CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Spider-Scents v2: Enhancing Gray-Box Scanning for Stored XSS Vulnerability Discovery

Master's thesis in Computer science and engineering

YUNPENG YIN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Spider-Scents v2: Enhancing Gray-Box Scanning for Stored XSS Vulnerability Discovery

YUNPENG YIN

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

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

Spider-Scents v2: Enhancing Gray-Box Scanning for Stored XSS Vulnerability Discovery

YUNPENG YIN

Supervisor: Eric Olsson, Department of Computer Science and Engineering
Examiner: Magnus Almgren, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Spider-Scents v2: Enhancing Gray-Box Scanning for Stored XSS Vulnerability Discovery

YUNPENG YIN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Stored XSS vulnerabilities pose significant security risks in modern web applications, yet detecting them remains challenging due to their complex data propagation paths and the limitations of traditional scanning tools. Spider-Scents addresses these challenges using a gray-box database-aware approach that directly injects payloads into backend storage, effectively bypassing the difficulties of conventional input-based fuzzing. Building on this foundation, we present Spider-Scents v2 — a substantial enhancement of the original Spider-Scents prototype.

Spider-Scents v2 introduces two key advancements: a refined table traversal strategy that models the database schema as a directed graph and leverages BFS and DFS to systematically explore injection paths; and a format-aware payload customization module specifically tailored for JSON-structured data, which is increasingly common in modern applications. To evaluate the practical impact of these enhancements, we conduct a series of experiments assessing Spider-Scents v2's performance on both the original PHP-based applications and a broader set of non-PHP applications. Furthermore, we investigate how Spider-Scents database synthesis algorithm can serve as a preparatory module to augment the vulnerability detection capabilities of other black-box scanners, including Black Widow, Burp Suite, ZAP, and SCNR.

Our results demonstrate that Spider-Scents v2 offers measurable improvements in stored XSS detection, achieving better coverage and uncovering new vulnerabilities that were previously undetected. Additionally, the integration of Spider-Scents data synthesis algorithm enhances the effectiveness of third-party scanners, highlighting its potential as a complementary tool for web security assessments.

Keywords: Stored XSS, gray-box scanning, database synthesis, Spider-Scents v2, graph traversal, vulnerability detection, security testing, web application scanner

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Eric Olsson, for his invaluable guidance, unwavering support, and insightful feedback throughout the course of this thesis. His expertise and encouragement have been instrumental in shaping the direction of this research and helping me overcome the challenges along the way.

I would also like to extend my appreciation to my examiner, Magnus Almgren, for his valuable comments and suggestions that helped refine my work and improve its clarity and rigor.

Finally, I would like to thank all those who have supported and encouraged me during my studies. Their presence and help have been truly appreciated.

Yunpeng Yin, Gothenburg, 2025-07-02

Contents

List of Figures

List of Tables

1

Introduction

This chapter provides an overview of the thesis work, outlines its scope, summarizes the main contributions, and discusses ethical and sustainability considerations.

1.1 Overview

Web applications are integral to modern computing infrastructures, enabling services such as e-commerce, social networking, cloud-based platforms, and government portals. While their functionality and accessibility continue to improve, their growing complexity has also introduced numerous security risks. Cross-Site Scripting (XSS) remains one of the most common and widely exploited vulnerabilities in modern web applications [31], which allows an attacker to inject malicious scripts into web pages viewed by other users. These scripts can be used to steal credentials, hijack sessions, deface websites, or deliver malicious payloads.

XSS vulnerabilities are typically categorized into three types: reflected, DOM-based, and stored. Of these, stored XSS is especially dangerous because the malicious payload is permanently stored on the server, often in a database, and automatically served to users when the affected content is displayed. This persistent nature makes stored XSS particularly insidious, as it may affect multiple users over time and requires no further interaction from the attacker once the payload has been injected.

In 2023, ESET Research discovered and reported a stored XSS vulnerability (CVE-2023-5631) in Roundcube Webmail that was exploited by the Winter Vivern threat group to target European government agencies. This attack allowed malicious JavaScript embedded in specially crafted emails to steal sensitive information from users' mailboxes [13]. The case highlights the severe impact that stored XSS vulnerabilities can have when leveraged by Advanced Persistent Threat (APT) groups against critical government infrastructure.

Given the serious impact of XSS attacks, it is essential for developers to adopt preventive measures such as secure coding practices, robust input validation, output encoding, and deployment of secure frameworks. These measures serve as the first line of defense, reducing the risk of XSS vulnerabilities in production systems. However, despite these preventative efforts, XSS vulnerabilities can still find their way into web applications due to human error, legacy code, or complex data flows. Consequently, detection tools play a crucial role in identifying any remaining XSS issues.

Existing detection tools, both static and dynamic, face challenges when attempting to identify stored XSS vulnerabilities due to the diversity of web application implementations, such as differences in technology stacks, data flow models, and framework-specific behaviors.

Static analyzers may fail due to code obfuscation, language diversity, or missing context — such as the inability to track data flow across database boundaries without concrete runtime information [37]. Dynamic black-box scanners must analyze the relationships between control flow and data flow within a web application to identify appropriate injection points for XSS payloads and subsequently locate the output pages where these payloads are reflected [12]. These limitations are further exacerbated in database-driven web applications, where malicious inputs may be stored in non-trivial formats and retrieved asynchronously or through complex rendering logic.

This thesis builds on the original Spider-Scents prototype [25], a gray-box database-aware scanning approach that directly injects XSS payloads into databases to identify potential unprotected outputs in web applications. Spider-Scents demonstrated that by focusing on the database level rather than solely relying on frontend injection, it could reveal data flows that were missed by traditional input-based scanning methods. Building on this foundation, this thesis presents Spider-Scents v2, an expanded version of the original Spider-Scents. Spider-Scents v2 focuses on improving the detection of stored XSS vulnerabilities by optimizing how payloads are inserted into databases and integrating with other scanners. Key improvements include a new algorithm for prioritizing the traversal of database tables, a new payload generation mechanism for structured JSON data, an expanded evaluation set covering real-world applications beyond the PHP ecosystem, and evaluating the possible performance gain from integrating other scanning tools with the data synthesis algorithm of the original Spider-Scents.

1.2 Research Questions

This thesis aims to improve the capabilities and applicability of Spider-Scents for detecting stored XSS vulnerabilities in modern web applications. To guide our work, we define the following research questions:

- **RQ1: How can we enhance Spider-Scents to improve its effectiveness in discovering stored XSS vulnerabilities?**

  This thesis focuses on increasing the scanners detection capability through improvements to table traversal logic and format-aware payload generation, particularly for structured formats such as JSON.

- **RQ2: How can we expand the applicability of Spider-Scents to support diverse technology stacks and integrate with other security scanning tools?**

  This thesis evaluates the generalized performance of the tool beyond PHP-based web applications and its interoperability with other external black-box

scanners, enabling broader usage in realistic security testing workflows.

These questions represent the two core goals of this thesis: improving the internal detection logic and enabling flexible integration in varied environments.

1.3 Scope of this thesis

A typical attacker model in the context of web security assessments generally refers to an external adversary who does not have access to the application's source code or internal logic but can freely interact with the application as a normal or malicious user. This includes typical activities like browsing pages, submitting forms, and manipulating input fields to uncover security weaknesses.

This thesis focuses on the detection of stored XSS vulnerabilities in database-backed web applications. The scanner operates under a gray-box testing paradigm, where the tester has partial access to the underlying database schema and data, but no access to the application's source code. While this setup may not fully reflect a typical attacker model, it still provides a practical and defensible framework, allowing testers such as application developers or system administrators to identify vulnerabilities based solely on observed application behavior and data flows, without relying on understanding complex internal logic.

The scope of this work explicitly excludes the detection of reflected and DOM-based XSS vulnerabilities, which often require distinct techniques such as client-side taint tracking or dynamic JavaScript instrumentation. Additionally, the system does not rely on static code analysis, as its design philosophy centers on runtime interaction with web applications and their real or emulated database content to simulate more practical attack scenarios.

Although both Spider-Scents and Spider-Scents v2 are designed to be language-agnostic, this thesis expands their evaluation beyond PHP web applications, targeting also applications built in Java, Python, and Node.js. These languages are prevalent in modern development and known to be susceptible to XSS issues due to widespread use of dynamic content generation. By incorporating applications from these diverse language ecosystems, we aim to demonstrate the broader applicability of this approach.

1.4 Contribution

The thesis makes the following key contributions, addressing important challenges in detecting stored XSS vulnerabilities in web applications:

- **Traversal Optimization:** The original Spider-Scents used a simple table iteration order that might miss some injection points. To address this, Spider-Scents v2 introduces an optimized table traversal algorithm that prioritizes fields with a higher probability of affecting output generation. This improves injection coverage and increases the likelihood of finding stored XSS vulnera-

bilities.

- **Payload Customization for Structured Data:** Many modern web applications use structured data formats like JSON to store user inputs, but generic payloads may break parsing or not reach rendering points. Spider-Scents v2 therefore introduces structured-data-aware payload customization. By parsing JSON objects and injecting context-aware payloads that preserve structural validity, Spider-Scents v2 achieves a higher success rate in script execution without causing parsing errors than Spider-Scents v1.

- **Expanded Evaluation:** To validate these enhancements, the evaluation of Spider-Scents v2 goes beyond the PHP-based applications tested in the original work. The system is tested against a broader set of real-world applications written in different languages, including Java, Python, and Node.js, to assess its generalization and effectiveness. Furthermore, Spider-Scents's data synthesis algorithm is integrated with third-party scanners such as Burp Suite and ZAP to evaluate whether it can improve their detection performance. This demonstrates that Spider-Scents-generated application states provide valuable inputs for other scanning tools, broadening the scope of XSS vulnerability discovery.

1.5 Ethical Considerations

In this project, we focus on improving the detection of stored XSS vulnerabilities by using Spider-Scents v2. While our research aims to empower security practitioners and advance the field of vulnerability scanning, it is essential to acknowledge that these same tools can be misused by malicious actors.

