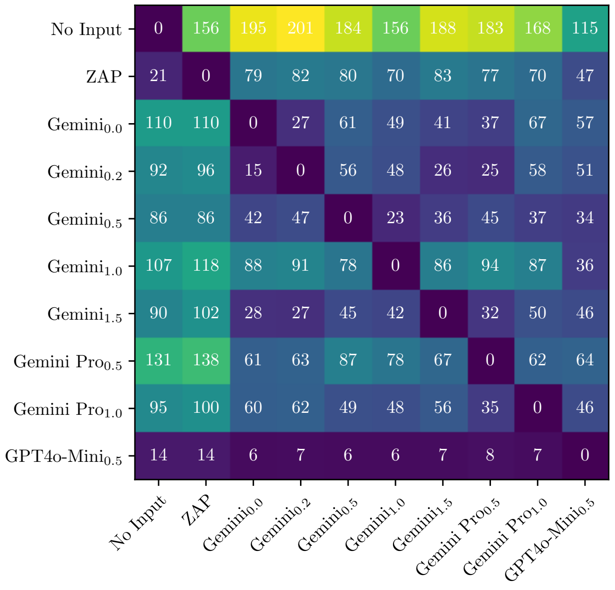
When examining all non-trivial forms and comparing them to the baseline of “ZAP” solving 40.2% of these forms, Gemini models can solve 47.1% of the forms tested. While we have a limited sample size for GPT4o-Mini, we see a further boost to 54.2% of these forms solved.

After restricting our data to forms GPT4o-Mini encountered, we see “ZAP” perform much worse. In Table 4 we also see this result when normalized to forms Gemini could test.

Comparative Performance. Another way to compare methods is how specific forms perform with pairs of (old, new) methods. We illustrate the comparative performance of methods by showing what forms start passing with a new method in Figure B.5. In this figure, we can see that methods with the brightest colored columns (larger numbers) perform worst - they were often improved upon by other methods. Alternatively, we can interpret methods with the darkest rows (smaller numbers) as performing best. We see the same results as the earlier analysis; our LLM method performs best. The same trend is shown if we restrict the data to non-trivial forms available to Gemini in Figure 7.

Performance Summary. While form solving performance in this open web evaluation setting can be hard to measure, the LLM-powered form solving

Figure B.5: Comparative performance of form solving methods for all forms. Each cell at (X,Y) counts the forms that were not solved by the Y method, but were solved by the X method.



	No Input	ZAP	Gemini$_{0.0}$	Gemini$_{0.2}$	Gemini$_{0.5}$	Gemini$_{1.0}$	Gemini$_{1.5}$	Gemini Pro$_{0.5}$	Gemini Pro$_{1.0}$	GPT4o-Mini$_{0.5}$
No Input	0	156	195	201	184	156	188	183	168	115
ZAP	21	0	79	82	80	70	83	77	70	47
Gemini$_{0.0}$	110	110	0	27	61	49	41	37	67	57
Gemini$_{0.2}$	92	96	15	0	56	48	26	25	58	51
Gemini$_{0.5}$	86	86	42	47	0	23	36	45	37	34
Gemini$_{1.0}$	107	118	88	91	78	0	86	94	87	36
Gemini$_{1.5}$	90	102	28	27	45	42	0	32	50	46
Gemini Pro$_{0.5}$	131	138	61	63	87	78	67	0	62	64
Gemini Pro$_{1.0}$	95	100	60	62	49	48	56	35	0	46
GPT4o-Mini$_{0.5}$	14	14	6	7	6	6	7	8	7	0

method we introduce in SpiderSapien shows promising performance. There is a substantial improvement from 40.2% of non-trivial forms solved by the baseline "ZAP" to 47.1% in the best Gemini model, and 54.2% in the GPT model with the most data. If we restrict the dataset to non-trivial forms available to the most promising methods, we instead see an improvement from either 34.0% to 48.6% and 54.2%, or 38.1% to 47.6% and 54.2%.

Limitations of this evaluation are presented in Section B.5.6. Discussion of specific instances where this method fails compared to the baseline methods is included in Section B.5.5.

B.5 Analysis / Discussion

B.5.1 Exploring Intended Code Paths

Our goal with this approach was to improve the exploration of what can be seen as "intended code paths"—paths in the application that the developer expects a user to follow. This goes against the classic notion of trying to "fuzz" the application with unintended values to find error-related code paths.

We believe that current web scanners focus too much on trying to find these unintended paths that they fail to explore many of the application's deeper functionalities, where vulnerabilities might reside. We cannot infer if a scanner is exploring the unintended paths by total coverage. For example, Section B.4.2 shows that ZAP has similar total coverage on TinyFileManager v.s. our approach. However, part of ZAP's unique coverage comes from error-handling code relating to incorrect CSRF tokens, because ZAP fuzz all requests parameters on the network level. In contrast, our method instead fully submits the form and avoids some unintended paths by ensuring such tokens are valid.

We cannot claim either method of exploration is strictly superior. If anything our results show the need for a combined approach or use of multiple scanners to ensure better coverage. By manually inspecting a combination of the coverage and state of the application after scans, it is clear that much of the functionality in web applications is still unexplored.

B.5.2 Destroying the State

A surprising problem we face as scanners improve is their increased potential to break the application. SpiderSapien performs relatively poorly on DokuWiki, see Section B.5.1. This is not because the other scanner possesses capabilities that our scanner lacks, but rather because our method finds and, thanks to the LLM-module, correctly submits the site configuration form. This creates an irrecoverable state by breaking the authentication method.

Future works could try to detect these dangerous actions. Either with an LLM approach similar to ours or by allowing the scanner to reset the application like Enemy-of-the-State [12], using heuristics to detect breakage [27].

B.5.3 False Negatives and Length of Scan

During the development of our scanner, we found vulnerabilities not reproduced in the final evaluation scan. We still report these to the developers for ethical reasons. Our scanner still supports each interaction necessary to find these additional vulnerabilities. We believe that the final scan did not find them due to two practical limits, First, this can be due to the evaluation time limit of eight hours. Currently, there is no consensus on the evaluation time for web scanning, with times ranging from 4 [32] hours to 24 hours [19]. From Figure B.3, we can also see that SpiderSapien, in contrast to ZAP, Wapiti and Arachni, has not plateaued or terminated. With the additional randomness in the scanner a longer evaluation could be valuable. The second reason could simply be that we destroy the application before exploring all the potential vulnerabilities, as discussed in Section B.5.2. To overcome these limits, future studies on web scanning should ideally incorporate both multiple runs and longer evaluation times.

B.5.4 Coverage Metrics

Server-side code coverage, such as that gathered through xDebug in PHP web applications, is the most common way to measure coverage in a web application scanning or fuzzing session. However, other types of coverage can be measured. For example, the recent SoK on web security crawlers [31] measures not only (1) server-side code coverage, but also (2) JavaScript source coverage, and (3) link coverage. JavaScript source coverage is defined by collecting the hashes of retrieved inline or external JS scripts.

JavaScript code coverage (4) can also be measured by retrieving the statistics of total and unused bytes of JS code through the Chrome DevTools. This has been used by Kang et al. [22], but has not yet been widely adopted by other scanning works. Comparative evaluation is hard, as such metrics need to be gathered by each scanner.

Therefore, we only present the coverage in the (1) metric: server-side code coverage. Possible client-side metrics for coverage (2, 3, 4) are more difficult to compare. Dynamically generated scripts and links will perturb these measurements.

B.5.5 LLM Failed Form Solves

As seen in Section B.4.3, despite the overall performance gains of this method, “No Input” and “ZAP” both solve forms the LLM could not. When compared to Gemini, these cases are often due to no function call being generated. Otherwise, some “successful” form solves happen when dynamic form content is not fully loaded, evading normal client-side validation.

Gemini Refusals. During the evaluation, we saw over 100 requests blocked by the content filter, mostly tending towards forms on adult websites. We also saw another set of at least 130 requests pass the content filter, but get rejected by the model. Instead of generating a function call, Gemini generates some explanation of its refusal. This also usually occurs on the same type of website. When we examine the set of forms which passed on “No Input”, but failed on a Gemini model, 44/78 of these did not generate a valid function call. 56/83 of the failures relative to “ZAP”, and 23/33 relative to GPT models, are due to the same reason.

GPT Failures v.s. “No Input”. We manually examine all 14 cases when “No Input” solves a form that GPT4o-Mini could not. 6/14 cases had valid inputs when manually tested, and could be due to form instability. Other failures included: not loading values while the network is blocked, not meeting our success conditions by navigating before submit (when an input element is changed), or a cookie consent popup preventing all interaction with the form.

GPT Failures v.s. "ZAP". We also examine some cases when "ZAP" improved upon GPT. Many of the forms and reasons from the prior comparison to "No Input" are repeated. Some new interesting examples are highlighted:

A defunct website to download YouTube videos had a search form that could accept a search term or a URL. The LLM understood that a URL could be entered in this field, and provided "https://testlink.com". However, the validation actually enforced that the text is either *not* a URL, or *only a YouTube* URL. The default text payload "ZAP" happened to work, as it was not a URL. Additional context on the page outside the form HTML could help an LLM correct its input.

Another form had input descriptions clashing with validation. The associated label specified that a *country* should be entered, while the input instead is of type *email*.

Finally, some forms allow semantically incorrect inputs. One form that specifies that a security code should be transcribed from a picture, yet accepted *no input* from "ZAP".

B.5.6 LLM Evaluation Limitations

Measuring the performance of form solving on the open web as described in Section B.4.3 is challenging, and there are limitations to this setup. This setting offers a greater variety of forms to test on, beyond the limited setting of the open-source web applications otherwise tested, but brings new challenges.

Form Instability. There is inherent instability in forms presented, due to network conditions and application changes. This results in different URLs and forms being seen by each method in Section B.4.3. We address this by normalizing to forms common to all compared methods in Table B.2, or by only comparing performance changes on the same forms in Figure B.5 and Figure 7. We observed a steady turnover in forms at URLs during evaluation; the lifetime for these pages can be short. Moreover, forms can even load differently, affecting what client-side validation might be present.

Evaluating a Successful Submit. The effects of a form successfully submitting vary. For example, a webpage might close a form dialog client-side. The criteria we use to evaluate success is navigating away from the form.

Therefore a successful form solve will not be detected, if it would normally only result in a form dialog closing client-side without a navigation. However, we must force the form to be visible, as we do not know the client-side steps to normally interact with the form. A full stateful crawl, such as with SpiderSapien, could alleviate this problem, but with further potential for harm.

FormWhisperer [23] instead judges success by intercepting a request with injected values using a network proxy. However, this success criterion could yield false positives with our method in this evaluation setting. The LLM can generate non-unique data that would already be in the request, or the request might be for input validation across the network rather than a successful submit.

Impossible Forms. Finally, a challenge in this evaluation is impossible forms; those that cannot be solved by any input. Some forms are also rendered impossible by the network-blocking setup. For example, when validation for a client-side elements occurs with server traffic. However, this cannot be fixed without allowing network traffic during submit, which could likely harm the websites tested. We also observe that some forms seem broken, and cannot successfully submit.

B.5.7 LLM Models and Prompting Choices

While we primarily evaluate on Gemini models, we developed our method on GPT models. The performance difference we observe between these families of models could therefore be due to the prompt not being re-engineered for each model.

However, our evaluation across both Gemini and GPT shows that while there may be performance differences, even the simple unoptimized prompting template used still can outperform prior methods. Future work could optimize the prompt to a specific model, such as with prompt tuning [24].

B.6 Related Work

Static Analysis. Static analysis of application source code has been applied [10, 16, 21] to detect various vulnerabilities. Dahse et al. [10] analyze PHP to find stateful second-order vulnerabilities by examining application dataflows. Huang et al. [21] find file upload vulnerabilities in PHP with symbolic analysis. Fang et al. [16] adapts dataflow analysis to use deep learning. Restler [3] analyzes only a REST API specification rather than application code. ReactAppScan [20] targets React single-page applications with abstract interpretation.

With our general approach, we can also detect and interact with React elements. While the applications in the evaluation did not use React, we were able to successfully both add and mark notes as completed in the TodoMVC React example [28]. In general, black-box dynamic approaches like ours inherently produce fewer false positives and offer precise support for dynamic language features across languages and frameworks.

Grey-box Fuzzing. Grey-box web fuzzing [35, 17, 34] typically instruments server-side code to provide coverage feedback to the fuzzing engine. Trickel et al. [34] apply this to detect both SQL and command injection. This approach is limited to single-shot reflected injection vulnerabilities. While the fuzzer can induce application changes, this approach does not reason about states. Gauthier et al. [17] abstract web applications into REST models, and then use a black-box fuzzer on Node.JS applications. Güler et al. [18] add bug oracles to provide feedback in an instrumented PHP interpreter. In contrast, our approach aims to instead generate structured inputs by correctly interacting with the client-side interface.

Black-box Scanners. CrawlJax [26] infers a state flow graph, including support for client-side user interface changes. This has since been incorporated into the ZAP scanner, and is therefore included in our evaluation. Enemy of the State [12] introduces another scanner that can model server-side state.

jÄk [29] improves detection of possible interactions, such as events, network APIs, and dynamic URLs and forms, through dynamic analysis of JavaScript code. This analysis through event registration hooking, is re-used in Black Widow.

Black Widow [14] combines navigation modeling with traversing and tracking inter-state dependencies. In this paper we evaluate SpiderSapien against Black Ostrich, which reuses much of Black Widow while extending input validations support. Our approach improves the core navigation modeling, crawling strategy, and form handling of the crawler increasing both coverage and vulnerability detection rate.

EvoCrawl [19] utilizes evolutionary search to crawl modern applications. This scanner is not included in our evaluation as the source code is unavailable. Their approach also improves code coverage and form submissions. However, the discovery of interactable elements is limited to a predefined set of element tags. Our method's interactable element discovery could improve the performance of their evolutionary crawler.

Creating Database State. SynthDB [7] prepares database-backed PHP web applications for security testing by synthesizing a database after collecting constraints with concolic execution. Spider-Scents [27] find portions of XSS vulnerabilities in web applications by inserting payloads directly into the database. In contrast, we attempt to create application state, including the database, with the client-side interface.

LLMs for Security Scanning. YuraScanner [32] leverages LLMs for task-driven scanning. LLMs help to execute tasks and workflows, such that deeper states multiple steps from the seed URL can be scanned for vulnerabilities. Their task-driven approach with a goal-based agent is a departure from the more typical crawler approach we assume, and can therefore have complementary results. The LLM form solving in this paper is also substantially different. We do not simplify and abstract the webpage for LLM form input, nor avoid the "click" actions present on the form (including checkboxes, radio elements, and dropdown selects). Additionally, we use error and completion feedback in our method, rather than assuming the form solver functions on its first attempt. We also offer an evaluation of the ability for an LLM to be easily adapted to solve forms. However, their evaluation offers further evidence that inputting values to a form is included within training data for the LLMs evaluated. Even though YuraScanner is not open-source and is subject to vetting, we hope to obtain the code and include it in the evaluation. This would help shed light on the differences between the techniques and the overlap of the new vulnerabilities found.

Yoon et al. [38] apply LLM agents to fetch interactable elements for task goals in Android applications. Future extensions of web crawling could align with the visual approach of ScreenAI [4], where a vision language model identifies elements like search bars in a UI.