To mitigate this risk, we adhere strictly to ethical guidelines for vulnerability research, inspired by established practices such as the Menlo Report [9]. All experiments in this work are conducted in a controlled, isolated testing environment using open-source applications deployed on local testbeds. No proprietary or production systems are involved, and no actual user data is collected or accessed during any phase of testing. This ensures that no real-world services, sensitive data, or critical infrastructure are put at risk.

Two new vulnerabilities we found during testing have been reported according to responsible disclosure practices, following the Google Project Zero policy [15], which is the standard policy used in industry today. This includes documenting the issue and informing the maintainers of affected systems to ensure that vulnerabilities are patched before any public disclosure. There are also no attempts are made to exploit or abuse discovered vulnerabilities beyond the scope of evaluation, and all research

activities strictly avoid any behavior that could be considered illegal, unethical, or in violation of cybersecurity laws and norms.

1.6 Sustainability

This research emphasizes the sustainability of the developed Spider-Scents v2 system from multiple dimensions. Technically, the system is built on well-established open-source technologies, which enhances its longevity, transparency, and fosters collaborative development. The approach adopted by Spider-Scents v2 ensures that the system remains maintainable and adaptable, allowing the broader community to continuously improve and extend its capabilities.

Operational sustainability is achieved through the use of lightweight, containerized environments (e.g., Docker) for conducting experiments and evaluations. By leveraging containerization, these experiments are both reproducible and portable across different computing infrastructures. This minimizes resource waste, reduces setup overhead, and simplifies the sharing of research outcomes. Such an approach ensures that future researchers can replicate, refine, or expand upon these experiments without significant energy or material resource investments.

Lastly, the environmental impact of the research is considered. Experiments are conducted in local, containerized environments to minimize idle resource consumption and to maximize resource efficiency. This low-impact approach not only reduces carbon footprints in scientific computing but also contributes to environmentally responsible research practices. By aligning with broader sustainability goals, this approach underscores the importance of minimizing the ecological impact of computational research while still delivering high-quality, impactful results.

1.7 Outline

This thesis is structured as follows.

Chapter 1: Introduction This chapter begins with an overview of the thesis and introduces the main research questions that guide the study. It then defines the scope of the work and outlines the key contributions. Ethical and sustainability considerations relevant to the research are briefly discussed. Finally, the chapter presents the overall structure of the thesis.

Chapter 2: Background. This chapter introduces XSS vulnerabilities with a focus on stored XSS, and presents XSS detection techniques including Spider-Scents approach in detail. It covers all the background knowledge necessary to understand the thesis work.

Chapter 3: Methodology. This chapter describes in detail the methodology, experimental setups and evaluation metric adopted in the development and assessment of Spider-Scents v2.

Chapter 4: Results. This chapter reports the experimental results and discusses their implications through detailed analysis.

Chapter 5: Conclusion. This chapter summarizes the thesis work and outlines some ideas for future research.

2

Background

In this chapter, we provide the foundational background and survey the most relevant literature for this thesis. We begin with an overview of stored XSS vulnerabilities, then explore various detection techniques, including white-box, black-box, and gray-box approaches, highlighting their respective strengths and limitations. Following this, we introduce Spider-Scents, a prototype detection tool designed specifically for stored XSS scanning, and discuss its core mechanisms. We introduced several popular black-box scanners that will later be used in our experiments for comparative evaluation. Finally, we reviewed relevant academic work on XSS detection.

2.1 Stored XSS

Cross-Site Scripting (XSS) refers to a vulnerability that allows attackers to inject malicious scripts into web content, which are then triggered within the browsers of end users [32]. The XSS vulnerability remains one of the most prevalent threats to web application security. According to the 2024 Cobalt Penetration report, XSS vulnerabilities account for approximately 9.4% of all reported web vulnerabilities, trailing only behind server security misconfigurations (28.4%) and missing access controls (19.2%) [5].

XSS vulnerabilities can be classified into three primary categories: reflected XSS, stored XSS, and DOM-based XSS. Reflected XSS occurs when malicious input is immediately reflected back to the user in a server response, requiring the attacker to trick the user into clicking a malicious link or submitting a specially crafted request. Stored XSS, in contrast, involves injecting payloads into persistent storage. DOM-based XSS differs from both in that it relies on the client-side JavaScript environment, exploiting dynamic DOM manipulations to execute malicious scripts entirely within the user's browser.

Figure 2.1 shows the distribution of different types of XSS in the real world. In the chart, Persistent refers to stored XSS, while Non-Persistent corresponds to reflected XSS. Mutation-Based represents a distinct variant, where the attack relies on browser-side parsing quirks to transform initially harmless input into executable code. Mutation-based XSS can occur in either reflected or stored form, depending on how the payload is introduced and later rendered.

Although stored XSS accounts for only about 2% of all XSS vulnerabilities, it is

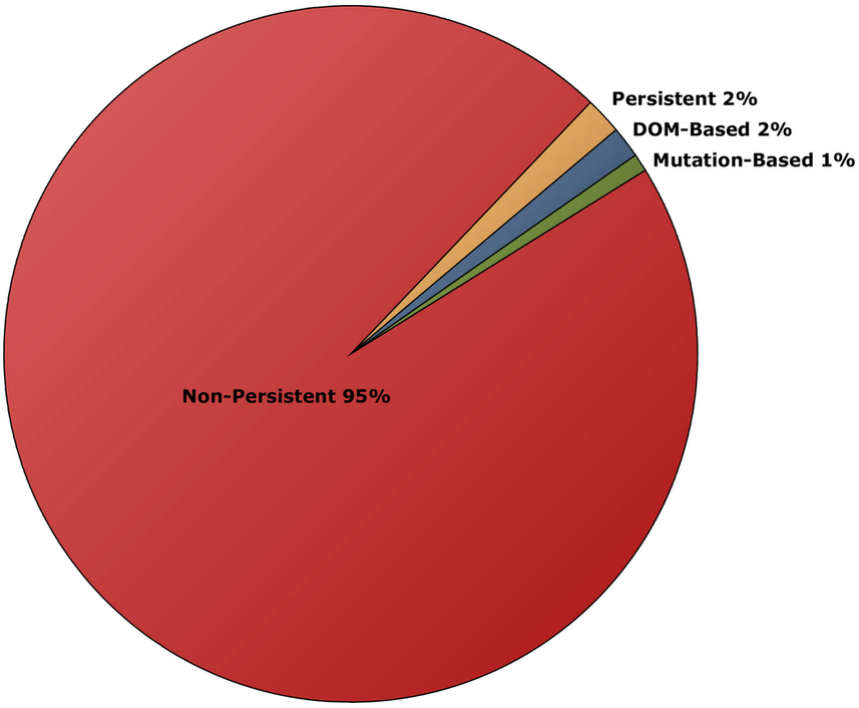

Figure 2.1: Distribution of Different Types of XSS Vulnerabilities in Real-World Applications (Source: [16] under fair use for academic purposes.)

considered the most dangerous form of XSS. In the stored XSS attack, malicious scripts are permanently stored on the target, commonly in a database, message forum, comment field, or other data persistence layers. When the victim loads a page containing the injected data, the script executes in the context of the victim's browser, effectively hijacking the client-side execution flow. The attack flow of stored XSS is shown in Figure 2.2.

Unlike reflected or DOM-based XSS, stored XSS does not require the attacker to trick the victim into clicking a crafted link. Instead, the malicious payload is automatically delivered whenever a user accesses the vulnerable resource. This makes it particularly effective for targeting multiple users, stealing session tokens, manipulating DOM elements, or even carrying out broader client-side attacks such as drive-by downloads and phishing.

A typical attack scenario involves an attacker injecting a payload such as

```
<script>document.location='https://attacker.com/steal?
cookie='+document.cookie</script>
```

into a comment field or profile description. Once a victim visits a page where this input is rendered without proper sanitization or encoding, the browser executes the script. The malicious code can then exfiltrate sensitive information like session cookies to a remote server controlled by the attacker. In more sophisticated cases, this stolen information can enable the attacker to perform privilege escalation. For example, by reusing a session cookie tied to an administrator account to gain higher-level access within the application, the attacker is not only able to impersonate the

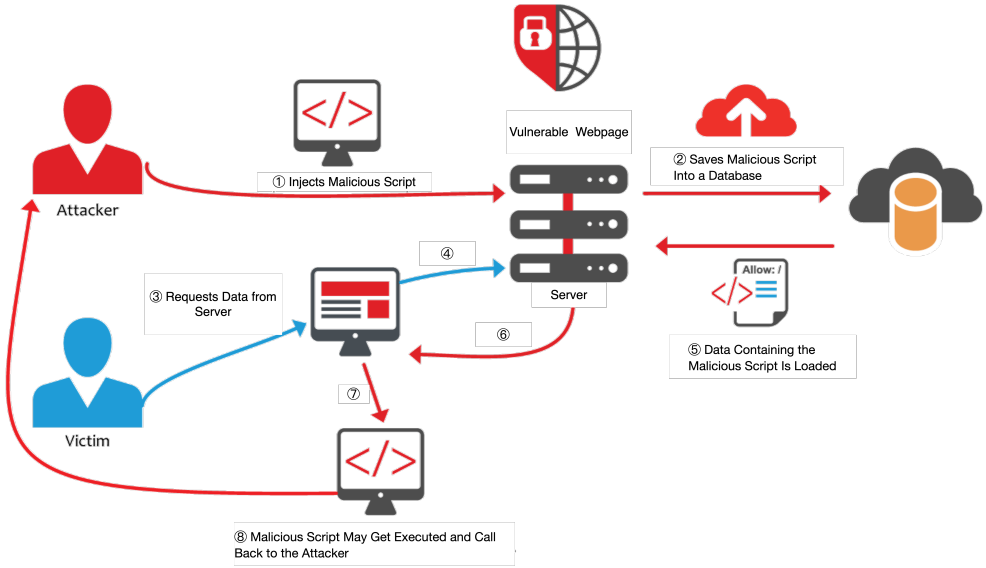

Figure 2.2: The Flow of a Stored XSS Attack.

user but also can perform privileged actions on their behalf, potentially leading to a complete compromise of the web application or underlying systems.

2.2 Mitigation

Mitigating XSS vulnerabilities is a multifaceted challenge that requires both proactive and reactive approaches. At the core of XSS prevention lies input validation and output encoding, two fundamental principles outlined by OWASP's best practices [26]. Input validation ensures that user-provided data adheres to the expected format and rejects any potentially dangerous input. Output encoding, on the other hand, ensures that dynamic content is safely integrated into web pages by encoding special characters in accordance with the context in which the data will be rendered (e.g., HTML, JavaScript and CSS). Together, these practices provide a robust first line of defense against injection attacks.

Beyond these foundational measures, modern web security also leverages the Content Security Policy (CSP) standard [22]. CSP enables web applications to specify allowed sources of executable scripts and other content, significantly reducing the risk of XSS exploitation by blocking unauthorized script execution. Implementing CSP properly can thwart many XSS payloads that rely on inline scripts or untrusted third-party resources.

Other technical countermeasures include setting HTTP security headers such as `X-Content-Type-Options: nosniff`, as well as adopting secure frameworks that

handle encoding and escaping automatically. Additionally, secure software development practices — such as code reviews, security-focused testing, and continuous integration of security scans — further bolster an application's resilience to XSS attacks.

Despite these mitigation strategies, XSS vulnerabilities continue to appear in real-world applications. This persistence underscores the limitations of prevention-only approaches, especially in complex, dynamic, and rapidly evolving web environments. Legacy code, third-party dependencies, and the interplay between client and server side logic all contribute to the difficulty of achieving perfect input sanitization and output encoding. Incorrect usage of development frameworks and misconfigurations of security setting by developers can also lead to the emergence of XSS vulnerabilities. Therefore, dedicated XSS detection remains an essential complement to these mitigations. Automated scanners and security-focused tools provide a complementary layer of defense by proactively identifying exploitable XSS weaknesses that may have bypassed preventive measures.

2.3 Detection Techniques

Detecting vulnerabilities in web applications is a critical aspect of ensuring software security. Over the years, researchers and practitioners have developed a wide range of methods to identify vulnerabilities at different stages of the software development and deployment lifecycle. These techniques can be broadly categorized in two dimensions: either based on their time of analysis, or based on their knowledge of the application's internal workings.

2.3.1 Based on Time of Analysis: Static Analysis and Dynamic Analysis

Static analysis inspects code, configuration files, or infrastructure definitions without executing the application. With this technique, one can identify potentially dangerous code patterns, data flows, or misconfigurations through rule-based or semantic analysis. Static analysis is particularly valuable during development and CI/CD pipelines, allowing early vulnerability detection. However, it may fail to capture runtime behaviors, especially those involving client-side code or dynamically generated content [28].

Dynamic analysis involves executing the application in a monitored environment (e.g., sandbox, instrumented browser, or testbed) and analyzing its behavior in response to various inputs. This includes observing network traffic, DOM mutations, cookie access, and script execution. While dynamic analysis is essential for validating runtime behaviors, it may fall short in scenarios requiring specific runtime conditions, such as stored XSS that only triggers when a privileged administrator accesses a particular page.

In such cases, static analysis may complement dynamic methods by reasoning about

abstract data flows across user contexts and application logic, without requiring the actual runtime interactions. However, static analysis alone cannot validate whether vulnerabilities manifest under real-world execution.

Within the scope of dynamic analysis, fuzzing techniques have emerged as a powerful approach to automatically explore program behavior [19]. By generating a wide range of inputs and monitoring application responses, fuzzing can exercise diverse execution paths and uncover vulnerabilities that may require complex workflows or privilege conditions to trigger. Although fuzzing operates dynamically, it benefits from feedback mechanisms such as code coverage to guide exploration more efficiently.

2.3.2 Based on Knowledge of the Internal Workings: White-box, Black-box, and Gray-box Scanning

White-box scanning assumes full access to the application's source code, configuration files, and infrastructure. Techniques in this category include static code analysis, Abstract Syntax Tree (AST) traversal, taint flow tracking, and symbolic execution. These methods are highly effective for identifying complex logic flaws and unreachable vulnerabilities early in the development cycle. However, they often suffer from high false positives and scalability issues in large, real-world applications.

Black-box scanning, in contrast, treats the application as an opaque system. The tester only interacts with the application through its inputs and observes its outputs, simulating an attacker's perspective. Tools based on black-box testing — such as Burp Suite, OWASP ZAP, and traditional fuzzers — send crafted payloads to inputs and inspect responses for signs of vulnerability (e.g., script execution, error messages). While black-box testing is language-agnostic and does not require source access, it may miss deeper flaws that only manifest under specific conditions or across multiple execution steps [2].

Gray-box scanning is a hybrid approach that leverages partial knowledge of the system's internals, such as database schemas, API documentation, or session logic, to enhance the efficiency and coverage of testing. Gray-box techniques often combine dynamic interaction with targeted instrumentation or metadata analysis, achieving a balance between depth and practicality [14]. For stored XSS in particular, gray-box testing can ease the difficult task of tracing payloads from storage points (e.g., user input forms) to sink points (e.g., HTML rendering), which is essential for multi-stage exploit paths.

2.4 Challenges in Detecting XSS

Detecting XSS vulnerabilities in modern web applications is challenging due to the complexity of input validation, data processing, and diverse rendering contexts. This section first reviews the general challenges that affect all types of XSS detection, and then focuses on the unique challenges posed by stored XSS in particular.

2.4.1 General Challenges in XSS Detection

Validation in modern web applications is pretty difficult. Attackers often employ sophisticated encoding and obfuscation techniques to evade detection, including URL encoding, Base64, or JavaScript functions like *String.fromCharCode()*. Such techniques dynamically construct scripts, making them harder to detect through static analysis or simple pattern matching [16].

Another significant challenge is handling structured and nested data formats — like JSON, XML, or HTML-rich fields — where payloads can be deeply embedded. Detection tools must be capable of parsing and decoding such formats to identify hidden payloads effectively.

Additionally, some payloads exploit browser parsing quirks or lesser-known HTML attributes (e.g., `onerror` in `<img>` tags). These tactics highlight the challenges of implementing robust input validation and output encoding in web applications, especially in frameworks that render rich or dynamic content.

A further complication is that the effectiveness of detection depends on context awareness. The execution of injected payloads varies depending on whether the payload appears in HTML content, attribute values, JavaScript contexts, or URLs. Consequently, security scanners must be context-sensitive to accurately model potential execution scenarios and identify vulnerabilities. That means the scanner should analyze not only the content of the payload but also its execution context, which may not be directly visible or straightforward to simulate.

2.4.2 Unique Challenges in Detecting Stored XSS

Stored XSS poses even more difficult challenges because of its two-phase nature: payload injection and later execution in a different user context. Specifically, stored XSS vulnerabilities arise when malicious inputs are persistently stored in a backend system (e.g., databases) and later retrieved and rendered as active content by unsuspecting users. This decoupling of injection and execution phases makes detection more complex and harder to fully simulate.

More generally, web application scanners targeting stored XSS face three key challenges [12]. First, they must model the application's state space and transitions. Second, they need to traverse complex workflows to discover how states are interconnected. Finally, they must track dependencies between state transitions, which may affect how and when payloads are triggered. These challenges significantly hinder the scanner's ability to achieve complete coverage and accurate detection.

Stored XSS detection also depends heavily on the coverage and capability of web crawlers. Many scanners rely on automated crawling and form submission to discover injection points and trigger payloads. However, modern web applications often include dynamic routing, authentication walls, and hidden parameters that limit crawler visibility, leading to missed attack surfaces.

Lastly, building high-fidelity, context-sensitive, language-agnostic simulation environments is technically demanding. Applications may use different server-side lan-

guages, client-side frameworks, or templating engines that affect how user inputs are processed and displayed. Achieving accurate payload injection, propagation, and execution modeling across such different platforms introduces significant technical and computational overhead. Moreover, heuristics or rules designed for one application often fail to generalize across others due to specific differences in application logic, input handling, or rendering behavior.

In summary, the detection of stored XSS requires handling multiple intertwined challenges: from injection path tracing and structured data parsing, to execution context modeling and output coverage. These complexities make stored XSS one of the most elusive and under-reported categories of XSS vulnerabilities, demanding advanced and adaptable detection strategies [36].

2.5 Spider-Scents

Spider-Scents [25] is a novel gray-box vulnerability scanner specifically designed to detect stored XSS vulnerabilities in database-backed web applications. Unlike traditional black-box scanners that attempt to inject payloads via user interfaces, Spider-Scents directly interacts with the database. This unique design enables it to bypass common challenges faced by other tools, such as input validation, multi-step forms, and encoding behaviors that often prevent effective payload injection.

Spider-Scents distinguishes between *unprotected outputs* and actual *XSS vulnerabilities*. Unprotected outputs refer to any database-stored content that is reflected on a web page without proper sanitization or encoding, representing a potential sink. To establish whether these unprotected outputs are truly vulnerabilities, Spider-Scents requires manual validation. This involves confirming that application inputs can be used to inject a payload that reaches the vulnerable database field. If such an input exists, the unprotected output is a complete stored XSS vulnerability.

2.5.1 Workflow

The workflow of Spider-Scents, presented in Figure 2.3, is designed to efficiently detect stored XSS vulnerabilities by combining direct database manipulation with automated web crawling and analysis. It follows a systematic pipeline that includes database preparation, payload injection, application integrity checks, and output analysis. This section outlines each phase of the workflow in detail:

① **Application Preparation with Data Synthesis**

The first step in the Spider-Scents workflow is to prepare the target web application for scanning. Unlike traditional scanners that operate directly on live data, Spider-Scents first ensures a robust starting point by synthesizing data into the application's database before any testing begins. This preparation is crucial because many real-world or demo applications lack sufficient or realistic datasets that reflect typical application states.