Input Validation. Heuristics are the most common way for scanners to satisfy input validation constraints. For example, YuraScanner [32] applies the rules to click all checkboxes and radio elements present on a form, and select the second entry when possible in a dropdown select. In contrast to these heuristics, we do not restrict the LLMs interaction with input elements in a form. Despite our shared observations of the difficult "click" action elements, we still see acceptable performance in evaluation.

Black Ostrich [15] focuses on input validation by solving regexes with SMT collected from JS patterns and methods. ExpoSE [25] is a symbolic execution engine for JS that can solve regex-based input validation constraints. FormWhisperer [23] uses symbolic analysis of HTML and JS that extracts and solves constraints on form inputs, including between multiple form fields. In contrast, we solve forms with more diverse input validation approaches. However, these approaches are complementary in that regex solving could provide an initial input for the LLM, or vice versa.

B.7 Conclusion

Our work showcases the challenges modern web applications pose for state-of-the-art black-box scanners. To combat these challenges and help developers secure their applications, we develop a novel method to better interact with these applications. Our method achieves immersive interaction by improving detection of interactable elements and ordering interactions with these elements, while using an LLM-based approach to tackle the diverse input validations in forms. We implement this method in our scanner SpiderSapien and evaluate it on 8 web applications. Our results show improvement in both coverage and vulnerability detection, with an average increase in coverage of 21.5% compared to the *union* of all scanners and a total of 36 XSS vulnerabilities across 6 applications. Additionally, we evaluate the our LLM-powered form solving capabilities on forms from real-world websites. Our new method can solve at least 23.3% more of the non-trivial forms compared.

B.8 Ethics and Open Science

B.8.1 Ethics Consideration

Responsible Disclosure of Vulnerabilities

During the development and evaluation of this scanner, we found possibly unknown XSS vulnerabilities in six of the web applications tested. We are handling these security vulnerabilities in accordance with the best practices of ethics in security [5]. We are in the process of reporting our findings to the affected vendors, following coordinated vulnerability disclosure for all discovered vulnerabilities. We will report responses from vendors in the final paper.

Balancing Open Science and Misuse

Risk of Scanner. As a vulnerability scanning tool, this scanner could be misused to exploit vulnerabilities without consent of application owners. While we believe our method can better scan modern web applications, the basic function of this tool v.s. others. Therefore, we will open-source the code for the scanner upon acceptance.

The LLM form-solving method could be used for unintended goals such as fake account creation. However, the tool we release is a black-box scanner that cannot be directly applied to such goals. This is in contrast to YuraScanner [32], which also considered this harm and elected to not open-source their scanner. Their task-driven crawler could more easily allow targeted misuse in a way this undirected black-box scanner cannot. Furthermore, the ability for an LLM to solve a form is not fine-tuned by any additional data provided in this method, but rather seems to already be present in the models tested.

Risk of Results. We will not release all the scans generated during evaluation, as they would precisely indicate vulnerabilities. However, we hope that some vendors either fix the issues we notify them of, or indicate that they don't consider them dangerous in their threat model, so that some scans could be included in our open science artifact.

Mitigating Experiment Cost and Risk

Local Open-Source Web Applications. We evaluate our scanner's ability to crawl and find vulnerabilities in a set of locally run open-source web applications. By actively scanning only our local clones of these applications in a controlled environment, we avoid any harm caused by scanners on the web.

Open Web Form Solving. Measuring the form solving performance of our method has two components with possible experimental overhead: collecting a dataset of form-containing URLs, and evaluating the ability of our scanner to solve these forms on live websites.

We collect the dataset of form-containing URLs with a script that limited its interaction to avoid cost. For each domain, we perform a limited crawl starting from its index page, and only load 30 pages on this domain with one request to the HTML content of each. We avoid crawling pages outside of the domain, and insert sleeps between requests. We believe these pages are intended to be public, as they can only be linked from the index page each domain is redirected to.

We evaluate form solving performance by loading a webpage containing a form in a browser, but disable network traffic before attempting to submit a form. Allowing forms to actually submit data to the website has an unacceptable risk of corrupting the system state of these websites. Therefore, we disable network traffic before form submission to prevent this possible harm. Otherwise, this interaction should be of low cost, similar to crawling by a search engine.

B.8.2 Open Science

Source Code

We will open source our implementation upon publication.

Artifacts

We will include modifications made to web applications to facilitate our evaluation across scanners in the same repository. At publication, we will also share more artifacts of this paper, such as complete Docker setups for these web applications, to facilitate at least functionality assessment.

Bibliography

[1] Arachni. `https://www.arachni-scanner.com`.

[2] Arachni. arachni ui_form.rb. `https://github.com/Arachni/arachni/blob/8e5c5d0a9fd6c5555bc3bbf411c8fa411f12b6db/lib/arachni/element/ui_form.rb#L19C26-L19C31`.

[3] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: stateful REST API fuzzing. In *ICSE*, pages 748–758. IEEE / ACM, 2019.

[4] Gilles Baechler, Srinivas Sunkara, Maria Wang, Fedir Zubach, Hassan Mansoor, Vincent Etter, Victor Carbune, Jason Lin, Jindong Chen, and Abhanshu Sharma. Screenai: A vision-language model for UI and infographics understanding. In *IJCAI*, pages 3058–3068. ijcai.org, 2024.

[5] Michael D. Bailey, David Dittrich, Erin Kenneally, and Douglas Maughan. The menlo report. *IEEE Secur. Priv.*, 10(2):71–75, 2012.

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *NeurIPS*, 2020.

[7] An Chen, Jiho Lee, Basanta Chaulagain, Yonghwi Kwon, and Kyu Hyung Lee. Synthdb: Synthesizing database via program analysis for security testing of web applications. In *NDSS*. The Internet Society, 2023.

[8] Firefox code to identify elements associated with events by frameworks. `https://github.com/mozilla/gecko-dev/blob/80a3d06820b31e1d95beb582f15e789cda9f6e03/devtools/server/actors/inspector/event-collector.js`.

[9] World Wide Web Consortium. Css basic user interface module level 3 (css3 ui). https://drafts.csswg.org/css-ui-3/#valdef-cursor-pointer, 2023.

[10] Johannes Dahse and Thorsten Holz. Static detection of second-order vulnerabilities in web applications. In *USENIX Security Symposium*, pages 989–1003. USENIX Association, 2014.

[11] ZAP default value generator code. `https://github.com/zaproxy/zaproxy/blob/4fe0566304b1d646d248f041aeec59e6e9e8bfaa/zap/src/main/java/org/zaproxy/zap/model/DefaultValueGenerator.java#L74`.

[12] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *USENIX Security Symposium*, pages 523–538. USENIX Association, 2012.

[13] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *DIMVA*, volume 6201 of *Lecture Notes in Computer Science*, pages 111–131. Springer, 2010.

[14] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. Black widow: Blackbox data-driven web scanning. In *SP*, pages 1125–1142. IEEE, 2021.

[15] Benjamin Eriksson, Amanda Stjerna, Riccardo De Masellis, Philipp Rümmer, and Andrei Sabelfeld. Black ostrich: Web application scanning with string solvers. In *CCS*, pages 549–563. ACM, 2023.

[16] Yong Fang, Shengjun Han, Cheng Huang, and Runpu Wu. Tap: A static analysis model for php vulnerabilities based on token and deep learning technology. *PloS one*, 14(11):e0225196, 2019.

[17] François Gauthier, Behnaz Hassanshahi, Benjamin Selwyn-Smith, Trong Nhan Mai, Max Schlüter, and Micah Williams. Backrest: A model-based feedback-driven greybox fuzzer for web applications. *CoRR*, abs/2108.08455, 2021.

[18] 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 *USENIX Security Symposium*. USENIX Association, 2024.

[19] Xiangyu Guo. Evocrawl: Exploring web application code and state using evolutionary search. Master's thesis, University of Toronto (Canada), 2023.

[20] Zhiyong Guo, Mingqing Kang, V. N. Venkatakrishnan, Rigel Gjomemo, and Yinzhi Cao. Reactappscan: Mining react application vulnerabilities via component graph. In *CCS*, pages 585–599. ACM, 2024.

[21] Jin Huang, Yu Li, Junjie Zhang, and Rui Dai. Uchecker: Automatically detecting php-based unrestricted file upload vulnerabilities. In *DSN*, pages 581–592. IEEE, 2019.

[22] Zifeng Kang, Song Li, and Yinzhi Cao. Probe the proto: Measuring client-side prototype pollution vulnerabilities of one million real-world websites. In *NDSS*. The Internet Society, 2022.

[23] Björn Karthein, Cristian-Alexandru Staicu, and Andreas Zeller. A generalized approach for solving web form constraints. In *ASE*, pages 2460–2461. ACM, 2024.

[24] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In *EMNLP (1)*, pages 3045–3059. Association for Computational Linguistics, 2021.

[25] Blake Loring, Duncan Mitchell, and Johannes Kinder. Sound regular expression semantics for dynamic symbolic execution of javascript. *CoRR*, abs/1810.05661, 2018.

[26] Ali Mesbah, Engin Bozdag, and Arie van Deursen. Crawling AJAX by inferring user interface state changes. In *ICWE*, pages 122–134. IEEE Computer Society, 2008.

[27] Eric Olsson, Benjamin Eriksson, Adam Doupé, and Andrei Sabelfeld. Spider-scents: Grey-box database-aware web scanning for stored XSS. In *USENIX Security Symposium*. USENIX Association, 2024.

[28] Addy Osmani. Todomvc. https://todomvc.com/, 2023.

[29] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. jäk: Using dynamic analysis to crawl and test modern web applications. In *RAID*, volume 9404 of *Lecture Notes in Computer Science*, pages 295–316. Springer, 2015.

[30] Victor Le Pochat, Tom van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *NDSS*. The Internet Society, 2019.

[31] Aleksei Stafeev and Giancarlo Pellegrino. Sok: State of the krawlers - evaluating the effectiveness of crawling algorithms for web security measurements. In *USENIX Security Symposium*. USENIX Association, 2024.

[32] Aleksei Stafeev, Tim Recktenwald, Gianluca De Stefano, Soheil Khodayari, and Giancarlo Pellegrino. Yurascanner: Leveraging llms for task-driven web app scanning. *NDSS*, 2024.

[33] Why Software Is Eating the World. `https://a16z.com/why-software-is-eating-the-world/`.

[34] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect SQL and command injection vulnerabilities. In *SP*, pages 2658–2675. IEEE, 2023.

[35] Orpheas van Rooij, Marcos Antonios Charalambous, Demetris Kaizer, Michalis Papaevripides, and Elias Athanasopoulos. webfuzz: Grey-box fuzzing for web applications. In *ESORICS (1)*, volume 12972 of *Lecture Notes in Computer Science*, pages 152–172. Springer, 2021.

[36] w3af. `http://w3af.org`.

[37] Wapiti. `https://wapiti-scanner.github.io`.

[38] Juyeon Yoon, Robert Feldt, and Shin Yoo. Autonomous large language model agents enabling intent-driven mobile GUI testing. *CoRR*, abs/2311.08649, 2023.

[39] Michal Zalewski. Skipfish, 2015.

[40] OWASP ZAP. `https://www.zaproxy.org`.

.1 Appendix

.1.1 LLM Form Solving Prompt

We provide additional details about the prompt for our LLM form solving method here.

Figure 6 contains the complete prompting template for LLM form solving We also provide the complete specification of our LLM function calling:

Figure 6: Prompting template for LLM form solving

System prompt:

```
You are a webpage tester. Do make assumptions about what values
to plug into functions. Randomly generate realistic values to test.
Match the language and region.
```

Few-shot examples:

Example user prompt

```
Send user input to the following form. <form> ... example ... </form>
```

Example LLM response

```
[(CSS selector, input value), ...]
```

User prompt

```
Required elements are ...
All input elements are: ...
Input to element ... must match regex ...
Send user input to the following form. <form> ... </form>
```

LLM response

```
[(CSS selector, input value), ...]
```

If previous form solve failed:

New user prompt

```
The previous input you gave me failed for
inputting input_val to css_sel, and ...

Furthermore, the previous input you gave
me failed for locating css_sel, and ...

There are errors on this form.
... Error messages ...

Generate new user input values for the previous form.
```

New LLM response

```
[(CSS selector, input value), ...]
```

```
functions = {
  "name": "send_user_input",
  "description": "Send␣valid␣input␣to␣the␣website␣form.",
  "parameters": {
    "type": "object",
    "properties": {
      "elements": {
          "type": "array",
          "description": "The␣CSS␣locator␣and␣user␣input␣for␣the␣
              input␣elements␣of␣a␣website␣form.",
          "items": {
            "type": "object",
            "properties": {
              "locator" : {
                "type": "string",
                "description": "A␣CSS␣locator␣for␣the␣website␣form
                    ␣element.",
              },
              "input" : {
                "type": "string",
                "description": "User␣input␣for␣the␣website␣form␣
                    element.",
              },
            },
            "required": ["locator", "input"]
          },
        },
      },
      "required": ["elements"]
}
```

.1.2 Form Dataset

We accumulate a dataset of forms by crawling a portion of the Tranco standard list of top-ranked web domains [30], downloaded at 2024-10-21. For each of the initial 9046 domains in the Tranco list, we do a simple crawl that follows static HTML links from a seed URL until a threshold of 30 pages are found, or links are exhausted.

As indicated by the Stafeev [31] SoK, we use randomized BFS navigation and URL path equality + query string key page equality. During the crawl, we log any pages that contain a `form` element. Across these crawled 9046 domains, we discover 118455 unique URLs with at least one `form` element. These crawls were performed on 2024-10-31. Out of the 118455 form-containing URLs, we find 6916/9046 domains with 1-30 URLs each, averaging

17.1 URLs.

We randomly shuffled these URLs, and selected the initial 2000 URLs to use in this evaluation. This sample contains 1653 domains with 1-4 URLs each, averaging 1.2 URLs.

Across the 2000 form-containing URLs selected for this evaluation, we find 4953 forms. This evaluation was performed in November-December 2024, at which point 1994/2000 of these URLs were still live. For this evaluation, we consider a form to be identical if it is in the same relative index on the page.

During the later evaluation of form solving methods, URLs on 1649/1653 domains could be reached, with between 1-4 URLs each, averaging 1.2 URLs per domain. 1744/1994 of these resolved URLs contained forms, with 1-279 forms per URL, averaging 2.8 forms.

.1.3 Additional Form Solving Data

We provide all form solving evaluation results on all forms in Table 3.

Table 3: Success rate of all form solving methods on all forms.

Evaluated method	URLs	Forms	Forms solved	Solve %
No Input	1974	4157	3233	77.8%
ZAP	1876	3492	2856	81.8%
Gemini (t=0.0)	1828	3308	2649	80.1%
Gemini (t=0.2)	1844	3327	2683	80.6%
Gemini (t=0.5)	1701	3017	2422	80.2%
Gemini (t=1.0)	1519	2669	2110	79.1%
Gemini (t=1.5)	1836	3243	2611	80.5%
Gemini Pro (t=0.5)	1730	3077	2430	79.0%
Gemini Pro (t=1.0)	1631	2785	2219	80.0%
GPT4-Turbo (t=0.5)	19	38	34	70.8 %
GPT4o (t=0.5)	122	181	153	84.5%
GPT4o-Mini (t=0.5)	784	1793	1519	84.7%

We provide a full version of the evaluation results on non-trivial forms in Table 4.

We also include the comparative performance of methods when restricted to the same set of forms available to a Gemini model in Figure 7.

.1.4 Web Applications

In Table 5 we present the exact version of each web application used in the evaluation.

Table 4: Success rate of all form solving methods on non-trivial forms.

All Non-trivial forms

Evaluated method	URLs	Forms	Forms solved	Solve %
ZAP	606	1029	414	40.2%
Gemini (t=0.0)	599	1021	472	46.2%
Gemini (t=0.2)	604	1025	473	46.2%
Gemini (t=0.5)	549	960	451	47.0%
Gemini (t=1.0)	493	822	370	45.0%
Gemini (t=1.5)	604	1024	483	47.1%
Gemini Pro (t=0.5)	562	948	432	45.6%
Gemini Pro (t=1.0)	535	873	402	46.1%
GPT4-Turbo (t=0.5)	7	16	13	81.2%
GPT4o (t=0.5)	43	65	41	63.1%
GPT4o-Mini (t=0.5)	244	568	308	54.2%

Non-trivial forms available to GPT4o-Mini

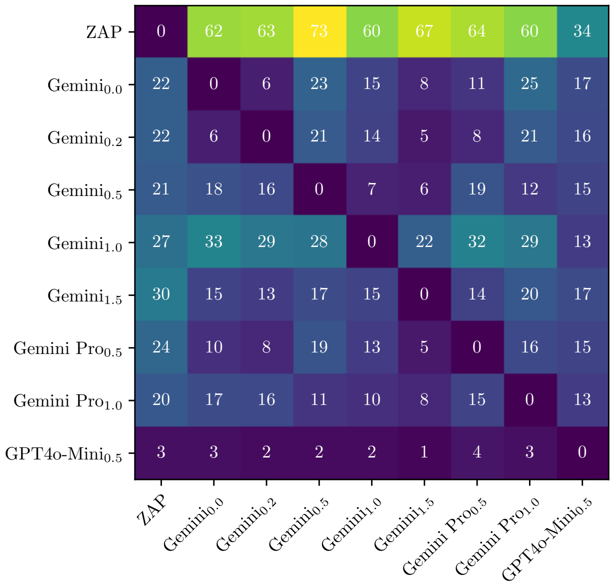
Evaluated method	URLs	Forms	Forms solved	Solve %
ZAP	234	356	121	34.0%
Gemini (t=0.0)	226	331	151	45.6%
Gemini (t=0.2)	228	346	153	44.2%
Gemini (t=0.5)	199	315	153	48.6%
Gemini (t=1.0)	205	310	150	48.4%
Gemini (t=1.5)	221	332	150	45.2%
Gemini Pro (t=0.5)	208	302	432	45.6%
Gemini Pro (t=1.0)	198	283	134	47.4%
GPT4-Turbo (t=0.5)	7	9	6	66.7%
GPT4o (t=0.5)	43	65	41	63.1%
GPT4o-Mini (t=0.5)	244	568	308	54.2%

Non-trivial forms available to Gemini (t=0.5)

Evaluated method	URLs	Forms	Forms solved	Solve %
ZAP	522	767	292	38.1%
Gemini (t=0.0)	539	836	387	46.3%
Gemini (t=0.2)	541	841	388	46.1%
Gemini (t=0.5)	549	960	451	47.0%
Gemini (t=1.0)	472	741	339	45.8%
Gemini (t=1.5)	545	853	386	45.3%
Gemini Pro (t=0.5)	518	759	353	46.5%
Gemini Pro (t=1.0)	512	783	369	47.1%
GPT4-Turbo (t=0.5)	7	4	7	57.1%
GPT4o (t=0.5)	38	52	30	57.7%
GPT4o-Mini (t=0.5)	199	315	166	52.7%

We made code changes to each application, which we will include in our open-source artifact upon publication. In particular, we modify the applications to help scanners automatically authenticate, either by instrumenting the authentication function or by simulating an automatic login. This is necessary as many of the other scanners we evaluate fail to correctly sign in on

Figure 7: Comparative performance of form solving methods for non-trivial forms available to the Gemini (t=0.5) model



some of these modern applications. It also helps the scanners re-authenticate even if they sign out by mistake. Avoiding sign outs is an orthogonal problem, but one that all scanners face. With these modifications we can learn more about the crawling capabilities of the various scanners without getting stuck on their varying support for successful authentication.

Additionally, since we are interested in detecting vulnerable *code* in web applications, we deactivate hardening functionality such as Content-Security Policy (CSP). In practice, this only affects Kanboard. We argue that it is still important to find XSS vulnerabilities even if they are mitigated by CSP as someone could run an instance without CSP. Furthermore, there is precedence for the Kanboard developers fixing XSS vulnerabilities [19] .

.1.5 Crawling Algorithm

A more detailed overview of the crawling strategy is presented in Algorithm 2.

.1.6 Coverage Results Details

In Table 6 we present the exact numbers for the coverage evaluation.

Table 5: The table presents the versions and web applications used in the evaluation.

Application	Version	Description
DokuWiki	2024-02-06b	CMS
TinyFileManager	2.6	File Explorer
Kanboard	1.2.40	Project Manager
WordPress	6.6.2	CMS
osCommerce	4.14.63493	Ecommerce
HotCrp	9588ab0	Conference Manager
Leantime	3.3.3	Project Manager
Piwigo	14.3.0	Photo Album

Table 6: Lines of code (LoC) executed on the server. Each column represents the comparison between SpiderSapien and another crawler. The cells contain three numbers: unique LoC covered by SpiderSapien ($A \setminus B$), LoC covered by both crawlers ($A \cap B$) and unique LoC covered by the other crawler ($B \setminus A$). The numbers in bold highlight which crawler has the best coverage.

Crawler	Arachni			Black Ostrich			Wapiti			ZAP		
	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$
DokuWiki	**9 510**	7 167	188	**1 630**	15 047	565	**3 068**	13 609	2 975	799	15 878	**2 113**
TinyFileManager	**549**	934	35	**481**	1 002	10	**553**	930	21	**154**	1 329	108
Kanboard	**9 136**	6 425	151	**6 635**	8 926	82	**11 353**	4 208	14	**7 049**	8 512	574
WordPress	**12 961**	41 750	2 843	**4 586**	50 125	2 474	**35 197**	19 514	52	**15 483**	39 228	1 192
osCommerce	**47 679**	22 611	6 029	11 855	58 435	**23 266**	**62 447**	7 843	5	**60 697**	9 593	467
HotCrp	**26 264**	4 284	23	**11 639**	18 909	910	**26 265**	4 283	18	**23 658**	6 890	300
Leantime	**9 063**	20 669	667	**4 390**	25 342	1 308	**8 010**	21 722	1 108	**3 797**	25 935	1 006
Piwigo	**6 993**	21 426	3 172	**7 032**	21 387	2 087	**23 625**	4 794	6	**16 485**	11 934	560

Algorithm 2 Algorithm for crawling strategy.

```
$elements \leftarrow getInteractables(url)$
$maxInteractions \leftarrow 10$ // Checked in tryInteract
while $link = elements.links.pop()$ do
    $interact(link)$
    $updateElements(getInteractables())$
    if $doneShallow()$ then
        $break$
    end if
end while
while $hasWork()$ do
    $tryInteractForm()$
    if $rand() < 0.05$ or $nothingRecentlyNew()$ then
        $tryInteractRandomLink()$
    end if
    $tryInteractNewActiveInput()$
    $tryInteractNewActiveElement()$
    $tryInteractNewInput()$
    $tryInteractNewElement()$
    $tryRandom()$
end while
```

Browser Extensions

C

FakeX: A Framework for Detecting Fake Reviews of Browser Extensions

Eric Olsson, Benjamin Eriksson, Pablo Picazo-Sanchez, Lukas Andersson, and Andrei Sabelfeld

19th ACM ASIA Conference on Computer and Communications Security (ACM ASIACCS 2024)

Abstract

Browser extensions boost user experience on the web. Similarly to smartphone app stores, browsers like Chrome distribute browser extensions via their Web Store, enabling a thriving market of third-party developed extensions. The Web Store incorporates a user review system to help users decide which extensions to install. Unfortunately, the open nature of the review system is subject to reputation manipulation. As browser vendors fight reputation manipulation, attackers employ more sophisticated methods to stay under the radar. Focusing on fake reviews, we identify several techniques attackers use: fake accounts, disjoint sets of fake accounts for different extensions, automation of generated reviews, and focusing on reviews rather than ratings. We present FakeX, a framework to detect fake reviews by focusing on inference from review metadata. FakeX employs five distinct methods, including temporal distribution analysis, relationship clustering, and ratio-based assessments, to unveil patterns indicative of fake reviews. Evaluation of over 1.7 million reviews reveals the effectiveness of FakeX in identifying hundreds of fake review campaigns. Furthermore, our investigation of these fake reviews uncovers 86 malicious extensions, mounting attacks that range from data-stealing to monetization, impacting over 64 million users. In addition, we collaborate with Adblock Plus and Avast to demonstrate FakeX in action, expanding a seed list of newly detected malicious extensions to discover a further 16 malicious extensions with millions of users, where, in some cases, attackers tried to improve malicious code.

C.1 Introduction

Browser extensions are user-friendly applications that personalize the browsing experience by introducing features and/or modifying the appearance of web pages. Developed using standard web languages like HTML and JavaScript, extensions empower developers to easily craft applications that interact smoothly with the browser's components and user interface. These extensions have attracted millions of users globally [7], contributing to the popularity of extension-enabled browsers such as Google Chrome.

Web browser vendors feature app stores for distributing approved extensions. The foremost example is the Chrome Web Store, which distributes extensions for the Chrome web browser as well as for other Chromium-based browsers such as Brave and Opera.

Table B.1: Fake review techniques vs detection methods.

Fake Review Technique	Description	Detection Method(s)	Section
Disjoint Sets of Fake Accounts	Disjoint sets of multiple accounts for reviewing.	ATW, HVC	Sections C.3.1 and C.3.2
Fake Accounts	Using a set of accounts to review multiple extensions.	CoR	Section C.3.3
Spamming	Large volume of reviews within a short period of time	Spam Detection	Section C.3.4
Written Review Dominance	Only writing reviews but not rating	Written Ratio	Section C.4.5

The Web Store incorporates a user review system, allowing users to share feedback on their installed extensions—uniquely identified by an ID[1]. This system is crucial for promoting high-quality extensions and assisting users in deciding which extensions to install [21]. Users can rate extensions on a 1 to 5-star scale and provide written reviews.

Unfortunately, the open nature of the Web Store's review system has led to the emergence of reputation manipulation. Reputation manipulation occurs when individuals or groups artificially enhance or undermine an extension's reputation, often through fake reviews and ratings. The problem is exacerbated by security experts recommending that users read reviews to enhance their Internet safety [43, 9, 38, 6]. This issue impacts the platform's credibility and may cause users to install low-quality or malicious extensions [45].

Faking reviews has become an attractive target for monetization on the black market [31], with several websites and communities offering to sell reviews online [42, 51, 16, 15, 52, 50]. These activities thrive despite vendors like Google explicitly forbidding reputation manipulation [19], including attempts at inflating reviews and ratings [18].

At the same time, detecting fake reviews is challenging, which explains why they persist on stores like the Web Store [27]. As we will see, modern fake reviews attempt to stay under the radar by avoiding obvious faking techniques so they will not be flagged for anomalies, such as excessive reviews from a single user. Motivated by these challenges, we propose two research questions:

RQ1: Can fake reviews be detected on the Chrome Web Store?

RQ2: Can methods for detecting fake reviews help discover malicious extensions?

The text of fake reviews is often generic, making it hard to distinguish from benign reviews. Sometimes, there is even no distinction as vague text can be copied from legitimate reviews to reapply as fake reviews to a broad swath

[1]For example, the official Google Translate extension has the unique ID aapbdbdomjkkjkaonfhkkikfgjllcleb. All IDs used in this study are included in Appendix C.5.

of extensions. Furthermore, fake review authors employ various concealment techniques to stay undetected. Indeed, we identify several techniques fake review authors use to operate while staying under the radar: fake accounts, disjoint sets of fake accounts for different extensions, automation of generated reviews, and focusing on reviews rather than ratings. A key observation is that we can still track these techniques through the temporal metadata of the reviews and user IDs.

To investigate our research questions, we introduce FakeX, a framework for detecting fake reviews. A principal strength of FakeX is that it does not rely on the reviews' content but focuses on the metadata, which includes the timestamps associated with users' Web Store reviews. In particular, we focus on 1) the temporal distribution of reviews, 2) the relationship between reviewers, and; 3) the ratios between ratings and reviews.

FakeX comprises five main methods, where three focus on the temporal distribution of the reviews, whereas the other two use the relationships between reviewers. In particular, FakeX offers 1) ATW, a novel method for identifying multiple accounts who post reviews in close temporal proximity; 2) HVC, a machine learning-based approach that clusters reviews by their timestamps, not only within the same extension but also across multiple extensions; 3) Spam Detection, a method focused on extracting bursts of high review activity in a short period within an extension; 4) CoR, a method that focuses on discovering clusters of reviewers who consistently review the same extensions, and; 5) Written Ratio, a method that use the ratio between rating and reviews to find extensions with an exceptionally high fraction of written reviews. Table B.1 summarizes the methods used to create fake reviews and which of our methods detect them.

To evaluate FakeX, we download all 1,782,702 reviews of all 115,124 extensions in the Web Store as of February 9, 2023. Answering RQ1 positively, FakeX uncovers hundreds of review campaigns sharing large numbers of reviewers, some consisting of thousands of accounts. One method in FakeX finds 59 clusters of 286 extensions sharing temporal patterns in their fake reviews.

Turning to RQ2, we examine extensions with fake reviews to determine if they are also malicious. In total, we find 86 malicious extensions, with attacks ranging from stealing search query data from users to redirecting users to fake surveys to win prizes. These extensions share a total of over 64 million users. Although the number of downloads can also be manipulated [38], it is still staggering.

In addition to our manual analysis, we collaborate with Adblock Plus and Avast to demonstrate how FakeX can be leveraged to expand a seed list of malicious extensions. Using Adblock Plus' list of 18 newly discovered malicious extensions [36], we discover 16 new associated malicious extensions with millions of users, where attackers sometimes tried to improve malicious code. This practical deployment of FakeX resulted in public acknowledgments of our findings by Avast and Adblock Plus [36, 35] and the removal of all of these extensions from the Web Store by Google.

In summary, the paper's contributions are the following:

- We analyze reviews in the Web Store and identify four techniques for fake reviews (Section C.2).
- Based on these techniques, we propose FakeX and describe our five novel methods to detect fake reviews (Section C.3)
- We evaluate our methods on all reviews in the Web Store to demonstrate how FakeX detects fake reviews (Section C.4).
- We show how clusters of extensions with fake reviews can be leveraged to find malicious extensions (Section C.5).

We discuss limitations in Section C.6, present related work in Section C.7, and conclude the paper in Section C.8.

We stress that although we uncover a correlation between fake reviews and malicious extensions, the techniques used by FakeX discover fake reviews rather than malicious extensions. Indeed, many of the extensions found with our method and highlighted in our analysis, whose review patterns indicate reputation manipulation, are not malicious as of February 2023 (see Appendix C.5).