To address this, Spider-Scents programmatically analyzes the database schema,

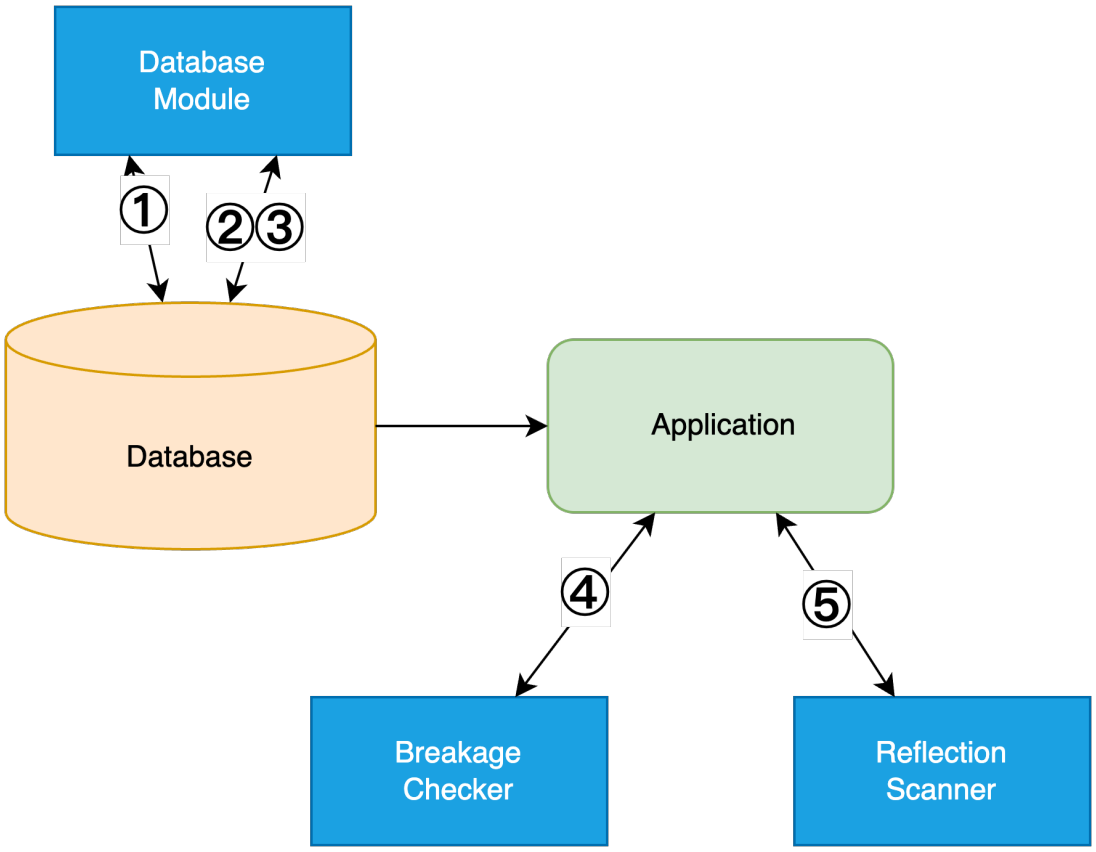

Figure 2.3: The Workflow of Spider-Scents. Adapted from [25] with permission.

identifies empty tables, and injects synthetic schema-compliant benign data into them. This synthetic data generation follows a structured approach (as detailed in Algorithm 1 of the paper [25]), ensuring that the inserted rows maintain integrity constraints and support the discovery of vulnerabilities that span across multiple tables.

Rather than relying on static or manually curated payloads, Spider-Scents performs context-aware data synthesis, ensuring that the inserted test data maintains application-level validity. This approach enables Spider-Scents to simulate realistic user inputs and better trigger XSS vulnerabilities embedded within structured data storage.

② Choosing a Database Cell for Modification

After preparing the database with synthesized data, Spider-Scents proceeds to select specific database cells for payload injection. This step is critical, as inappropriate modifications may either fail to reveal vulnerabilities or cause unintended application breakage.

For each cell, Spider-Scents first ensures that the column is of a textual type and long enough to hold the full XSS payload. Only fields capable of preserving the payload structure — like `VARCHAR`, `TEXT`, or similar — are considered valid

injection points.

To prevent application breakage or data inconsistency, Spider-Scents actively avoids injecting into sensitive rows. A sensitive row refers to one where deletion, modification, or reinsertion causes the application to misbehave — such as reverting changes or triggering database conflicts. This sensitivity is determined using an algorithm to detect such sensitive rows, which performs test deletions, insertions, and crawls to identify any rollback mechanisms or application defenses triggered by modifying a particular row.

Once non-sensitive rows are identified, Spider-Scents uses an iterative strategy to modify each row within the selected tables. This comprehensive strategy ensures that non-uniform row behaviors — such as conditional reflection logic based on key-value structures — are taken into account. In such scenarios, each entry must be individually analyzed since the application may treat them differently.

Moreover, the selection strategy acknowledges that the order of modification can impact detection success. Thus, Spider-Scents allows configurable traversal strategies (e.g., alphabetical or random table order), which can significantly affect the scanning coverage and results in a limited time-budget. However, these traversal strategies may still miss certain cases in large and complex schemas because they rely on static orders rather than real data dependencies or usage patterns. This issue is a key focus of this thesis and will be discussed further in Section 3.1.1.

③ Payload Insertion

After selecting a suitable database cell that is not marked as sensitive and can accommodate the payload, Spider-Scents directly injects a specially crafted XSS payload into the database, bypassing the front-end user interface entirely. This approach stands in contrast to traditional black-box scanners that rely on interacting with the web application's forms or APIs to deliver input.

Each payload is assigned a unique identifier (ID), which is embedded within a JavaScript snippet to aid in later detection. The default payload structure follows a pattern like:

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

This helps avoid false positives by ensuring that any output can be unambiguously tied back to the scanner's own injected ID. Importantly, the scanner only considers an injection successful if the xss(ID) function is actually executed during page rendering, rather than merely present in the JavaScript code or DOM. It also simplifies detection in cases where multiple payloads are evaluated simultaneously.

Importantly, the tool accounts for structured data formats within database fields. When it detects serialized formats — such as PHP serialized objects or image paths — Spider-Scents attempts to inject payloads into valid positions

within these formats, rather than naively inserting a raw string. This enhancement improves detection accuracy in applications where data is not stored in plain text fields.

However, the original version of Spider-Scents does not attempt to handle all structured formats. For example, JSON or other more uncommon formats were out of scope for this initial version. In this thesis, we specifically address the challenge of handling JSON-formatted data.

④ **Application Breakage Detection**

Once the payload is injected into the database, Spider-Scents performs a critical step: it checks whether the application is still functioning normally. This step is essential because an improperly injected payload could unintentionally break the application's front-end, backend logic, or even its routing behavior.

For example, injecting an XSS payload into a field that stores the website's domain name might corrupt how URLs are generated across the entire site. In some Content Management Systems (CMSs), this single change can make the admin interface unreachable. Traditional scanners would likely fail silently in such cases, unable to recognize that their actions have compromised the test environment. Note that such breakages are less likely to occur with traditional scanners, as they typically do not directly manipulate the application's internal state via the database. Spider-Scents, in contrast, interacts with stored data more directly, which increases both its detection capability and its potential to disrupt application behavior.

To detect such breakage, Spider-Scents compares the application's behavior before and after the injection. Specifically, it crawls a list of known URLs — gathered in earlier steps — and measures key properties of each page, such as HTTP status codes, HTML length, and the number of links. If the differences exceed a certain threshold, the system concludes that the injected payload has caused breakage.

But Spider-Scents doesn't stop at detection. It includes a repair mechanism to automatically revert the injected change if the payload is found to be responsible for breaking the application. This ensures that the testing process is safe and recoverable, even when aggressive payloads are used.

⑤ **Reflection Scanning** After payloads have been inserted into selected database cells, Spider-Scents proceeds to analyze how the modified content is reflected on the web interface. To perform this reflection analysis, Spider-Scents integrates with Black Widow, a black-box reflection scanner. Black Widow is responsible for crawling the application and recording all instances where the unique IDs (embedded in the XSS payloads) appear in the DOM/HTTP responses, or are executed as code.

Spider-Scents applies minor modifications to Black Widow, enabling inter-module communication and making the scanner aware of which IDs are being tracked in a specific execution session

2.5.2 Evaluation Methodology

To evaluate the effectiveness of Spider-Scents in detecting stored XSS vulnerabilities, Olsson et al. conducted a set of experiments across twelve open-source, database-backed web applications. These included modern content management systems such as WordPress, Joomla, and PrestaShop, as well as reference systems with known vulnerabilities. The experiments were performed in a controlled environment using standard setups of applications to reflect realistic deployment scenarios.

The evaluation focuses on three primary dimensions: database coverage, vulnerability discovery, and exploitability. Database coverage measures the scanner's ability to inject payloads into reachable storage fields, verified by comparing the database state before and after scanning. Vulnerability discovery tracks the number of stored XSS issues identified, and exploitability assesses whether the reflected payloads could be used by an attacker to elevate their privileges.

2.5.3 Result

Across all targets, Spider-Scents demonstrated strong effectiveness in identifying stored XSS vulnerabilities through its end-to-end workflow.

In total, Spider-Scents discovered 133 unique stored XSS issues, 85 of which were verified to be vulnerable in realistic usage scenarios. These vulnerabilities were confirmed by observing unsanitized reflections of injected payloads in application output and validating the feasibility of triggering execution through normal user interactions.

The tool successfully inserted payloads into the vast majority of writable database fields, including those embedded within structured or serialized data formats. It also proved resilient in applications with multi-step navigation or form workflows, maintaining stable execution and recovery even when injected payloads caused temporary breakage.

These results affirm the core design goals of Spider-Scents — namely, achieving high injection coverage, enabling accurate tracking of reflected payloads, and supporting practical exploit validation — without requiring invasive instrumentation or full access to application code. In addition, Spider-Scents is significantly faster than large-scale fuzzing campaigns or costly static analysis, which often require many hours or even days to achieve comparable coverage. In contrast, Spider-Scents can typically complete scans in a matter of minutes to a few hours, highlighting its practical efficiency for stored XSS detection.

2.6 Other XSS Scanners

Several widely-used vulnerability scanners provide built-in support for detecting XSS vulnerabilities. While these tools vary in architecture and sophistication, most rely on black-box or hybrid approaches, attempting to inject payloads through user

interface inputs and monitoring the resulting application responses. This section briefly introduces four commonly used scanners in the context of XSS detection.