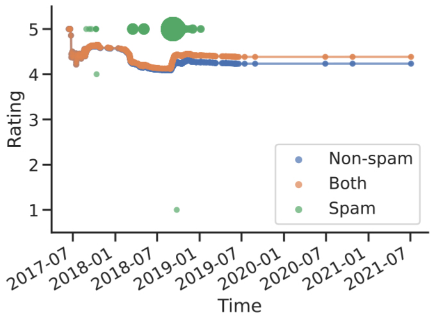
Coordinated Disclosure, Ethical Considerations, and Open-source Artifacts.. We have reported our findings to Google, including the malicious extensions we find with FakeX. In total, we find 86 and of these 44 have been removed so far.

In line with the ethical principles for cybersecurity research from the Menlo Report [41], our research does not cause any harm to users or developers. We include extension and user IDs and names to aid the reproducibility of the results presented in this paper. We open-source the code for FakeX[2].

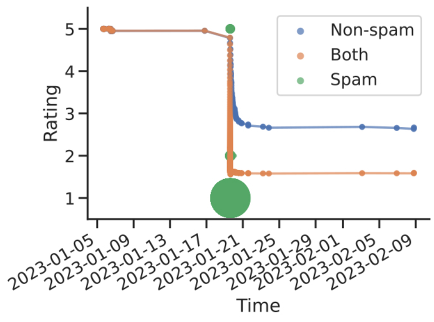
C.2 Fake Reviews on Chrome Web Store

Reviews on Web Store. The Chrome Web Store [17] is the main repository for Chrome browser extensions. The Web Store allows users to write reviews and rate extensions. To submit a review, users must log in and install the

[2]FakeX code and sample data: `https://www.cse.chalmers.se/research/group/security/fakex`



(a) Promotion



(b) Demotion

Figure B.1: Promotion and demotion examples of browser extensions. “Spam” refers to reviews within three minutes of each other (More details in Section C.3.4). The size of a point indicates the number of reviews. Every point represents new review activity of the extension. For “Non-spam” and “Both” the moving average up to that point is presented, as would be shown to real users of the Chrome Web Store.

extension. Users can leave a review, including text and a star rating, or rate the extension. These ratings, the number of stars given by users, are presented as an average across all user ratings in the Web Store. We cannot know the exact breakdown of the individual ratings nor when they were added. However, we can access the text, username, user ID, timestamp, and rating for each full review. Consequently, we can only know the entire rating history if all ratings come from reviews.

Reviews abused. Reviewers can influence how the Web Store ranks extensions. It is possible both to promote (posting positive reviews and high ratings) and demote (posting negative reviews and low ratings) extensions [21]. Consequently, reviewers can damage extensions’ reputation and reduce their perceived quality [14].

Fake review authors might cooperate and organize campaigns using different techniques to promote or demote extensions based on their reviews. For example, they might use multiple accounts or spread reviews out in time to avoid detection. These review campaigns aim to add as many reviews as possible in as short a time as possible, without being detected and removed [51, 50].

Figure B.1a includes an example of reviewers promoting an extension[3]

[3]hgjdbeiflalimgifllheflljdconlbig

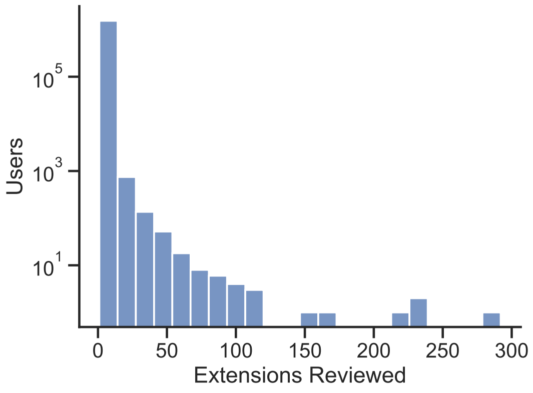


Figure B.2: Distribution of the number of browser extensions users review, with at least one review, in the Web Store as of February 2023. On average these users review 1.16 extensions.

by artificially increasing the rating. We identify *spam* reviews as reviews within three minutes of each other (as we detail in Section C.3.4). Around 2019, a larger spam campaign took place, after which we can see that the combined rating is higher than the non-spam one. We also include an example of reviewers demoting another extension[4]. In Figure B.1b, we see a large amount of 1-star spam-marked reviews around the 21st.

Fake review techniques. We identify four techniques aimed at manipulating the reputation of extensions on the Web Store while evading detection. We refer to these reviews produced with the goal of reputation manipulation as *fake reviews*. The four techniques we focus on are *fake accounts*, *disjoint sets of fake accounts*, *spamming*, and *written ratio reviews*. Furthermore, we define a *review campaign* as a coordinated effort using multiple reviews to manipulate the reputation of one or more extensions. For example, if someone pays for 20 fake reviews to promote two extensions, these 20 reviews would be part of the same campaign.

1) *Disjoint Sets of Fake Accounts.* A motivated attacker might use disjoint—or partially disjoint, sets of fake accounts to write reviews. Using unique accounts makes fake reviewers less likely to be detected [31]. For example, five can review one extension from a set of ten accounts while the other five review another, making it harder to track the campaign.
2) *Fake Accounts.* A weaker version of the previous attack is to use multiple accounts, but not necessarily unique ones. Still, attackers might prefer

[4]fiikommddbeccaoicoejoniammnalkfa

to use a set of accounts instead of simply using one account to write all fake reviews to avoid detection. In Figure B.2, we show how many extensions reviewers review. We observe that the majority of reviewers, over 91%, who review at least one extension only review one. That is, an overwhelming majority of reviewers only review one extension.

3) *Spamming.* Fake review authors may use automated tools or bots to submit many reviews without the need for human interaction. Unlike the previous methods, this approach requires analyzing the frequency and timing of reviews to distinguish them from legitimate user feedback.
4) *Written Review Dominance.* The Web Store allows users to rate extensions (1-5 stars) or write a review and rate the extensions. Since a valid account is needed for rating and reviewing, the attacker has no additional technical challenge. Adding text together with a rating is more persuasive. In the case of malicious extensions, malicious reviewers can include keywords such as "safe" and "secure" to trick users. Full-text reviews are also what is being provided by fake review services [50, 51, 15, 52].

Motivated by these manipulation techniques, we set out to propose a general framework for detecting fake reviews and evaluate it on reviews from the Web Store.

C.3 FakeX: Framework

This section presents FakeX, a framework that combines five methods to detect fake reviews of extensions in the Web Store. Our primary goal is to detect review campaigns (RQ1) and only then identify potentially malicious extensions (RQ2).

Detecting Fake Reviews in the Web Store. In the following, we present three methods to detect fake reviews of the Web Store: *Aggregated Time Window* (ATW), *Horizontal Vertical Clustering* (HVC), and *Co-Reviewer* (CoR) analysis. While all three methods attempt to detect abnormal review patterns by detecting coordinated reviews on extensions and generating clusters of extensions with shared behavior, they have different approaches. As we will see, the main difference between these methods is that CoR primarily targets clusters of reviewers reusing their accounts, while both ATW and HVC address the case where fake review authors create new accounts or multiple accounts are otherwise used.

C.3.1 Aggregated Time Window (ATW)

Intuition. The ATW method aims to link reviews posted within close temporal proximity. This is known in the literature as a *burst* and is formally defined as short periods of intensive activity followed by long periods of inactivity [3]. By focusing on the temporal aspect, instead of reviewer IDs or relationships, we can detect attacks using disjoint sets of accounts, as explained in Section C.2.

Figure B.3 presents an example with two extensions, A and B, with circles representing the reviews they get over time. ATW will connect reviews one-to-one within the same time bursts. This effectively creates a graph with reviews as nodes and edges connecting them if these reviews are within the same burst. Multiple burst connections are possible. Review 2 can connect to either review 1 or 3 in this example. We maximize the number of connections. Note that if review 2 connects with review 3, then 1 and 4 will not be connected. ATW does not consider internal connections such as 1 to 3. Finally, we consider the individual and common (shared) number of reviews before clustering and filtering those that do not meet a specified threshold. In this example, reviews 1, 2, 3, and 4 are in common, while review 5 is not. This filter is crucial to avoid clustering all extensions with very frequently reviewed extensions.

Method. First, we connect all reviews in the same burst. At this stage, we do not have a one-to-one constraint. Second, we filter these connections with a threshold for the ratio of burst shared between two extensions to the max of the two extensions' total reviews. We remove any connection where the extensions share less than four bursts to further remove noise. This helps remove coincidental connections where one extension with few reviews happens to be paired up with a frequently reviewed extension with potentially hundreds of reviews. Given the constructed graph, we face the issue of overlapping connections, as the algorithm connects every review to every other in close temporal proximity. On this graph, we apply discrete optimization to match every review with at most one other, optimizing the total number of connections.

When implementing this algorithm, a critical detail is that no review is connected to another review of the same extension. The lack of such connections naturally makes the graph of one extension pair bipartite. Bipartite graphs are graphs in which vertices can be divided into two separate, non-intersecting groups. If a graph is bipartite, it also implies that the incidence matrix of that graph is guaranteed to be unimodular [30]. This property allows discrete optimization to be applied with low computational overhead.

Finally, after the discrete optimization ensures the reviews are connected

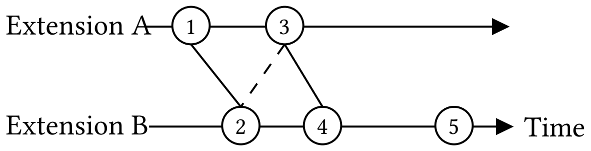


Figure B.3: Example of two extensions with reviews and the connections ATW makes. The dotted line shows non-maximizing connections that discrete optimization will remove.

one-to-one, we perform a second filtering. Similar to the first filtering, we ensure that the connected reviews make up a significant portion of the total reviews.

C.3.2 Horizontal Vertical Clustering: A Machine Learning Approach

Inspired by previous research comparing timeseries [39], we implement a ML-based approach. This approach involves clustering reviews into what we call "horizontal clusters" for reviews in the same time series of one extension and "vertical clusters" for clustering multiple time series/extensions using the centroids of the horizontal clusters produced before. Intuitively, a horizontal cluster represents a burst of review activity for one extension, and a vertical cluster represents a shared burst for multiple extensions. After this core idea, we denote our method, Horizontal Vertical Clustering (HVC). Similar to ATW, the focus is on temporal data, allowing us to detect attacks using disjoint sets of accounts, as explained in Section C.2.

Specifically, we use the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm for horizontal and vertical clustering. DBSCAN [5] is an unsupervised clustering algorithm that groups based on an epsilon hyperparameter, which is the local radius for expanding clusters. In this method, epsilon is the threshold for the maximum time distance between reviews, or centroid of review clusters, within a group. This group consists of either reviews for a single extension or centroids of multiple extensions. It is worth mentioning that DBSCAN has been previously used in malware analysis [26] and clustering browser extensions for malware detection [37].

While DBSCAN performs well in its role as the clustering algorithm for this method, we are not taking advantage of some of its unique characteristics, such as finding arbitrarily-shaped clusters. Therefore, we believe that it can be substituted for other algorithms.

In Table B.2, we include a real example illustrating how we use HVC to form clusters of extension reviews horizontally and vertically. In this example,

Table B.2: Example of a vertical cluster DBSCAN produces together with its horizontal clusters. We use 0.0001 and 1.5e-06 as the epsilons for the horizontal and the vertical clusters, respectively. All the reviews of this table were written on the same day (2023-01-11) between 09:46:42 and 10:44:15.

Extension	Username	Time	Centroid
C[7]	Dennis	09:46:42	10:11:33
	Yuriy	09:57:52	10:11:33
	William	10:08:38	10:11:33
	Аркадий	10:23:55	10:11:33
	Jamie	10:40:36	10:11:33
B[6]	Dennis	09:48:46	10:13:19
	Yuriy	09:59:13	10:13:19
	William	10:10:18	10:13:19
	Аркадий	10:25:16	10:13:19
	Jamie	10:43:03	10:13:19
A[5]	Dennis	09:49:59	10:14:38
	Yuriy	10:00:31	10:14:38
	William	10:11:53	10:14:38
	Аркадий	10:26:32	10:14:38
	Jamie	10:44:15	10:14:38

we consider a vertical cluster comprising of three extensions A[5], B[6], and C[7]. We first create the horizontal clusters, i.e., grouping reviews written close enough in time per extension. This forms a summary of the review activity for a single extension. For instance, we can see that HVC clusters all the reviews within a timestamp of over 30 minutes (i.e., 9:46:42 and 10:40:36) of the C[7] extension in the same horizontal cluster (see “Horizontal Cluster” column of Table B.2).

After that, we compute the centroid (Horizontal Centroid in the table), which serves as the datetime value for clustering extensions vertically (see Vertical Cluster column). Interestingly, in this example, the centroids of the reviews for these three extensions are within a radius of less than two minutes (0:01:32.3). We also include a graphical representation of the same example in Figure B.4.

C.3.3 Co-Reviewer

This method identifies connections between accounts that frequently review the same extensions, regardless of when this shared activity occurs in bursts.

[5] geokkpbkfpghbjdgbganjkgfhaafmhbo
[6] mpiihicgfapopgaahidedijlddefkedc
[7] lgjdgmdbfhobkdbcjnpnlmhnplnidkkp

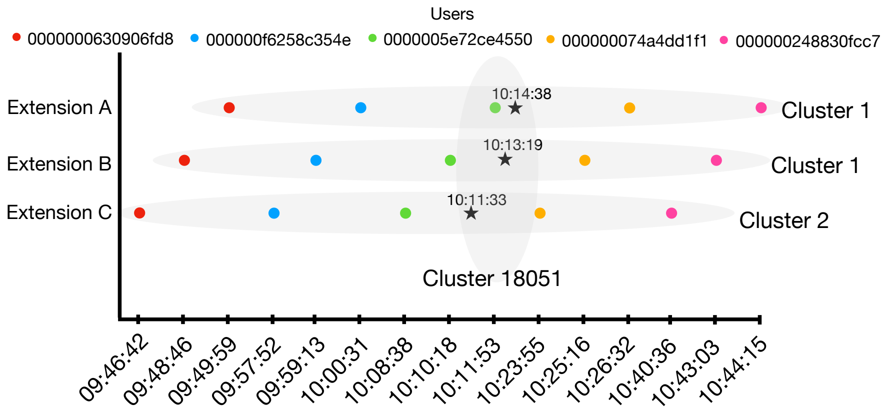


Figure B.4: Example of five reviewers of three real extensions A[5], B[6], and C[7] that DBSCAN groups in the same vertical cluster (18051) using 0.0001 and 1.5e-06 as the epsilons for horizontal and vertical clustering. We mark the centroid of every horizontal cluster with a star.

This approach helps uncover clusters of fake accounts, a technique we discuss in Section C.2, which manipulates the reputation of a common set of extensions. The primarily targeted model of reputation manipulation for this method is *reviews campaigns*, detected when accounts are reused and review several heavily overlapping extensions.

In contrast to ATW or HVC, we preprocess the data here to filter out all accounts with only one written review. Since many reviewers only write one review (see Figure B.2), we also improve runtime performance by removing them early in the process.

After this preprocessing, the algorithm iterates through each account and the extensions they reviewed, calculating the overlap between the current account and other accounts that reviewed the same extensions. This process establishes the degree of overlap between accounts. We use a threshold to form clusters of accounts with a high degree of overlap. Based on these clusters of accounts, we finally extract the list of shared extensions they reviewed.

C.3.4 Spam Detection

Spam detection aims to find attackers who post as many reviews as possible in a very short time. For example, an extension might get a legitimate review once a day but then get ten reviews in just three minutes. Our spam detection method aims to detect this type of attack. To achieve this task, we define the time between two consecutive reviews in an extension as Δt. Then, for every pair of consecutive reviews, we mark them as suspicious if the Δt is less than a threshold, which we set to three minutes in this study. In Appendix C.3, we look closer at the general distribution of Δt for all extensions and further motivate our choice of threshold.

C.3.5 Written Ratio

This method leverages users' choice to leave a rating or attach text to their review. Since it takes extra effort to write text, we suppose that not all users who leave a rating will also write a review. We detect some abnormal review patterns by analyzing the ratio between the written reviews and ratings. An example of this type of attack is on the extension "D365-UI-Test-Designer"[8]. This extension has 141 ratings, all of which include written reviews, resulting in a written ratio of 100%. As such, this method will report extensions with a suspiciously high fraction of written reviews.

C.4 Evaluation

We implement FakeX in Python and deploy the framework on a Windows computer using an AMD Ryzen 5 5600X CPU and 32GB of RAM. In this section, we present the results of FakeX together with our analysis and insights.

We crawled the Web Store as of February 9th, 2023. In total, we collected the extension's name, ID, and all written reviews of 1,782,702 reviews across 55,107 extensions (out of a total of 115,124 extensions). For every review of an extension, we have associated metadata: user's name, user's ID, review text, rating, timestamp of the initial review, and timestamp of the latest modification.

C.4.1 Aggregated Time Window

The ATW method uncovers 59 clusters with three or more extensions, with an exceptionally high number of temporally shared reviews. For this evaluation, we use a burst length of 60 minutes. In Table B.3, we present the number of

[8]lfcoehhlodiaehjepemaogbgadfoipog

Burst (min)	Clusters	Extensions
1	14	29
2	35	81
3	47	124
4	55	156
5	69	189
10	85	243
15	92	282
20	115	329
30	133	387
45	154	458
60	176	520

Table B.3: Number of ATW clusters and included extensions for different bursts.

Table B.4: Visualization of extensions with temporally shared reviews. The second and third columns represent the time and author of each review of Ninja Cut Unblocked, and the same goes for columns four and five, respectively, for X-Trial Racing Unblocked.

Date	Ninja Cut Unblocked		X-Trial Racing Unblocked		Delta
	Time	Reviewer	Time	Reviewer	
05/01	15:47:21	Caden	15:49:37	Patricia	2m 16s
16/01	10:32:32	Tracey	10:33:46	Monika	1m 14s
17/01	15:53:06	Lea	15:54:21	Ahsan	1m 15s
23/01	15:23:44	Hobart	15:24:36	Hobart	52s
24/01	19:02:13	Claire	19:03:34	Bernadette	1m 21s
02/02	17:35:35	Mason	17:36:58	Mason	1m 23s
07/02	15:14:24	Aroni	15:15:36	Aroni	1m 12s

clusters, including small clusters with only two extensions and extensions for different burst lengths. As we allow for a larger burst length, reviews farther apart in time can be clustered, and as such, the number of clusters increases. Using an excessive burst length will result in less precise results, including false positives.

To help visualize the context of temporally shared reviews, Table B.4 shows the timestamp and the username from each review of two separate extensions[9]. Note that while some shared reviewers exist in this example, ATW would still cluster the extensions even if the reviewers were entirely different. In this example, the two extensions share all their reviews temporally, with less than 3 minutes between every correlated review.

[9] iehblepfbknonbbncbidbaggofaomjop, ebllbagoalbkholngmhdlbcgfjhapdpk

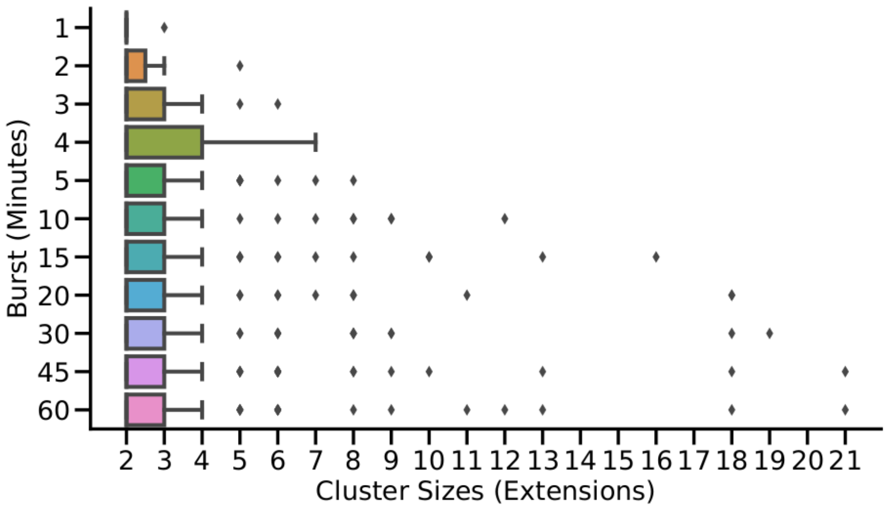


Figure B.5: Cluster size (X-axis) of ATW instances based on different burst thresholds (Y-axis). Cluster size is in terms of extensions included.

To better understand the impact of burst length on the number and size of ATW clusters, we plot the sizes of clusters for different burst lengths in Figure B.5. The highest density of clusters is always around 2-4 included extensions regardless of burst length. However, as the burst length increases, the largest clusters increase, as expected. This is shown in the figure by the increasing number of cluster size outliers. Increasing the burst period naturally lowers accuracy, though this is not a problem in the shown bursts. We can see in Appendix C.3 that the review density for extensions in this dataset should, on average, be far more than our max burst length of one hour. Because of the low density of reviews, i.e., long time between reviews on average, the probability that reviews are written in short succession across multiple extensions is very low.

In Appendix C.2, we include two examples of real extensions that ATW detects and clusters together.

C.4.2 Horizontal Vertical Clustering

The HVC method, configured with a horizontal epsilon of 0.005 and a vertical epsilon of 5e-06, generates 69,618 vertical clusters. As mentioned in Section C.3.2, these vertical clusters can be interpreted as a shared burst of review activity for multiple extensions. This shared burst could be part, one execution, of a larger review campaign. Therefore, these clusters can be repeated—some extensions will be repeatedly grouped together.

Of these 69,618 vertical clusters, 30,344 are unique (not repeated). Of the 55,107 extensions, 11,226 (20.4%) are in these clusters. The unique clusters have an average membership of 2.09 extensions, with 6,241 clusters including more than two extensions. Of those clusters with more than two extensions, they have an average membership of 3.44.

Manually analyzing these 30,444 unique clusters, or even only the 6,241 clusters with more than two extensions, is infeasible. However, knowledge of some extensions being repeatedly grouped together can inform which extensions we select for analysis from these clusters. One simple way to select extensions is to pick those that most frequently occur. This can be interpreted as these extensions being most frequently manipulated in review campaigns.

Another way, which we ended up using, is to select maximal clusters—those that are not subsets of other clusters. The intuition here is that this filters out the noise of popular extensions being included in the clusters which otherwise describe a review campaign's single execution. Both extension or cluster selection mechanisms yield similar results, with the maximal cluster selection having slightly better quality in our opinion. Selecting maximal clusters with lengths greater than 2 yields 5,585 extensions, comprising 10.1% of the extensions in the entire dataset.

Even with smaller time epsilons, some interesting patterns of relationships between reviews that indicate fake reviews are evident in these clusters. In particular, we see in Table B.5 that clusters often rediscover the relationship of extensions sharing a common developer (we share the extension IDs in Appendix C.5). These relationships can be found even when the extensions have a different scale in the number of reviews or have many reviews. This ability is unique to HVC among our methods (see Section C.6.2). At the same time, HVC is liable to false positives due to coincidental reviews of popular extensions—in this example, a popular grammar and spelling checker plugin, unlikely to have been part of the review campaign with some obscure extensions by the same developer, is included in a cluster.

C.4.3 Co-Reviewer Analysis

The Co-Reviewer analysis results in a total of 275 clusters. As we see in Table 12 (see Appendix C.1), only 9% of reviewers post more than one review, making the natural occurrence of these larger clusters of co-reviewing accounts uncommon.

This method produces clusters of extensions and reviewers similar to ATW. We can see the clusters generated by this method in Figure B.6. The

Table B.5: A selection of HVC clusters that re-discover relationships that indicate fake reviews, namely a shared developer. These fake reviews are correlated even when extensions have many reviews and different scales of the number of reviews. With horizontal and vertical epsilons of 1e-06 and 5e-06.

Cluster	Extension name	Dev	Reviews
1	Grammar and Spelling checker…	Ginger	667
1	YT Thumbnail Downloader	Sagor	26
1	Ultimate Auto History Cleaner	Sagor	20
2	Share Google Contacts with…	GAPPS	84
2	Share Google Contacts Plugin	GAPPS	48
3	Aliexpress Search by image	ganes	227
3	Aliexpress Seller Check	ganes	183
4	SelectorsHub	Sanjay	903
4	SelectorsHub Pro	Sanjay	5
5	SelectorsHub	Sanjay	903
5	TestCase Studio	Sanjay	87

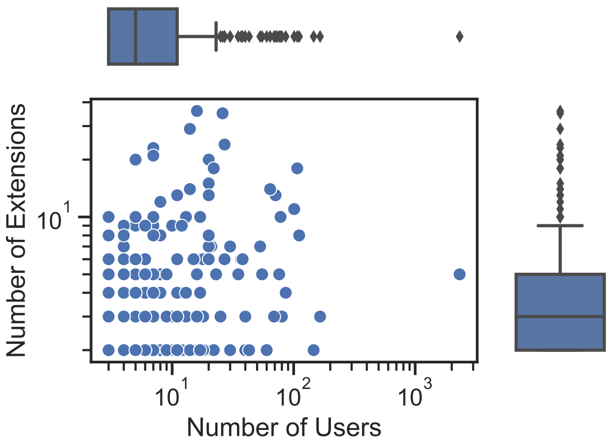


Figure B.6: Clusters generated by ATW. Every point represents a cluster of a given number of extensions and reviewers.

Table B.6: High scoring CoR cluster with 76 reviewers and five extensions. The table also includes the number of reviewers from the cluster that reviewed the extension and the ratio of all reviewers included in the cluster.

Extension name	Reviews from cluster	Ratio
Search by Image on Aliexpress	73	96.05%
Just vpn	71	93.42%
Search by Image on Alibaba	70	92.11%
Product search by image	69	90.79%
Boomtubes	35	46.05%

graph shows that the average cluster size is still very low, like the ATW results, but there are substantially more instances of large clusters. There are also generally more reviewers and extensions in the clusters of CoR. We hypothesize this is due to it being a more common attack that is also easier to detect. In the visualization, we also include the number of reviewers in clusters—notice that there are many reviewers in many clusters, emphasizing that this is a widely used attack technique. There is one extreme case of the massive outlier cluster in terms of included reviewers, that is a cluster containing 2,322 reviewers. This case results from these 2,322 reviewers mainly co-reviewing three extensions relating to the "Sui" cryptocurrency.

As expected, CoR analysis results reveal some overlap with the ATW method, as coordinated reputation manipulation on the same accounts across extensions creates patterns that both CoR and ATW discover in their clusters. ATW clusters occur when reviewers manipulate reputation simultaneously, creating a temporal correlation between their accounts.

In Table B.6, we present a high-ranking cluster with 76 reviewers and five extensions. One of the reviewers in this cluster is "Mark", who reviewed all of the top four extensions in the table in only three minutes, which we regard as extremely fast and suspicious. Interestingly, the fifth extension in the table, "Boomtubes", was also reviewed by Mark three weeks later and has a much lower ratio than the other extensions. This could indicate that this cluster is comprised of two separate review campaigns, using slightly different sets of reviewers.

C.4.4 Spam Detection

The spam detection method uncovers 86,894 reviews, about 4.9% of all reviews, which are within three minutes of each other. In Table B.7, the top ten extensions containing spam reviews are shown, where the method is run with the threshold of three minutes, meaning that every time the extension

Table B.7: Reviews detected as spam when executed with a threshold of three minutes. The ratio column shows the ratio between spam reviews and total reviews.

Rank	Name	Spam Reviews	Ratio	Rating
1	Ethos Sui Wallet	10,250	80%	5.0
2	Sui Wallet	4,095	79%	5.0
3	Swash	3,790	76%	4.8
4	Price Tracker...	3,752	68%	4.7
5	Glass wallet ...	3,713	55%	5.0
6	Fewcha Move Wallet	2,491	49%	5.0
7	Adobe Acrobat: PDF ...	2,317	14%	4.3
8	FlipShope - Price Tracker ...	1,961	45%	4.6
9	Morphis Wallet	1,832	75%	5.0
10	Bitfinity Wallet	1,672	74%	4.9

receives a review within three minutes after the last review was submitted, the spam count is increased by one. These numbers are incredibly high, a view supported by the vast top density. Also, notice that in the top ten, there are almost exclusively crypto-related extensions and price trackers, besides Adobe Acrobat. Adobe has a high number of spam reviews but a lower ratio compared to the others. Still, it is quite high compared to other popular extensions on the Web Store, e.g., NordVPN[10] (1%), MetaMask[11] (0.5%), and Skype[12] (0.5%).

In Appendix C.4, we include a visual example of how spammed reviews look on the Web Store, including the timestamp of the reviews. We also highlight the rating distribution between spam reviews. The vast majority of spam reviews leave a rating of 5.

C.4.5 Written Ratio

Figure B.7 depicts the distribution of written ratios, illustrating how unlikely extensions are to have these high ratios, especially considering how many reviews they have. The x-axis indicates the number of reviews the extension has to have strictly above to be included in the subset. For example, the first distribution includes all extensions. A big spike at the top indicates that many extensions have close to a 100% ratio. This is mainly due to extensions with few reviews, explaining the spikes at 33%, 50%, 66%, and 100% when including all extensions. However, as in the other distributions, the distribution quickly moves to the bottom for extensions with more reviews. The data reveal that

[10] fjoaledfpmneenckfbpdfhkmimnjocfa

[11] nkbihfbeogaeaoehlefnkodbefgpgknn

[12] lifbcibllhkdhoafpjfnlhfpfgnpldfl

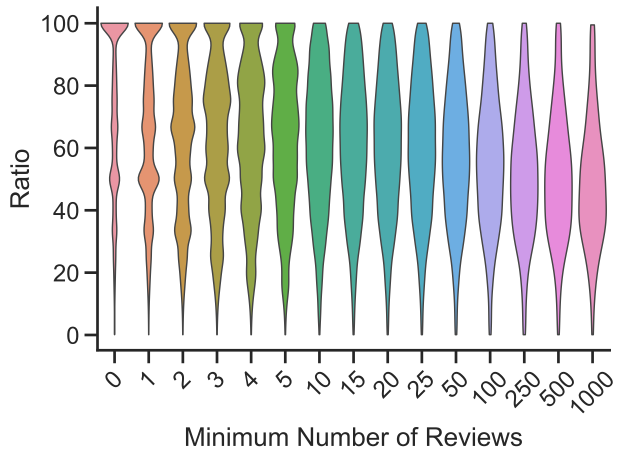


Figure B.7: Relationship between thresholds (on number of reviews) and written ratios.

extensions with a 100% written review ratio are highly improbable to occur naturally, especially in the subsets of extensions with above 25 reviews.