2.6.1 Black Widow

Black Widow [12] is a black-box web application scanner that incorporates advanced navigation modeling and inter-state dependency tracking to enhance its vulnerability detection capabilities. Its core strength lies in a dynamic JavaScript crawler that can traverse both static and dynamic elements within modern web applications. This includes traditional structures such as links, forms, and frames, as well as event-driven behaviors like mouse clicks, hover actions, and window resizes — interactions that are often crucial for uncovering XSS vulnerabilities embedded within complex front-end logic.

One of Black Widow's key contributions to XSS detection is its ability to construct and utilize a navigation model that maps the paths users take through an application. By recording the sequence of user interactions necessary to reach specific application states, the tool can reliably reproduce these sequences to probe for XSS conditions that only manifest in particular user contexts. Moreover, Black Widow employs dynamic taint tracking to detect potential injection points and their corresponding sinks. It achieves this by injecting unique taint markers into input fields and subsequently analyzing the application's responses for occurrences of those markers in output locations such as HTML content. This approach allows the scanner to identify whether user input is improperly sanitized and reflected or stored in a way that enables script execution, which is essential for discovering both reflected and stored XSS vulnerabilities.

However, its emphasis is primarily on client-side navigation and event replay. It does not deeply model the backend data lifecycle, which can make it harder to detect stored XSS vulnerabilities that depend on specific database interactions across sessions or user states.

2.6.2 Arachni

Arachni [20] is a robust and extensible web vulnerability scanner developed in Ruby, designed to support complex and dynamic web applications. It operates across major platforms — including Windows, macOS, and Linux — and offers multiple modes of deployment, including a command-line interface, a scripting API, and a multi-user web interface for collaborative assessments.

One of Arachni's key strengths lies in its integrated browser-based scanning engine, which allows it to effectively simulate user interactions and process modern web technologies such as JavaScript, HTML5, DOM manipulation, and AJAX. This enables the tool to navigate and assess highly dynamic content where XSS vulnerabilities may be deeply embedded in client-side execution paths. With respect to XSS detection, Arachni is equipped to identify both reflected and stored XSS issues. It performs input vector analysis by injecting payloads and monitoring for execution in output contexts, while also maintaining stateful awareness of the application to

handle session-dependent and multi-stage attack surfaces. Its adaptive scanning engine allows it to learn the behavior of individual web resources and tailor its approach accordingly, increasing its accuracy and reducing false positives in real-world deployments.

While Arachni's browser-based strategy enhances coverage for client-side XSS, it may face challenges with stateful, multi-step vulnerabilities, where an injected input needs to persist across different application stages — an area where Spider-Scents' database-aware model offers distinct advantages.

It is worth mentioning that Arachni has become obsolete now. After its development stopped in 2022, Codename SCNR [11] was introduced by a former Arachni contributor as its modern successor, offering improved performance, lower resource consumption, and broader compatibility with modern web applications.

2.6.3 OWASP ZAP

OWASP ZAP [27] is an open-source web application security scanner developed by the Open Web Application Security Project (OWASP). Designed for both beginners and professional security testers, ZAP provides an intuitive interface and a wide range of automated and manual tools for vulnerability discovery. Among its core capabilities is the XSS vulnerabilities.

For XSS detection, ZAP operates by injecting crafted payloads into discovered input fields during active scans and monitoring application responses for signs of successful injection. It also includes a powerful Fuzzer component that allows users to manually craft and customize attack payloads, extending its capabilities beyond its built-in scan rules. To improve coverage on modern web applications, ZAP integrates with CrawlJax-based [24] browser automation. This allows it to explore dynamic, JavaScript-driven content, supporting the discovery of input points that are hidden behind user interactions or DOM events.

However, while ZAP is highly flexible and extensible, its default scanning strategy often relies on black-box exploration and heuristic guessing. It may face challenges when applications require multi-stage workflows, complex authentication, or involve stored data dependencies across sessions.

2.6.4 Burp Suite

Burp Suite [30], developed by PortSwigger, is one of the most popular and comprehensive platforms for web security testing. It combines automated vulnerability scanning with powerful manual tools such as Proxy, Repeater, Intruder, and Extender, offering users complete control over the testing process.

Burp Suite's Active Scanner is capable of detecting a wide range of XSS vulnerabilities, including reflected, stored, and DOM-based types. It employs payload injection strategies during automated scans, and monitors application behavior to flag suspicious responses. Through the Extender framework, Burp can be extended with custom plugins, significantly increasing its adaptability to specific application

behaviors or attack surfaces. One of Burp's key strengths is its ability to combine automation with human-guided exploration. Experienced testers can use its manual tools to fine-tune attacks, replay requests, and deeply investigate application logic that automated tools might miss.

Nonetheless, when relying solely on its default automated scanning, Burp tends to operate primarily in a black-box mode, without insight into backend data handling. In applications where stored XSS vulnerabilities are tied to database-specific workflows or complex field dependencies, Burp's automated processes may not always fully exercise these paths unless specifically configured or manually guided.

2.7 Related Work

A variety of approaches have been proposed to detect XSS and stored XSS. This section reviews relevant prior work in the three main classes of method: static analysis, black-box scanning, and gray-box scanning. We also review related work on data synthesis, which plays a critical role in enabling effective testing of web applications.

2.7.1 Static Analysis

The work by Dahse and Holz [8] represents a foundational contribution in the field of detecting second-order vulnerabilities, including stored XSS. Their approach extends static analysis tools (notably RIPS) to perform delayed taint tracking, enabling detection of vulnerabilities where malicious input is stored and later reused in security-critical contexts.

Wang et al. [38] proposed a static analysis method to detect stored XSS vulnerabilities in PHP web applications by integrating program slicing techniques. Their approach focuses on identifying two critical stages of stored XSS: threat injection (user input without proper sanitization stored in files or databases) and threat release (unsanitized data retrieved and rendered). By generating program slices encompassing both phases, their method offers a fine-grained view for manual verification or dynamic analysis. Their tool extends Pixy's [18] static analysis capabilities to specifically target stored XSS, addressing limitations of prior taint tracking methods by introducing more precise array alias handling.

Another noteworthy tool in the field of stored XSS detection is Splendor [37], a static analysis framework designed specifically for modern web applications. Splendor focuses on analyzing PHP applications with Data Access Layers (DALs), employing a novel triple-matching strategy that identifies taint flows between database writes and subsequent reads. This static approach enables it to uncover stored XSS vulnerabilities by examining code-level relationships, without requiring the application to be executed. While static analysis methods like Splendor excel at identifying deeply buried flows and vulnerabilities in complex codebases, they often face challenges such as high false-positive rates and the inability to validate exploitability in live environments. Despite these limitations, Splendor provides a complementary

perspective to dynamic analysis approaches, making it a valuable addition to the stored XSS detection landscape.

2.7.2 Black-box Scanning

McAllister et al. [21] introduced an approach that leverages real user interactions to drive fuzzing-based scanning for stored XSS vulnerabilities. By replaying authentic usage patterns — such as navigation paths and form submissions — their method could uncover vulnerabilities that might be missed by traditional fuzzing tools. However, this reliance on actual interaction data can be a limitation: in real-world settings, such data may not always be available or may not comprehensively cover all possible application states.

KameleonFuzz [10] uses an evolutionary algorithm and a genetic attack grammar to generate, evolve, and prioritize fuzzed inputs, mimicking the behavior of a human attacker. By combining control-flow and precise taint flow, it aims to find XSS vulnerabilities missed by other black-box scanners. Moreover, the need to reset the application state for each run can be time-consuming.

Both jÄk [29] and CrawlJax [23] focus on addressing the challenges of modeling client-side state in modern web applications. Specifically, jÄk hooks JavaScript APIs to observe dynamic behaviors and generates a navigation graph that captures these runtime interactions, enabling the crawler to explore state transitions that static approaches miss. CrawlJax, meanwhile, similarly targets dynamic content by interacting with candidate elements — such as clickable or form-submitting components — to build a state-flow graph that reflects the evolving user interface. These methods highlight the importance of dynamic analysis for comprehensive vulnerability scanning, particularly in complex, single-page applications where stored XSS might be hidden behind user-triggered state changes.

Black Widow [12] is a data-driven web vulnerability scanner that addresses three key challenges: navigation modeling, workflow traversal, and inter-state dependency analysis. Unlike traditional scanners that often treat these challenges separately, Black Widow combines them into a unified approach to improve coverage and vulnerability detection. One of its notable strengths is that it does not assume the ability to reset the web application, making it more practical in real-world scenarios where full system resets are often infeasible.

2.7.3 Gray-box Scanning

Steinhauser and Tůma [36] introduced a gray-box approach to detect stored and context-sensitive XSS vulnerabilities by intercepting the communication between the web application and its database. Unlike traditional black-box scanners, their technique directly injects exploit payloads into database responses, systematically uncovering vulnerabilities stemming from improper handling of stored data or encoding mismatches across different browser contexts. While Spider-Scents also leverages database-level interactions to overcome input validation challenges, it differs by using an explicit database synthesis phase and controlled injection of payloads directly

into database cells, rather than intercepting database traffic during application runtime. Spider-Scents thus provides more explicit control over which data paths are exercised, enabling fine-grained evaluation of stored XSS scenarios.

GELATO [17] employs a feedback-driven, guided crawling approach for dynamically exploring modern JavaScript applications, with a focus on discovering DOM-based and reflected XSS vulnerabilities. It leverages taint tracking and targeted exploration to improve coverage and detection accuracy, especially for client-side issues. However, it does not address stored XSS vulnerabilities that originate in backend data persistence layers — an area specifically tackled by Spider-Scents, which also evaluates how stored payloads propagate from databases to output contexts.

WebFuzz [34] is also a gray-box fuzzing tool that instruments PHP applications to gain coverage feedback for more effective vulnerability discovery. By leveraging this feedback, it can detect stored XSS vulnerabilities in real-world applications, outperforming traditional black-box scanners. However, it struggles with Single Page Applications (SPAs) that rely heavily on JavaScript for dynamic content rendering.