Table B.8 presents the total ratings, written reviews, and written ratio for the top 10 scoring extensions. Every single one of the selected extensions has a written ratio of 100%. The “Percentile” column shows at what percentile the specific extensions rank if we exclude all extensions with strictly fewer reviews and compare the written reviews between the remaining extensions. For example, the top row means that no extension exists with the same number of reviews or more with the same written ratio, 100%.

Many cases in Table B.8 present other types of suspicious behavior or other indications of being fake reviews. For example, in both “D365-UI-Test-Designer” and “DigiNovo screen sharing for A1 shop” *all* reviews contain the exact same review text, “I like it!”. Many of the other ones show other signs of fake reviews, for example, a large number of reviews in a short time period.

Consider the distribution data shown in Figure B.7—while extensions with only a few reviews can, and do, have high written ratios, extensions with many reviews should not consist of solely written reviews. Given this distribution, the near-100% written reviews of the extensions with 100s of reviews in Table B.8 should be exceptionally rare. Every single extension in Table B.8 is at least in the top sub-one percent of their respective threshold in Figure B.7.

Table B.8: Top 10 scoring extensions by the Written Ratio method with their total ratings, written ratio, and percentile.

Name	Ratings	Written Ratio	Percentile
Opened or Not - Free Email...	705	100%	100.00%
TwitterScan - Find NFT Gems...	384	100%	99.96%
D365-UI-Test-Designer	141	100%	99.96%
DigiNovo screen sharing for...	136	100%	99.94%
AliExpress Search By Image...	126	100%	99.93%
Cashback beruby	116	100%	99.91%
RippleHouse	103	100%	99.87%
Jetstream	103	100%	99.87%
BROSH for LinkedIn and Gmail	102	100%	99.87%
Marucast Desktop Capture	99	100%	99.86%

C.5 Malicious Extensions

In this section, we explore the relationships between extensions with fake reviews and their maliciousness. We both present qualitative examples and a more quantitative case-study of the clusters generated by ATW. While we would want to analyze the maliciousness of all extensions, generating this ground truth is prohibitively slow and labor intensive.

C.5.1 Security Analysis

To evaluate the correlation between fake reviews and malicious extensions we first need a ground truth of malicious and benign extensions. While there are a plethora of malicious actions extensions can perform, we limit our analysis to the following attacks.

1) *Query stealing* [9]. This attack steals users' search queries from popular engines, either by presenting a search bar in a new tab or injecting code into search engines. A common pattern is that the attacker's search form will lead to a third-party server, which in turn redirects the user to the real search engine.
2) *History stealing* [9]. This attack focuses on tracking every URL a user visits. For example, by injecting code that fetches data from a third-party server on every URL.
3) *Affiliate fraud* [25]. In this attack, attackers try to make money when users shop online. If a user buys something on, for example, Amazon.com, a malicious extension might use *cookie stuffing* to give the extension developer a commission on any purchase.
4) *Survey scams* [8]. Survey scams force or trick users into completing online surveys in order to use a service. The surveys usually collect personal data

Table B.9: ATW cluster containing “New Tab” extensions. A *connected review* is a review that happened within the same burst as another in the cluster.

Extension Name	Total Reviews	Connected Reviews
SimpleTab	11	11
TopTab	10	8
NWTab	10	7
Handy Tab	9	6
Summer Tab	10	6
Amazing Tab	10	6
ToDoTab	10	6
Charming Tab	11	6
AmTab	10	5

while tricking the user into paying for a “prize” and stealing their credit card number.

We choose these attacks because they were explored in previous works and are relatively straightforward to detect. More specific attacks, like stealing information from social media sites, are difficult to detect as they might require valid accounts.

We manually analyze extensions by inspecting their source code and executing them to ensure malicious behavior exists. This task is labor-intensive, averaging over 10 minutes per extension. Since we are analyzing clusters, in most cases, the first extension takes longer to review. We can then generate a code signature to identify other extensions exhibiting similar characteristics. In total, we manually analyzed 299 extensions for malicious behavior.

C.5.2 Case-study of ATW clusters

To better understand the relationship between fake reviews and maliciousness, we perform a security analysis of the 286 extensions in the 59 clusters found by ATW. We detect 12 suspicious clusters (ratio of malicious extensions above 80% in Table B.10). After a manual analysis, we find a cluster composed of “New Tab” extensions (see Table B.9), which are notoriously malicious and often involve query stealing [9]. This confirms that ATW detects malicious extensions.

In Table B.10, we report on the clusters with three or more extensions that ATW finds using bursts of 60 minutes. Interestingly, we note that many clusters are either strictly benign or strictly malicious. This indicates that review campaigns for malicious extensions do not mix with review campaigns for benign extensions. Furthermore, this supports ATW’s ability to find meaningful clusters of related extensions.

Table B.10: Number of clusters and percentage of malicious extensions in the clusters. From ATW using 60-minute bursts.

	Maliciousness						#Total
	0%	0%-25%	25%-50%	50%-75%	75%-100%	100%	
#Clusters	39	2	1	4	2	11	59
#Extensions	180	17	3	19	22	45	286

C.5.3 New Tab Clusters

We further explore the extensions marked by all methods to find new malicious ones. We focus on a particular breed of extensions known for malicious behavior, the “New Tab” extensions [9]. These hijack the browser’s home tab, replacing it with an alternative that modifies its functionality and appearance. In many cases, they also maliciously steal the users’ search queries by redirecting traffic to their servers before redirecting them back to a real search engine.

Currently, there are 111 extensions flagged by all our methods. Among them, 13 are classified as “New Tab” extensions and, consequently, subjected to a thorough manual inspection. Astoundingly, each of these 13 turns out to be a query stealer. This discovery led to a further exploration of the clusters these culprits were associated with, according to the ATW and CoR methods. Following an exhaustive analysis of these clusters, the tally of malicious query stealers swells to 26. Interestingly, these extensions are dispersed across four distinct clusters, with the largest cluster harboring 16 of the 26 extensions.

C.5.4 Large Malicious Cluster

We look closer at the largest ATW cluster, composed of 18 extensions, of which 17 are malicious. All the malicious extensions use the same attack, namely *malicious surveys*. These are web pages that look like genuine surveys but trick the user into expensive subscriptions or malicious file downloads. For extensions, they also act as “human validation” needed before exposing malicious behavior.

Using FakeX, we find the game extension “Bloons Tower Defense Unblocked”[13] that was flagged by all methods except *Spam Detection*. We manually verify that the extension presents users with a survey from `stallmobiles.com`, which is part of a malware list [44]. However, the extension uses one level of redirection by first loading the `http://gameunblocked.pl/bloonstdgame-`

[13]monljmeefnongjlfefogaoldojpchhpg

`newtab` web page, which redirects to `stallmobiles.com`, making static code analysis harder. Interestingly, the only extension in this cluster that is not malicious had the same code structure, but no URL.

C.5.5 Expanding from Known Malicious Extensions

Finally, we demonstrate that FakeX can be used to expand a list of known malicious extensions. We collaborate with Adblock Plus and Avast on this particular deployment of FakeX. Utilizing a list of newly discovered 18 popular yet malicious extensions from Adblock Plus [36], we compare it against the results from ATW and CoR.

Interestingly, at least one of our methods flagged 16 of the 18 extensions. On delving into the clusters in CoR and ATW, we find that the union of clusters having at least one of these 16 extensions comprises 40 extensions. We present this list to Adblock Plus for further analysis. Based on this, Adblock Plus finds and confirms by Avast that 16 more extensions contain similar malicious code [36]. Furthermore, 8 of these used an improved version of the malicious code, compatible with the new manifest v3 [35]. Adblock Plus publicly acknowledged [36, 35] our contribution to discovering the additional 16 malicious extensions, and Google subsequently removed all of these extensions from the Web Store.

C.6 Discussion

In this section, we compare our methods, exploring their implications and relevance in the broader context. Additionally, we address the limitations inherent in our study, providing transparency about potential constraints and avenues for future research.

C.6.1 ATW and CoR

The main difference between ATW and CoR is that CoR primarily detects accounts with many reviews. In contrast, ATW often detects new accounts with only one singular review and strong temporal connections within clusters of unique accounts.

Since both ATW and CoR create clusters, we want to explore if overlaps in these clusters can help us find new malicious extensions. Since we already generated the ground truth for the ATW clusters, we search for CoR clusters that overlap with the ATW clusters. We present one such cluster in Table B.11. If only ATW were used, only three malicious extensions would be found. However, combining ATW and CoR allows us to find three new extensions,

Table B.11: A cluster of malicious extensions found by CoR compared with a subset found by ATW.

Extension	ATW	Malicious
Film Links Now | Default Search		✓
Autumn Tab	✓	✓
Primary Tab	✓	✓
Tasks Area		✓
Black Tab	✓	✓
Age Calculator		✓

which manual analysis confirms to also be malicious. This shows the power of combining the methods to find more malicious extensions.

C.6.2 Comparing ATW and HVC

Since both ATW and HVC base their clustering on temporal data, a comparison of the two methods is valuable. The methods are not directly comparable as ATW combines reviews from multiple time windows while HVC only considers two time radii.

In Table B.9, ATW detects a cluster of nine extensions with fake reviews. However, it does miss the "NiceTab StartPage" [14] extension that HVC finds in a cluster together with "Charming Tab" [15]. ATW misses this extension because compared to the other nine in the cluster, this one had multiple reviews before the shared review campaign. Since ATW only clusters extensions with a significant overlap in reviews, it is not included. In general, ATW has difficulty clustering extensions that already had many reviews before getting fake reviews in the course of a review campaign. Reducing the threshold for the needed overlap could decrease the number of false negatives and allow ATW to cluster all ten of these extensions. However, it might also increase the false positive rate and possibly add unrelated extensions to this cluster.

On the other hand, HVC does not cluster the other nine as a separate cluster. This is because other popular extensions, like NordVPN [16], also got reviews within only 20 minutes of the other new tabs. A general pattern we note is that ATW is often more precise in its clustering, with the downside of occasionally missing some extensions HVC finds in its clusters.

[14] dobmhnlkolhhklmcaodfefhejoonalni
[15] kbnpeiabjlfcakokkpbcgalbgiljoddf
[16] fjoaledfpmneenckfbpdfhkmimnjocfa

Goals of ATW and HVC - FP vs FN

While we do not have labeled fake review data to train a model on, we can still compare these two methods for detecting fake reviews based on related metrics. One metric can be malware labeling performance—which extensions containing malware are flagged by each method?

An obstacle is the lack of labeled data. A relatively independent labeling is that used by the Chrome browser—extensions can be labeled “malware”, which will affect their ability to be loaded into the browser. Conversely, Chrome also has a notion of “good” extensions. The Chrome browser can allow some extensions in ESB (Enhanced Safe Browsing), which can be interpreted as being “safe” extensions. Using these labels provided by Chrome, we label “true positives” as being on the malware list, and “true negatives” as being on the ESB allowlist. False positives by this metric are extensions flagged by ATW or HVC that are not included in the Chrome malware list. Similarly, false negatives are extensions flagged by ATW or HVC that are considered safe by Chrome—on the ESB allowlist.

Metrics based on this labeling are conservative—we can establish some floor on false positives and false negatives. These Chrome-provided labels still do not cover the entirety of our dataset of extensions with reviews. Furthermore, these metrics are not an accurate description of performance.

This is plotted in Figure B.8. The trends of ATW and HVC illustrate our previous observation of ATW having more accurate clusters—it has low false positives. When scanning for malware, a low false-positive rate can be valuable. ATW provides this low false-positive solution, while HVC can be used for better recall.

C.6.3 Focus on metadata

In this work, we solely focus on *metadata* to detect fake reviews of browser extensions in the Chrome Web Store. While *content* can also be used to detect fake reviews (see Section C.7), we argue that detection mechanisms reliant on content are more easily eluded. Content can be faked easily and cheaply — fake review authors can copy existing review text, or generate them with a variety of methods. Metadata can also be faked, including with the fake review techniques discussed in Section C.2. However, timestamps and user relations are harder and more expensive to fake, in that obscuring these temporal and user connections requires more time and user profiles to conduct a review campaign.

This metadata is not unique to the Chrome Web Store, and the fake review detection techniques we propose should be applicable to other online

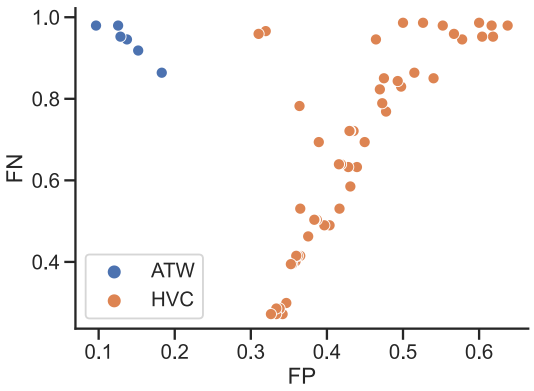


Figure B.8: Performance of ATW and HVC according to Chrome malware labels, with each data point being a different parametrization of the respective methods. In general, ATW provides a low false-positive solution, while HVC offers low false-negatives.

marketplaces. However, the timestamps utilized by the ATW, HVC, and Spam Detection methods are not always readily available - we note that the Yelp [23] and Amazon [34] datasets only have coarse date precision for their review timestamps.

C.7 Related Work

Browser extensions. Researchers explore methods for detecting malicious extensions, ranging from scrutinizing downloads [38] and dynamically analyzing extension behavior, including information sent to external parties [9], to examining the sequences of API calls of common malicious actions [1]. Furthermore, studies have explored trends and values, detecting anomalous ratings [37], and monitoring extension executions to identify content modifications, such as injecting advertisements [2]. Static analysis to detect malicious JavaScript code could also be repurposed to detect malicious extensions [11]. Conversely, extensions have also served as vectors for attacks, i.e., exploiting some vulnerabilities in the extensions' source code, enabling activities like user tracking [49, 47, 48], acquiring sensitive information such as user history [6, 12], and facilitating remote code execution [46, 12]. Static analysis has been employed to discover such vulnerabilities [53, 13].

Pantelaios et al. [37] analyze anomalous extension ratings, code changes, and keyword pattern matching. However, their methods are limited to extensions with at least 50 reviews. FakeX focuses on the time component rather than the content of the reviews, and therefore, has no limitation on the number of reviews.

Despite the significant progress in browser extension security [24, 43, 9, 25], the orthogonal focus on user reviews and reputation manipulation in the Web Store, as highlighted in our research, represents a promising and fruitful direction that complements the prior studies.

Fake reviews. Fake review detection has been studied in other marketplaces, where users can also post reviews of products. However, prior methods in this area are often supervised and require ground truth labeling. This is not available for our application to extensions in the Web Store. Furthermore, the labeled datasets used in prior works can lead to biased results and methods. For example, datasets derived from Yelp business reviews are commonly used [4, 33, 22]. These methods assume that the user-submitted reviews Yelp does not label as "Trustful" are fake. This can introduce a bias towards learning the techniques Yelp uses internally for its filtering. Other approaches [10, 20] solicit fake reviews through Amazon Turk. This could also introduce a bias towards detecting fake reviews produced with a specific generating system. Unsupervised methods, such as FakeX, avoid being biased in this fashion.

Despite these difficulties with either applying previous fake-review detection approaches to the browser extension ecosystem, or evaluating their performance, these works are still useful to compare to the methods of FakeX. Fake review detection approaches can be grouped by the data used. Some methods use reviewer networks [40, 22, 28], similar to the CoR analysis in this paper. Methods using timestamps [29] have some similarities to ATW, HVC, and Spam detection in this paper. The Written Ratio method uses review text (in its presence/absence), similar to [33, 20]. Finally, some methods also combine multiple features [10, 4, 32] as FakeX does. Despite these surface similarities, FakeX offers a novel unsupervised framework for detecting fake reviews with methods that are suited to finding malware in browser extensions, primarily using temporal review graph features. We expand on this by comparing it to the mostly closely related works below.

Rathore et al. [40] obtain partial ground-truth information about fraud reviewer IDs by soliciting fake reviews for applications on the Google Play Store, using Fiverr, a platform that connects freelancers to people or businesses looking to hire. While a partial ground truth would be valuable to informing both our method and evaluation, soliciting fake reviews could lead to bias in the data. Furthermore, buying fake reviews will not provide useful temporal data, necessary for the central ATW and HVC methods of FakeX.

Li et al. [28] also use partial ground-truth, assuming fake review labels from the review-hosting site Dianping have high recall. This enables a Positive and Unlabeled (PU) learning approach, where the reviewer graph with 'fake reviewer' labels is iteratively extended from an initial set, using the association of shared IPs. The Web Store does not offer either fake review labels or user IP addresses.

He et al. [22] form a ground truth about Amazon product reviews by monitoring Facebook groups that act as marketplaces for buying and selling fake reviews. From these groups, they identify which products buy fake reviews and train a model on the review network to detect these. Given the absence of evidence linking concrete buyer-seller networks, such as Facebook groups, to fake reviews in the Web Store, investigating such a relationship emerges as a potential avenue for future research. Unlike the focus of this paper, FakeX employs CoR analysis to identify highly clustered fake reviewer networks, while ATW or HVC can uncover more nuanced relationships that exploit disjoint sets of fake accounts.

The method proposed by Liu et al. [29] is perhaps the closest to our application of ATW and HVC, in that they utilize review timestamp metadata to identify anomalous review activity. The authors crawl Amazon China for review records, then apply different time windows to bucket review activity for a particular product. The authors use an unsupervised clustering algorithm (isolation forest) to identify products whose review activity is anomalous. When applying this simple bucketing approach, we were unable to recreate the interesting fake-review and malware clusters produced by FakeX. A potential issue with applying their method to extension data is the larger timespan considered. Simple bucketing will yield a list of buckets from the dataset start time to end time, most of which will be empty. Comparing these high-dimensional points is challenging. In contrast, FakeX's HVC performs horizontal clustering to ease the somewhat analogous vertical clustering task.

Barbado et al. [4] propose a Fake Review Framework (F3), which combines reviewer and review text features, and apply this to the Yelp dataset. Reviewer features are mainly about the reviewer's activity, though this does include direct relationships such as friend networks. In contrast to this work, FakeX utilizes indirect relationships by reviewing the same extension through related reviews (ATW, HVC, Spam Detection, Written Ratio) or accounts (CoR).

Mukherjee et al. [32] also combine reviewer and review text features. They use expert labels of Amazon fake reviewer groups to hand-craft a set of spam indicator features containing reviewer behavior, relationships, and content similarity. The 'spamicity' of reviewers and groups is then iteratively refined with these features. This method does seem applicable to the Web Store setting, but the base features will likely have to be adjusted when transposed from the original setting. We did not re-implement this method to evaluate, as the source code is unavailable.

Other solutions are based on the links of the reviewers, reviews, products, and merchants [10], or the sentiment analysis of the reviews [33, 20]. FakeX does not use these additional features. While the knowledge graph proposed by Fang et al. [10] can contain the review graph features used in this paper, this method is inapplicable without both a more complete understanding of the fake review ecosystem for browser extensions, and labeled data to train models to both use the knowledge graph and evaluate its complex construction. The integration of additional features into the analysis of browser extensions is a promising avenue for future research.

C.8 Conclusion

We propose FakeX, a framework for detecting fake reviews in browser extensions. FakeX leverages five methods, ATW, HVC, CoR, Written Ratio, and Spam Detection, to identify extensions with fake reviews. Our evaluation unveils hundreds of review campaigns used on the Web Store, as well as, different attack techniques used in the campaigns. In particular, we find 59 clusters across 286 extensions with fake reviews sharing temporal patterns. This positively answers our first research question, whether reputation manipulation exists on the Web Store.

While fake reviews do not necessarily imply malicious intent, they put extensions with fake reviews into the spotlight and motivate further scrutiny for security risks. This leads to the positive answer to our second research question on leveraging our methods to detect malicious extensions. Using FakeX we find a total of 86 extensions with a total of 64 million users. After reporting to Google, 44 of these extensions were removed. Finally, we collaborate with Adblock Plus and Avast to demonstrate FakeX in action, expanding a seed list of newly detected malicious extensions to discover a further 16 malicious extensions with millions of users, where in some cases attackers tried to improve malicious code.

Acknowledgements. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, the Swedish Foundation for Strategic Research (SSF), and the Swedish Research Council (VR).

Bibliography

[1] A. Aggarwal, B. Viswanath, L. Zhang, S. Kumar, A. Shah, and P. Kumaraguru. I spy with my little eye: Analysis and detection of spying browser extensions. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 47–61, Washington, D.C., 2018. IEEE Computer Society.

[2] S. Arshad, A. Kharraz, and W. Robertson. Identifying extension-based ad injection via fine-grained web content provenance. In F. Monrose, M. Dacier, G. Blanc, and J. Garcia-Alfaro, editors, *Research in Attacks, Intrusions, and Defenses*, pages 415–436, Cham, 2016. Springer International Publishing.

[3] A.-L. Barabási. The origin of bursts and heavy tails in human dynamics. *Nature*, 435(7039):207–211, 2005.

[4] R. Barbado, O. Araque, and C. A. Iglesias. A framework for fake review detection in online consumer electronics retailers. *Information Processing & Management*, 56(4):1234–1244, 2019.

[5] D. Birant and A. Kut. St-dbscan: An algorithm for clustering spatial–temporal data. *Data & knowledge engineering*, 60(1):208–221, 2007.

[6] Q. Chen and A. Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1687–1700, New York, NY, USA, 2018. Association for Computing Machinery.

[7] Chrome Extensions Stats, 2023.

[8] P. Ducklin. Anatomy of a survey scam – how innocent questions can rip you off, Jun 2020.

[9] B. Eriksson, P. Picazo-Sanchez, and A. Sabelfeld. Hardening the security analysis of browser extensions. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, SAC '22, page 1694–1703, New York, NY, USA, 2022. Association for Computing Machinery.

[10] Y. Fang, H. Wang, L. Zhao, F. Yu, and C. Wang. Dynamic knowledge graph based fake-review detection. *Applied Intelligence*, 50:4281–4295, 2020.

[11] A. Fass, M. Backes, and B. Stock. Jstap: a static pre-filter for malicious javascript detection. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 257–269, New York, NY, USA, 2019. Association for Computing Machinery.

[12] A. Fass, D. F. Somé, M. Backes, and B. Stock. Doublex: Statically detecting vulnerable data flows in browser extensions at scale. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1789–1804, New York, NY, USA, 2021. Association for Computing Machinery.

[13] A. Fass, D. F. Somé, M. Backes, and B. Stock. Doublex: Statically detecting vulnerable data flows in browser extensions at scale. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1789–1804, New York, NY, USA, 2021. Association for Computing Machinery.

[14] FreeAddon. Warning: Fake one-star reviews & ratings are bombarding freeaddon extensions!, Sep 2018.

[15] GetReview. Seo agency/company london uk | digital marketing agency in high wycombe. `https://getreview.co.uk`, Jan. 2024.

[16] R. GG. Buy positive reviews online at cheap prices on review community. `https://reviewgg.com/`, Jan. 2022.

[17] Google. Chrome web store, 2023.

[18] Google Developers. Spam and abuse. `https://developer.chrome.com/docs/webstore/program-policies/spam-and-abuse/`, 2022.

[19] Google Developers. Spam policy faq. `https://developer.chrome.com/docs/webstore/spam-faq/#ratings-and-reviews`, 2022.

[20] P. Hajek, A. Barushka, and M. Munk. Fake consumer review detection using deep neural networks integrating word embeddings and emotion mining. *Neural Computing and Applications*, 32:17259–17274, 2020.

[21] M. A. Harris, R. Brookshire, and A. G. Chin. Identifying factors influencing consumers' intent to install mobile applications. *International Journal of Information Management*, 36(3):441–450, 2016.

[22] S. He, B. Hollenbeck, G. Overgoor, D. Proserpio, and A. Tosyali. Detecting fake-review buyers using network structure: Direct evidence from amazon. *Proceedings of the National Academy of Sciences*, 119(47):e2211932119, 2022.

[23] Y. Inc. Yelp open dataset. `https://www.yelp.com/dataset`, Jan. 2024.

[24] N. Jagpal, E. Dingle, J.-P. Gravel, P. Mavrommatis, N. Provos, M. A. Rajab, and K. Thomas. Trends and lessons from three years fighting malicious extensions. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 579–593, Washington, D.C., Aug. 2015. USENIX Association.

[25] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 641–654, San Diego, CA, Aug. 2014. USENIX Association.

[26] J. Kinable and O. Kostakis. Malware classification based on call graph clustering. *Journal in computer virology*, 7(4):233–245, 2011.

[27] B. Krebs. Using fake reviews to find dangerous extensions, 2021.

[28] H. Li, Z. Chen, B. Liu, X. Wei, and J. Shao. Spotting fake reviews via collective positive-unlabeled learning. In *2014 IEEE International Conference on Data Mining*, pages 899–904, Washington, D.C., 2014. IEEE Computer Society.

[29] W. Liu, J. He, S. Han, F. Cai, Z. Yang, and N. Zhu. A method for the detection of fake reviews based on temporal features of reviews and comments. *IEEE Engineering Management Review*, 47(4):67–79, 2019.

[30] J. Matoušek and B. Gärtner. *Understanding and using linear programming*, volume 33. Springer, New York, NY, USA, 2007.

[31] R. Mohawesh, S. Xu, S. N. Tran, R. Ollington, M. Springer, Y. Jararweh, and S. Maqsood. Fake reviews detection: A survey. *IEEE Access*, 9:65771–65802, 2021.

[32] A. Mukherjee, B. Liu, and N. Glance. Spotting fake reviewer groups in consumer reviews. In *Proceedings of the 21st international conference on World Wide Web*, pages 191–200, Lyon, France, 2012. Association for Computing Machinery.

[33] A. Mukherjee, V. Venkataraman, B. Liu, N. Glance, et al. Fake review detection: Classification and analysis of real and pseudo reviews. *UIC-CS-03-2013. Technical Report*, 2013.

[34] J. Ni. Amazon review data (2018). `https://nijianmo.github.io/amazon/`, Jan. 2018.

[35] W. Palant. How malicious extensions hide running arbitrary code. `https://palant.info/2023/06/02/how-malicious-extensions-hide-running-arbitrary-code/`, 2023.

[36] W. Palant. More malicious extensions in chrome web store. `https://palant.info/2023/05/31/more-malicious-extensions-in-chrome-web-store/`, 2023.

[37] N. Pantelaios, N. Nikiforakis, and A. Kapravelos. You've changed: Detecting malicious browser extensions through their update deltas. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 477–491, New York, NY, USA, 2020. Association for Computing Machinery.

[38] P. Picazo-Sanchez, B. Eriksson, and A. Sabelfeld. No signal left to chance: Driving browser extension analysis by download patterns. In *Proceedings of the 38th Annual Computer Security Applications Conference*, ACSAC '22, page 896–910, New York, NY, USA, 2022. Association for Computing Machinery.

[39] I. Portugal, P. Alencar, and D. Cowan. A framework for spatial-temporal trajectory cluster analysis based on dynamic relationships. *IEEE Access*, 8:169775–169793, 2020.

[40] P. Rathore, J. Soni, N. Prabakar, M. Palaniswami, and P. Santi. Identifying groups of fake reviewers using a semisupervised approach. *IEEE Transactions on Computational Social Systems*, 8(6):1369–1378, 2021.

[41] T. M. Report. The menlo report. `https://www.dhs.gov/sites/default/files/publications/CSD-MenloPrinciplesCORE-20120803_1.pdf`, 2012.

[42] G. Reviews. Get reviews. `https://getreviews.buzz/`, Aug. 2020.

[43] F. Schaub, A. Marella, P. Kalvani, B. Ur, C. Pan, E. Forney, and L. F. Cranor. Watching them watching me: Browser extensions' impact on user privacy awareness and concern. In *NDSS workshop on usable security*, volume 10, Reston, Virginia, U.S., 2016. The Internet Society.

[44] ShadowWhisperer. Malware. `https://github.com/ShadowWhisperer/BlockLists/blob/master/Lists/Malware`, 2023.

[45] R. Soares, C. P. Benjamin Ackerman, and A.-A. Team. Keeping spam off the chrome web store, Apr 2020.

[46] D. F. Somé. Empoweb: Empowering web applications with browser extensions. In *2019 IEEE Symposium on Security and Privacy (S&P)*, pages 227–245, Washington, D.C., 2019. IEEE Computer Society.

[47] O. Starov, P. Laperdrix, A. Kapravelos, and N. Nikiforakis. Unnecessarily identifiable: Quantifying the fingerprintability of browser extensions due to bloat. In *The World Wide Web Conference*, WWW '19, page 3244–3250, New York, NY, USA, 2019. Association for Computing Machinery.

[48] O. Starov and N. Nikiforakis. Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 1481–1490, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee.

[49] O. Starov and N. Nikiforakis. Xhound: Quantifying the fingerprintability of browser extensions. In *2017 IEEE Symposium on Security and Privacy (S&P)*, pages 941–956, Washington, D.C., 2017. IEEE Computer Society.

[50] B. R. Store. Provide online reviews marketing five star rating & reviews services. `https://buyreviewstore.com`, Jan. 2024.

[51] R. Sub. Free or buy google reviews, amazon reviews & more. `https://www.reviewsub.com/`, 2022.

[52] B. S. World. Buy smm world - digital marketing and reviews service provider. `https://buysmmworld.com`, Jan. 2024.

[53] J. Yu, S. Li, J. Zhu, and Y. Cao. Coco: Efficient browser extension vulnerability detection via coverage-guided, concurrent abstract interpretation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 2441–2455, New York, NY, USA, 2023. Association for Computing Machinery.