In addition, Song et al. [35] introduced a gray-box fuzzing approach that integrates static analysis with reinforcement learning to efficiently detect reflected and stored XSS vulnerabilities in Java web applications. Their method first employs static analysis to identify potential input points across Java code, configuration files, and HTML. Then, they leverage reinforcement learning to generate optimized attack payloads for fuzzing. Experimental evaluations demonstrate that this hybrid approach significantly outperforms four state-of-the-art web scanners in both detection coverage and speed, with no false positives.

2.7.4 Data Synthesis

SynthDB [4] proposes a sophisticated database synthesis framework that leverages concolic execution[1] to generate realistic and integrity-compliant database states, aimed at improving the input-output mapping accuracy of black-box scanners. While SynthDB's reliance on application-level constraints provides high fidelity, it introduces significant overhead and requires access to application source code. In contrast, Spider-Scents adopts a lighter-weight database synthesis approach that populates empty tables with schema-compliant data directly, without code analysis. This design choice makes Spider-Scents easier to deploy across diverse applications and languages. Although SynthDB's rigorous data synthesis can enhance vulnerability scanning coverage, Spider-Scents achieves competitive or superior database coverage in practice with less complexity and faster deployment.

EvoSQL [3] focuses on search-based test data generation for database applications, particularly targeting SQL queries. It leverages evolutionary algorithms to generate data that satisfies complex database constraints and triggers specific query behaviors, making it valuable for testing and verifying database-centric applications. However, unlike Spider-Scents— which is designed to work directly with web applications and

[1] Concolic execution is a software testing technique that combines concrete and symbolic execution to systematically explore program paths.

detect stored XSS through database manipulation and realistic crawling — EvoSQL does not extend to the runtime aspects of web applications, limiting its effectiveness in detecting vulnerabilities like XSS. Its primary focus remains on database query correctness and coverage, rather than vulnerability scanning in dynamic web contexts.

Additionally, DOMINO [1] is a tool designed for efficient and diverse test data generation specifically targeting relational database schemas. By leveraging domain-specific operators and intelligently re-using values or introducing fresh values, DOMINO achieves high coverage and diversity, making it effective for uncovering database integrity constraint violations.

2.8 Positioning of Our Work

Spider-Scents provides a compelling foundation for further development due to its unique gray-box scanning approach that directly targets stored XSS vulnerabilities at the database level. It bypasses traditional input-based injection methods by inserting payloads into backend storage, thereby exposing vulnerabilities that may otherwise remain hidden. In addition to its novel methodology, the original Spider-Scents also demonstrated strong results in detecting stored XSS vulnerabilities in its evaluation, further supporting our decision to build upon it.

However, the prototype also presents several limitations, including simplistic table traversal, limited adaptability to structured payload formats such as JSON, and lack of integration with other scanning tools. Moreover, its evaluation was confined to PHP-based applications, leaving its applicability to other technology stacks unverified.

We summarize these aspects of the original Spider-Scents to motivate our decision to extend this particular system — rather than building a new tool from scratch or adopting an alternative model. In this thesis, we propose Spider-Scents v2, an enhanced version of Spider-Scents that addresses these limitations. The design and implementation of our system are described in detail in the following chapter.

3

Methodology

This chapter outlines the methodology employed to enhance and further evaluate the effectiveness of Spider-Scents. We call the original Spider-Scents as Spider-Scents v1. The proposed Spider-Scents v2 incorporates several improvements over its predecessor, aiming to increase vulnerability detection, and verify that this generalizes across different web applications. The methodology consists of two major components: system enhancement (Section 3.1) and experimental design (Section 3.2).

3.1 System Enhancements

The core contribution of this thesis lies in the development of Spider-Scents v2, which incorporates multiple technical improvements over the original.

3.1.1 Table Traversal Strategy

The table traversal strategy of Spider-Scents v1 was relatively simple, relying mainly on either an alphabetical or a random order to traverse and modify database tables. However, this approach did not consider the potential dependencies or relationships between tables, such as foreign key constraints or logical data flows, which can impact how payloads propagate through the application and where vulnerabilities manifest.

In Spider-Scents v2, the database schema is abstracted as a directed graph to facilitate the traversal of related tables in a principled and scalable manner. Specifically, each database table is modeled as a graph node, and each foreign key constraint is represented as a directed edge from the referencing table to the referenced table. This abstraction captures the relational structure of the database and allows for flexible navigation through the schema.

Once the schema graph is constructed, graph traversal algorithms are employed to explore the structure and determine optimal paths for propagating input data. Breadth-First Search (BFS) and Depth-First Search (DFS) are chosen because they provide fundamental and widely-used strategies to exhaustively explore different graph structures. Compared to more complex algorithms like A* or Dijkstra's algorithm (often used for weighted graphs), BFS and DFS are simpler and well-suited for unweighted, schema-driven graph traversal typical of database schema representations [6]. These characteristics make them robust default choices for structural

exploration without introducing unnecessary complexity.

Breadth-First Search (BFS). In BFS, traversal begins at a selected root node and proceeds level by level, visiting all directly connected nodes before moving deeper into the graph. This method ensures that all reachable tables at the same distance from the root are visited early. BFS is particularly suitable for scenarios where the goal is to identify all tables within a certain number of relational hops from a starting point—for example, when locating all tables that might be affected by a form input associated with a specific entity. Additionally, BFS tends to yield shorter and more interpretable traversal paths, which are beneficial when tracing data propagation paths during vulnerability analysis.

Depth-First Search (DFS). In contrast, DFS explores as far as possible along one branch before backtracking. This strategy is more effective when the focus is on uncovering deeply nested foreign key chains or exploring complete dependency paths from a starting table. DFS can help identify vulnerabilities that are triggered only through long relational paths—for example, when a user input is inserted into one table but later used in a distant table joined through multiple foreign keys. The use of DFS enables Spider-Scents v2 to simulate such multi-hop propagation flows and capture vulnerabilities that would be missed by shallower strategies.

By leveraging both BFS and DFS in different contexts, Spider-Scents v2 achieves a comprehensive exploration of the relational structure of databases. This flexible traversal framework allows for more effective generation of payloads targeting stored XSS vulnerabilities across interconnected tables, improving both detection accuracy and coverage.

To illustrate the table traversal strategy more concretely, consider a simplified e-commerce database schema involving four tables: `Users`, `Orders`, `Order_Items`, and `Products`. In this schema, each `User` can place multiple `Order`s, establishing a one-to-many relationship via the foreign key `user_id`. Each `Order` can contain multiple `Order_Item`s, and each `Order_Item` is associated with both an `Order` and a specific `Product` through the foreign keys `Order_Item.order_id` and `Order_Item.product_id`, respectively. The table structure is shown in Figure 3.1.

When modeling this schema as a graph, each table becomes a node, and each foreign key becomes a directed edge pointing from the referencing table to the referenced table. This yields a Directed Acyclic Graph (DAG) structure that captures the dependencies across the schema, as shown in Figure 3.2. Using graph traversal algorithms such as BFS or DFS, Spider-Scents v2 systematically explores the relational structure of a database schema modeled as a directed graph. As a result, traversal begins at nodes with zero in-degree — those that are not referenced by any other tables.

In the provided schema, `Order_Item` is the only node with zero in-degree, making it the root of the traversal graph. A BFS traversal starting from `Order_Item` would visit its immediate dependencies `Order` and `Product`, and then continue to `User`. This level-by-level traversal is effective for capturing surface-level data propagation patterns and understanding how input flows outward through relational links.

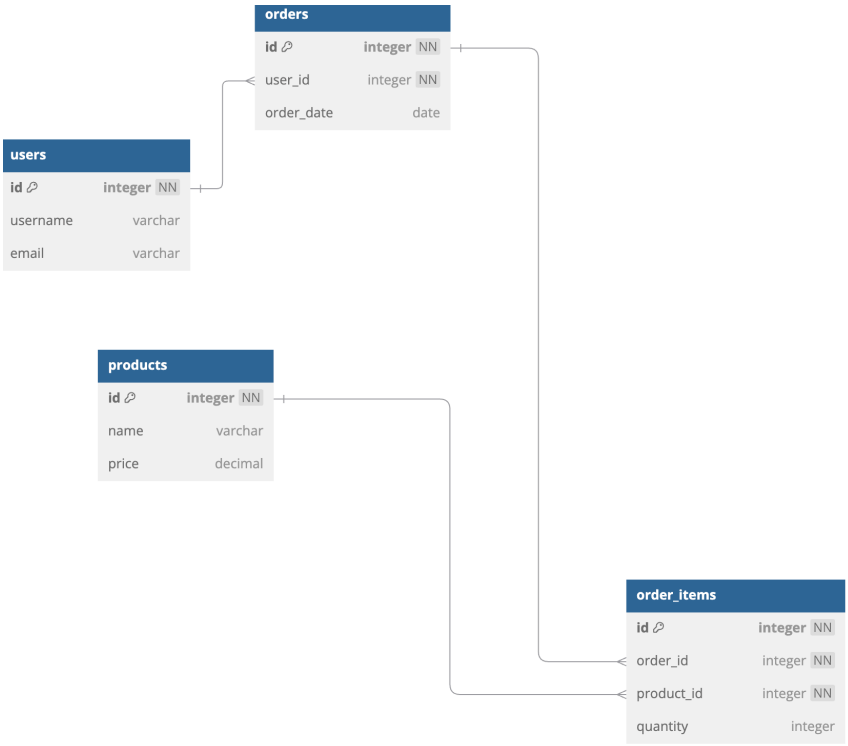

Figure 3.1: Tables of Example Database Schema. This schema consists of four tables: `users`, `orders`, `products`, and `order_items`. The `orders` table references `users` through the `user_id` foreign key, while `order_items` links `orders` and `products` via `order_id` and `product_id` foreign keys, respectively. These relations illustrate a typical e-commerce order management structure.

Conversely, DFS proceeds by following each path as deeply as possible before backtracking, allowing Spider-Scents v2 to uncover complex multi-hop data flows. Starting from `Order_Item`, a DFS traversal can reach `User` by passing through `Order`, or reach `Product` directly, capturing scenarios where deeply nested references may introduce latent vulnerabilities. Such deep path exploration is essential for detecting stored XSS issues that arise from indirect dependencies across multiple tables.

3.1.2 Format-Aware Payload Customization for JSON Data

JavaScript Object Notation (JSON) [7] is a lightweight, text-based data interchange format that has become the de facto standard for structured communication between client and server in modern web applications. Due to its simplicity, human readability, and native compatibility with JavaScript, JSON is widely adopted for transmitting configuration data, user inputs, API requests, and responses across a broad range of platforms.

JSON represents data as key-value pairs enclosed within curly braces {}, with support for nested objects and arrays. Keys must be strings, while values can be strings, numbers, booleans, arrays, objects, or null. For example, a typical JSON payload submitted via a web form might look as follows:

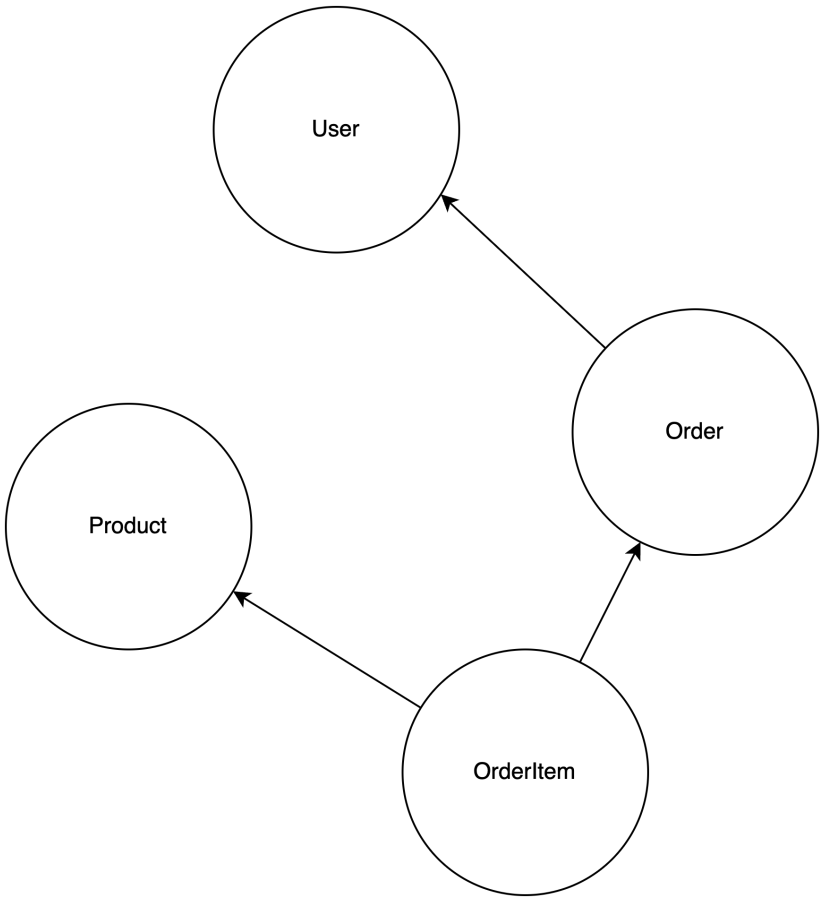

Figure 3.2: The Graph Generating from Tables Visually Represents the Relationships in the Schema. Each node corresponds to a table and each directed edge indicates a foreign key constraint linking two tables.

```
{
  "name": "Alice",
  "email": "alice@example.com",
  "profile": {
    "bio": "I love coding",
    "age": 25
  },
  "tags": ["developer", "security", "xss"]
}
```

In this example, the top-level object contains both primitive fields (`name, email`), a nested object (`profile`), and an array of strings (`tags`). Such structured representations allow applications to encapsulate complex user input or system configurations within a single, parseable unit. However, this also presents challenges for security testing tools, as any payload injection must preserve the overall syntax and data types of the JSON structure to avoid parsing errors on the server side.

3.1.2.1 Payload Customization

The original version of Spider-Scents incorporated format-specific payload customization mechanisms, particularly for image path fields and PHP serialized data. These strategies allowed the scanner to inject payloads that preserved the syntactic integrity of the target format, such as appending script-laden query parameters to image paths or injecting serialized payloads into PHP objects without breaking deserialization.

Building on this foundation, Spider-Scents v2 introduces a structure-aware injection strategy, extending its payload customization to effectively support complex JSON content types. The scanner first parses the JSON payload into an internal representation, such as an abstract syntax tree, and identifies viable injection points — string values, numeric fields, array elements, or object properties. It then generates context-sensitive payloads that maintain JSON validity, correctly escape characters, and respect the surrounding structure.

This enhancement enables Spider-Scents v2 to explore deeper input hierarchies and reach vulnerable rendering points that may be triggered after deserialization, templating, or client-side usage. By extending its customization logic from legacy formats to structured JSON, Spider-Scents v2 may improves its applicability and effectiveness in modern web environments.

3.1.3 Integration with other scanners

To support integration with other security scanners, Spider-Scents v2 reuses the first phase of its internal workflow, the program preparation stage, which is responsible for exploring and generating application state representations. This phase leverages the tool's core strength: a schema-guided data synthesis algorithm that traverses input vectors, generates valid payload-ready requests, and drives the application into diverse and meaningful runtime states.

Instead of continuing into its own scanning phase, Spider-Scents v2 exposes the output of this preparation stage — the reconstructed application states — as an input to external scanners. These states include a wide range of pre-filled forms, pre-populated parameters, and authenticated session paths.

By simulating realistic user interactions and database-backed workflows, Spider-Scents ensures that the web application is placed into a scanning-ready state that reflects true usage scenarios. The workflow is displayed in Figure 3.3.

This strategy allows general-purpose vulnerability scanners to operate more effectively. Rather than initiating scanning from default or shallow entry points, these tools can now begin from semantically rich, deeper program states synthesized by Spider-Scents v2. This integration improves both the depth and relevance of the scanning process, enabling the detection of vulnerabilities that might only be reachable through multi-step navigation or specific input combinations.

3.1.4 Implementation Notes

For the alternative BFS and DFS table traversal strategies, we extended the database access module of the original Spider-Scents. Specifically, we first implemented logic to extract foreign key relationships from the database schema, enabling the construction of a directed graph where each node represents a table and each edge denotes a foreign key dependency. We then added a graph construction component that abstracts the relational schema into this graph structure. Finally, we implemented both BFS and DFS algorithms to compute different table traversal orders.

In addition, we extended the payload injection module to support JSON data formats. We implemented logic to detect whether the data in a database cell is formatted as JSON, and if so, we parsed the structure and identified appropriate injection points within the value fields. The payloads were then inserted while preserving the syntactic correctness of the JSON format, ensuring proper escaping of special characters to avoid breaking parsing on the server side.

To validate the correctness of our table traversal strategies and the effectiveness of JSON-specific payload injection, we developed a simple test application. This lightweight PHP web application uses a MySQL database and includes only a few pages. It allowed us to perform controlled experiments to confirm that our BFS/DFS traversal logic behaves as expected and that payloads are correctly injected into JSON-formatted data without breaking the application logic.

3.2 Experimental Design

This section outlines the experimental methodology used to assess the design goals of Spider-Scents v2. The evaluation is structured to investigate whether the system's enhancements improve the coverage and practical utility of stored XSS detection.

3.2.1 Evaluation Objectives

To guide the evaluation, we define the following evaluation objectives:

- **EO1:** Do the improvements introduced in Spider-Scents v2 (e.g., table traversal and JSON-aware payloads) lead to better coverage or detection capability compared to Spider-Scents v1?
- **EO2:** Is Spider-Scents v1 and v2 usable for web applications developed in languages other than PHP?
- **EO3:** Can the data synthesis algorithm of Spider-Scents improve other security scanners' effectiveness when used as a preparatory step?

3.2.2 Experiment Structure

To answer these questions, the evaluation is divided into three experimental components:

- **Experiment A System Improvements (EO1):** Apply Spider-Scents v2 to previously evaluated PHP applications and compare its performance with the original v1 system to measure the benefits introduced by improved traversal strategies and format-aware payloads.
- **Experiment B Language Generalization (EO2):** Test Spider-Scents v1 and v2 on a set of open-source web applications developed in non-PHP languages (e.g., Python Flask, Node.js Express) to assess its conceptual applicability beyond its original evaluation scope.
- **Experiment C Integration with Scanners (EO3):** Integrate Spider-Scents with external black-box scanners by reusing its application preparation phase to generate enriched input states. Compare scanner behavior both with and without the integration.

3.2.3 Target Applications

To evaluate the effectiveness and generalizability of Spider-Scents v2, we selected a diverse set of web applications that satisfy two key criteria: (1) the ability to deploy and control the application in an isolated testing environment, and (2) support for a MySQL-compatible relational database to enable integration with Spider-Scents' database synthesis techniques.

The target applications are divided into two categories: the PHP applications from the original Spider-Scents paper, and newly selected non-PHP applications.

(1) PHP Applications from the Original Spider-Scents Paper. We reuse a subset of the applications evaluated in the original Spider-Scents study to ensure compatibility with the previous results and allow for direct comparison. We specifically chose this subset because these applications can easily be run in Docker containers, while other applications evaluated in the original study require very old versions of PHP that are difficult to deploy and maintain. These applications are all written in PHP and include:

- **Doctor**
- **Hospital**
- **MyBB**
- **Opencart**
- **Piwigo**
- **User Login**
- **WordPress**
- **CMSMS**

(2) Newly Selected Non-PHP Applications. To evaluate Spider-Scents on diverse modern web stacks, we manually selected a set of open-source applications developed in various programming languages other than PHP. These applications were chosen based on popularity, availability of source code, compatibility with MySQL, and ease of Docker-based deployment. The selected applications span multiple languages and frameworks. Each of these applications is containerized using Docker, and customized configurations were created to ensure proper database setup, application accessibility, and compatibility with Spider-Scents' scanning workflow. We include a table listing each application's name, language, GitHub star count, version and release date.

Table 3.1: Overview of Newly Selected Non-PHP Applications.