C.1 Browser Extensions Reviewers

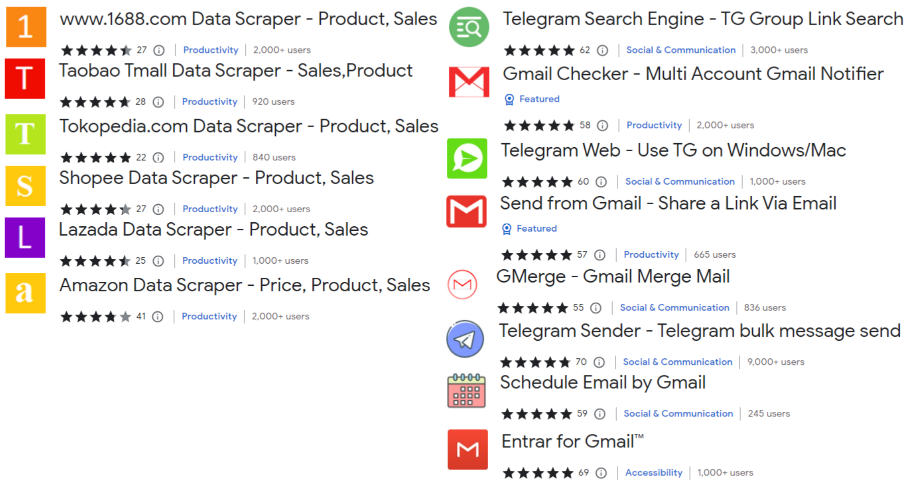
In Table 12, we include the overall number of extensions with reviews (first row), reviewers (second row), and reviews (third row). We further break down both the reviewers and the reviews they post by the type of reviewer. Reviewers are either Single, if they have posted only one review, or Multi, if they have posted multiple.

Table 12: Reviews distribution as of February 2023

	Reviewers		
Metric	Single	Multi	Total
Total Extensions			55,107
Total Reviewers	1 402 687 (91.29%)	133,819 (8.71%)	1 536 506
Total Reviews	1 402 687 (78.68%)	380,015 (21.32%)	1 782 702

C.2 Aggregated Time Window Examples

Figure 9 depicts visual examples of high-scoring clusters. Notice the similarities in amount of reviews, which are closely related in quantity. They also share a temporal pattern of when the reviews were submitted, which is why ATW clustered them.



www.1688.com Data Scraper - Product, Sales
★★★★★ 27 | Productivity | 2,000+ users

Taobao Tmall Data Scraper - Sales,Product
★★★★★ 28 | Productivity | 920 users

Tokopedia.com Data Scraper - Product, Sales
★★★★★ 22 | Productivity | 840 users

Shopee Data Scraper - Product, Sales
★★★★★ 27 | Productivity | 2,000+ users

Lazada Data Scraper - Product, Sales
★★★★★ 25 | Productivity | 1,000+ users

Amazon Data Scraper - Price, Product, Sales
★★★★★ 41 | Productivity | 2,000+ users

Telegram Search Engine - TG Group Link Search
★★★★★ 62 | Social & Communication | 3,000+ users

Gmail Checker - Multi Account Gmail Notifier
Featured
★★★★★ 58 | Productivity | 2,000+ users

Telegram Web - Use TG on Windows/Mac
★★★★★ 60 | Social & Communication | 1,000+ users

Send from Gmail - Share a Link Via Email
Featured
★★★★★ 57 | Productivity | 665 users

GMerge - Gmail Merge Mail
★★★★★ 55 | Social & Communication | 836 users

Telegram Sender - Telegram bulk message send
★★★★★ 70 | Social & Communication | 9,000+ users

Schedule Email by Gmail
★★★★★ 59 | Social & Communication | 245 users

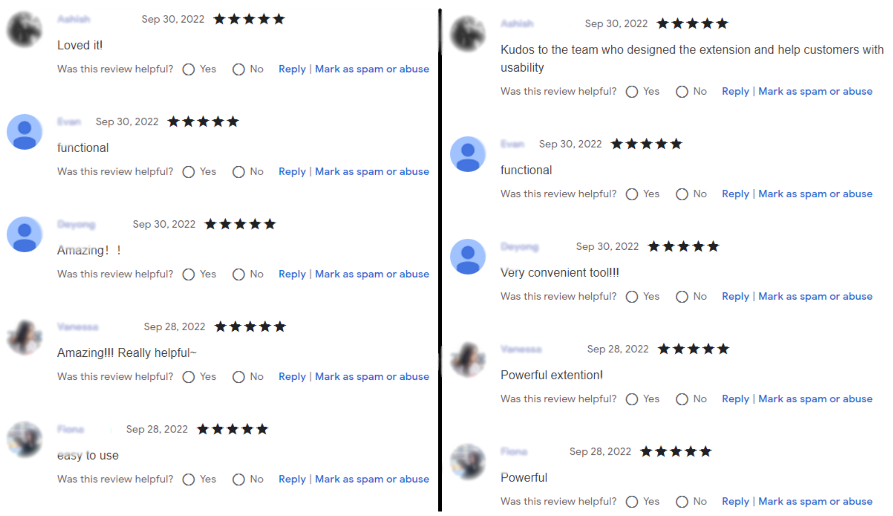
Entrar for Gmail™
★★★★★ 69 | Accessibility | 1,000+ users

Figure 9: Visual example of two clusters found by the ATW method.

Figure 10 compares reviews of two extensions from the Gmail cluster. Notice that they got reviews at the same time between extensions. In this case, it also happens to be the same reviewers, resulting in this particular pattern being clustered by CoR as well.

C.3 Review time distribution

In Figure 11 we see how reviews are distributed in time. Interestingly, the data seem to follow a log-normal distribution (the x-axis is in a logarithmic


[redacted] Sep 30, 2022 ★★★★★
Loved it!
Was this review helpful? ◯ Yes ◯ No Reply | Mark as spam or abuse

[redacted] Sep 30, 2022 ★★★★★
functional
Was this review helpful? ◯ Yes ◯ No Reply | Mark as spam or abuse

[redacted] Sep 30, 2022 ★★★★★
Amazing! !
Was this review helpful? ◯ Yes ◯ No Reply | Mark as spam or abuse

[redacted] Sep 28, 2022 ★★★★★
Amazing!!! Really helpful~
Was this review helpful? ◯ Yes ◯ No Reply | Mark as spam or abuse

[redacted] Sep 28, 2022 ★★★★★
easy to use
Was this review helpful? ◯ Yes ◯ No Reply | Mark as spam or abuse

[redacted] Sep 30, 2022 ★★★★★
Kudos to the team who designed the extension and help customers with usability
Was this review helpful? ◯ Yes ◯ No Reply | Mark as spam or abuse

[redacted] Sep 30, 2022 ★★★★★
functional
Was this review helpful? ◯ Yes ◯ No Reply | Mark as spam or abuse

[redacted] Sep 30, 2022 ★★★★★
Very convenient tool!!!
Was this review helpful? ◯ Yes ◯ No Reply | Mark as spam or abuse

[redacted] Sep 28, 2022 ★★★★★
Powerful extention!
Was this review helpful? ◯ Yes ◯ No Reply | Mark as spam or abuse

[redacted] Sep 28, 2022 ★★★★★
Powerful
Was this review helpful? ◯ Yes ◯ No Reply | Mark as spam or abuse

Figure 10: Temporally shared reviews between extensions, clustered by both ATW and CoR. Last names are removed from reviewers for ethical reasons.

scale) where the average Δt is over 19.5 days. Also notice the spike at 1, which only contains reviews with a second or less time delta, which is the most extreme case of spam. We even observe 14 cases of sub-millisecond deltas; since the Web Store has only millisecond accuracy, these have the same recorded timestamp. Approximately, 95% of the reviews have a Δt of more than three minutes, making three minutes a reasonable threshold to find abnormally fast reviewing activity.

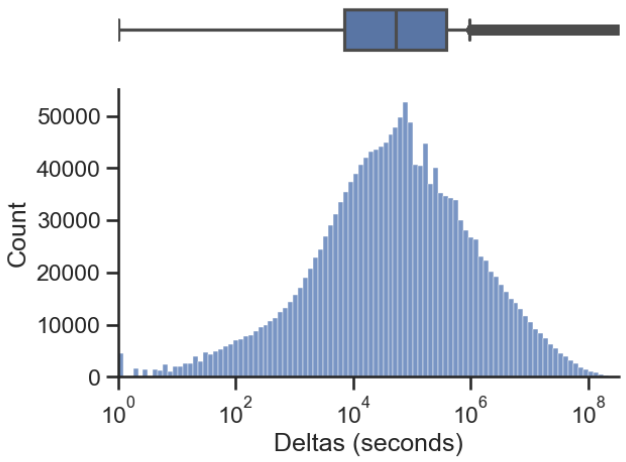
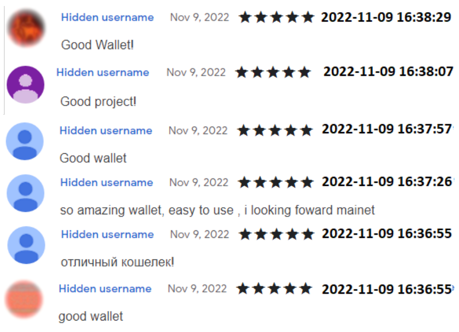


Figure 11: Data distribution of the difference in time between reviews (deltas) in an extension.

C.4 Spam detection rating

Figure 12 shows how spammed reviews look on the Web Store, including the timestamp of the reviews. Note that there are only a few seconds between each review, except for two reviews in the same second.



Hidden username Nov 9, 2022 ★★★★★ 2022-11-09 16:38:29
Good Wallet!

Hidden username Nov 9, 2022 ★★★★★ 2022-11-09 16:38:07
Good project!

Hidden username Nov 9, 2022 ★★★★★ 2022-11-09 16:37:57
Good wallet

Hidden username Nov 9, 2022 ★★★★★ 2022-11-09 16:37:26
so amazing wallet, easy to use , i looking foward mainet

Hidden username Nov 9, 2022 ★★★★★ 2022-11-09 16:36:55
отличный кошелек!

Hidden username Nov 9, 2022 ★★★★★ 2022-11-09 16:36:55
good wallet

Figure 12: Spammed reviews with timestamps on the extension *Ethos Sui Wallet.*

Figure 13 presents the distribution of ratings within spam reviews, i.e., reviews made in less than three minutes of each other. The graph shows that a large majority of the spam reviews are five star reviews, indicating these reviews are mainly used to promote extensions.

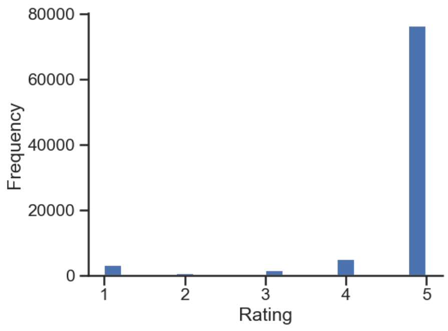


Figure 13: Rating distribution of spam marked reviews, using a threshold of three minutes.

C.5 Extension IDs for examples named in tables

In Table 13, we present all the extensions used in tables throughout the paper together with their IDs, and if we consider them malware.

Table 13: Extension IDs for examples named in tables.

Table	Extension name	Extension ID	Malware
Table B.5	YT Thumbnail Downloader	akfikgmhbajiaekdbgchbmhkceclncda	No
	Aliexpress Search by image	jkcacbjiofjgbnaknoojjboeiinempoa	No
	Grammar and Spelling checker by Ginger	kdfieneakcjfaiglcfcgkidlkmlijjnh	No
	SelectorsHub Pro	kodoloplfbnhlfcepehlafnbojbfgglb	No
	TestCase Studio	loopjjegnlccnhgfehekecpanpmielcj	No
	Ultimate Auto History Cleaner	nfnjemoofkhppjhjcehbddolbalmibkg	No
	Aliexpress Seller Check	mibmplgflabdmnnoncnedjfdpidjblnk	No
	SelectorsHub	ndgimibanhlabgdgjcpbbndiehljcpfh	No
	Share Google Contacts with Shared Contacts	nhmihkokjnmeaagjihlamgohjfmapehj	No
	Share Google Contacts Plugin	nllecbomigehlngfclbgjeghfmfajfgp	No
Table B.6	Just vpn	apmomfapnjopaiiidbockbmbkklcfgni	Yes
	Search by Image on Aliexpress	chdmkeeecofpljchimdkliaknhaibkgm	Yes
	Boomtubes	igjenkfpfgfhoaagnmbbidjfbobmkohe	No
	Product search by image	inbbmabopknohmlmilkhjdidlmbhhofd	Yes
	Search by Image on Alibaba	pamfkmlimebecnfjoikmacloehbkhhoj	Yes
Table B.7	Flipshope: Price Tracker and much more	adikhbfjdbjkhelbdnffogkobkekkkej	No
	Swash	cmndjbecilbocjfkibfbifhngkdmjgog	No
	Fewcha Move Wallet	ebfidpplhabeedpnhjnobghokpiioolj	No
	Adobe Acrobat: PDF edit, convert, sign tools	efaidnbmnnnibpcajpcglclefindmkaj	No
	Morphis Wallet	heefohaffomkkkphnlpohglngmbcclhi	No
	Price Tracker - Auto Buy, Price History	hegbjcdehgihjohghnmdpebepnoalode	No
	Bitfinity Wallet	jnldfbidonfeldmalbflbmlebbipcnle	No
	Glass wallet | Sui wallet	loinekcabhlmhjjbocijdoimmejangoa	No
	Ethos Sui Wallet	mcbigmjiafegjnnogedioegffbooigli	No
	Sui Wallet	opcgpfmipidbgpenhmajoajpbobppdil	No
Table B.8	BROSH for LinkedIn and Gmail	bhjeblnbniahjoghbcngookdjdjjllde	No
	RippleHouse	dbjdhpndplhpppleinigdfnbibilkmod	No
	Opened or Not - Free Email Tracker	dmchdoholidpalbigibcgkkifklkcnil	No
	TwitterScan - Find NFT Gems & Trending Tokens	dmlbdfmbofhfnkneodciekpgaacbgdfo	No
	Jetstream	ijancdlmlahmfgcimhocmpibadokcdfc	No
	Marucast Desktop Capture	fjfnbddkahphhfhpmgknhgfbbnbbajkh	No
	D365-UI-Test-Designer	lfcoehhlodiaehjepemaogbgadfoipog	No
	AliExpress Search By Image | Rovalty	lijlkcihmpnnaijedioieaafmghjdnca	No
	Cashback beruby	lldknhffmfbndpbknmcckoelpidapidf	No
	DigiNovo screen sharing for A1 shop	pmpmejbonomjlbhphkkbeeeecpnknkpn	No
Table B.9	NWTab	abcmjdhbopfnfkdonmkadfdghgipdeic	Yes
	Amazing Tab	agpoehmhgoieigdbjhgphpagmloehamn	Yes
	SimpleTab	ajjhojeehlipcemlodoncklkdoficgdi	Yes
	Summer Tab	dclbdlgnlaodfbjghpdjiodbnlicgalo	Yes
	AmTab	jalfhdofagnilegabknbiollkndbebei	Yes
	ToDoTab	jgealhbknfjhffedciigejkicpdnmhli	Yes
	Charming Tab	kbnpeiabjlfcakokkpbcgalbgiljoddf	Yes
	Handy Tab	kfnpaphhpnngikfmnofpkakbaekbafil	Yes
	TopTab	oilcbojeghcfkidelcmjbnbmaplfegbj	Yes
Table B.11	Black Tab	coadpnfaiboiicgpgeggcpkkgpbbcele	Yes
	Age Calculator	gbaakcccffklmhhjhfamehdfcieojmbb	No
	Film Links Now | Default Search	hfgpkllpjcfpakbldligbhmkgkajjndk	No
	Primary Tab	mkakgkpinfpfapnliafpjkeccjphjgjf	Yes
	Autumn Tab	omcgiabgadgmpcplhdlniiddjbcocaah	Yes
	Tasks Area | Task Management Tool	pahcgdhpimolppohfdgcnfjeglelonab	No