Application	Programming Language	Stars	Version	Release Date
mall	Java	80.5K	1.0.3	2024-05
djangoCMS	Python	10.5K	4.1.6	2025-04
Answer	Golang	14.5K	1.5.0	2025-04
Redmine	Ruby on Rails	5.6K	5.1.8	2025-03
Keystone	Node.js	9.6K	6.4.0	2025-02
ghost	Node.js	49.4K	5.11.7	2025-04

3.2.4 Compared Scanners

For Experiment C, we integrated Spider-Scents with four external black-box scanners to assess its enhancement impact on stored XSS detection. The selected scanners—Black Widow, OWASP ZAP, Burp Suite, and Codename SCNR—were previously introduced in Section 2.6. Each scanner represents a different approach to dynamic web application security testing, offering diverse features and scanning strategies.

3.2.5 Evaluation Metrics

To systematically assess the effectiveness of Spider-Scents v2, we adopt a set of evaluation metrics inspired by those used in the original Spider-Scents study [25]. These metrics are designed to capture different aspects of the scanner's capability in identifying and reaching stored XSS vulnerabilities within web applications. The two primary metrics are as follows:

- **Unprotected Outputs**

  This metric quantifies the number of unprotected outputs discovered by the scanner. An unprotected output refers to any HTML-rendered content that is sourced from the database and reflected in the page without proper sanitization. Although not all unprotected outputs lead directly to XSS vulnerabilities, they represent critical potential sink points.

- **Verified Stored XSS Vulnerabilities.**

To evaluate the scanner's effectiveness in detecting actual vulnerabilities, we manually verify whether any of the identified unprotected outputs can be turned into a full stored XSS vulnerability. A reported unprotected output is considered verified if a payload inserted via the application input is executed as JavaScript in a rendered page. This metric represents the confirmed, end-to-end success cases of injection, storage, and execution.

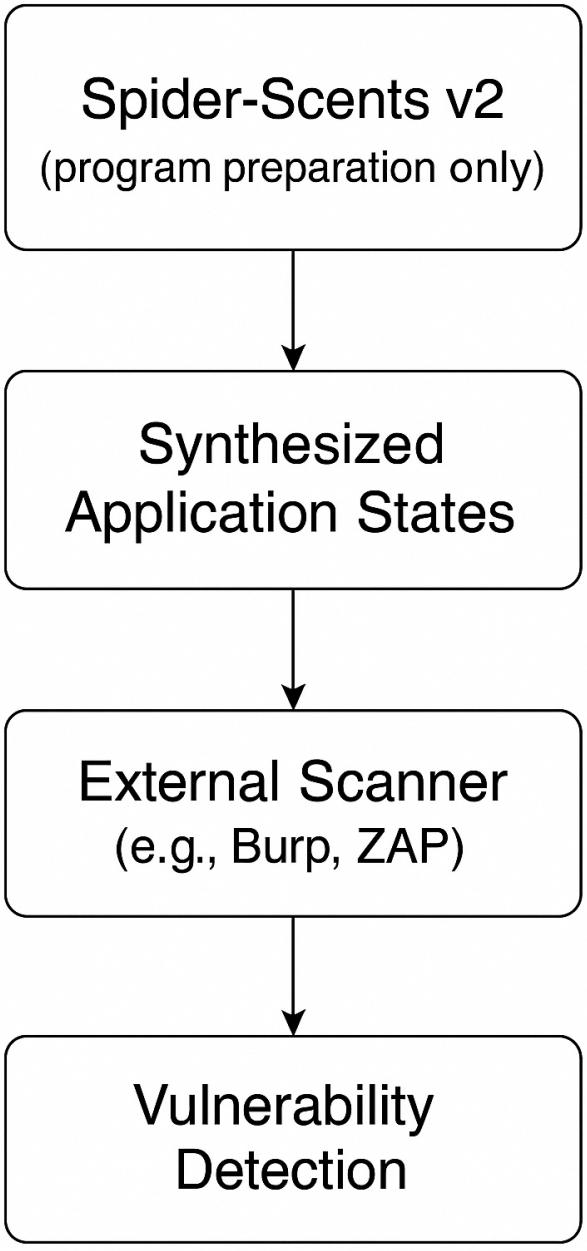

Figure 3.3: Integration with Other Scanner.

4

Results

In this chapter, we present the results of evaluating Spider-Scents v2 across a range of web applications. Section 4.1 presents the comparison between Spider-Scents v1 and v2 in terms of output coverage under different traversal strategies. Section 4.2 examines the applicability and performance of Spider-Scents v1 and v2 on non-PHP applications, demonstrating its generalizability beyond the PHP ecosystem. Finally, Section 4.3 analyzes the indirect impact of data synthesis on third-party scanners.

4.1 Experiment A

Table 4.1 summarizes the number of unprotected outputs identified by Spider-Scents v1 and its improved v2 version using both BFS and DFS traversal strategies. For each application, we also compute the absolute difference in unprotected outputs between v1 and v2. The final column reports how many of the identified unprotected outputs resulted in confirmed stored XSS vulnerabilities after manual validation. Since stored XSS vulnerabilities are pretty difficult to detect, especially in the latest versions of modern applications, even discovering one or two instances can be considered a valuable outcome.

The most prominent improvement was found in **Piwigo**, which exhibited a substantial increase in unprotected outputs, with +7 under BFS and +6 under DFS. **CMSMS** demonstrated a slight decrease (1) in the BFS mode but a moderate increase (+2) in the DFS mode. In contrast, **Hospital** was the only application that showed a consistent reduction in the number of unprotected outputs across both traversal strategies, with 4 for BFS and 1 for DFS. For the remaining applications — **Doctor**, **MyBB**, **Opencart**, **User Login**, and **WordPress** — the number of unprotected outputs remained unchanged between the two versions. Importantly, among all the applications tested, 2 verified stored XSS vulnerabilities were ultimately discovered only in **Piwigo**.

To further analyze the improvements observed, we closely examined the additional unprotected outputs discovered in the **Piwigo** application. Specifically, the scanner identified seven new unprotected database fields:
`piwigo_images.file`, `piwigo_groups.name`, `piwigo_tags.name`, `piwigo_categories.name`, `piwigo_plugins.id`, `piwigo_plugins.version`, and `piwigo_categories.uppercats`.

Table 4.1: Comparison of Unprotected Outputs Across Versions and Verified XSS Confirmed from Spider-Scents v2 Improvements.

Application	**Outputs v1**	**Outputs v2 (BFS)**	**Outputs v2 (DFS)**	**Δ v1→v2 (BFS)**	**Δ v1→v2 (DFS)**	**Verified XSS**
Doctor	8	8	8	0	0	0
Hospital	**33**	29	32	-4	-1	0
MyBB	6	6	6	0	0	0
Opencart	6	6	6	0	0	0
Piwigo	5	12	11	**+7**	**+6**	**2**
User Login	3	3	3	0	0	0
WordPress	7	7	7	0	0	0
CMSMS	18	17	20	-1	**+2**	0

Among these, `piwigo_images.file` and `piwigo_categories.uppercats` were determined to have no corresponding user input sources, rendering them non-vulnerable in the context of stored XSS. The field `piwigo_groups.name` was linked to an input vector but was found to be subject to filtering of dangerous characters such as the `<` symbol, which prevented successful payload injection. Ultimately, the only confirmed stored XSS vulnerabilities were identified in `piwigo_categories.name` and `piwigo_tags.name`.

Although no XSS vulnerabilities were identified in `piwigo_plugins.id` and `piwigo_plugins.version` during this evaluation, these fields still represent potential attack vectors. If an attacker is able to introduce a malicious plugin — such as by developing and uploading a custom plugin containing crafted payloads — these fields could be leveraged to trigger stored XSS under certain conditions. The details are demonstrated in Table 4.2.

Table 4.2: Analysis of Additional Unprotected Outputs in Piwigo.

Field	**Input Source**	**Filtering**	**Vulnerable**
`piwigo_images.file`	No	N/A	No
`piwigo_groups.name`	Yes	Yes (`<` filtered)	No
`piwigo_tags.name`	Yes	No	**Yes**
`piwigo_categories.name`	Yes	No	**Yes**
`piwigo_plugins.id`	Unknown	Unknown	No evidence
`piwigo_plugins.version`	Unknown	Unknown	No evidence
`piwigo_categories.uppercats`	No	N/A	No

The experimental results suggest that our traversal strategies had limited impact on most applications, with the exception of Piwigo, where notable differences were observed. This indicates that the effectiveness of traversal strategies is not solely determined by the number of tables in the database, but is also strongly influenced by the complexity of inter-table dependencies.

4.2 Experiment B

To assess the generalizability of Spider-Scents beyond PHP-based applications, we conducted an additional evaluation on a set of popular open-source projects developed in other programming languages, including Java, Python, Go, Ruby on Rails, and Node.js. As shown in Table 4.3, Spider-Scents v2 was able to identify unprotected outputs in two cases: `Answer` (written in Go) and `Ghost` (based on Node.js) and compared to v1, it can identify more outputs on `Answer`. In the remaining applications, both versions detected no unprotected outputs. Importantly, no verified stored XSS vulnerabilities were found in any of the evaluated applications.

Table 4.3: Spider-Scents Evaluation on Non-PHP Applications.

Application	**Programming Language**	**Outputs v1**	**Outputs v2**	**Verified XSS**
mall	Java	0	0	0
djangoCMS	Python	0	0	0
Answer	Go	3	4	0
Redemin	Ruby on Rails	0	0	0
Keystone	Node.js	0	0	0
Ghost	Node.js	6	6	0

In the case of the `Answer` application, Spider-Scents v2 identified four unprotected output fields: `answer.parsed_text`, `question.parsed_text`, `tag.parsed_text`, and `site_info.content`. We observed that the first three fields are derived from user-submitted content that has been parsed and sanitized by the application prior to rendering. As a result, any injected payloads are effectively neutralized through escaping or filtering mechanisms, rendering them non-executable in the context of stored XSS. In contrast, `site_info.content` was found to be a static output without any corresponding user input vector.

Notably, the field `site_info.content` was not detected in Spider-Scents v1, and its inclusion in v2 demonstrates an improvement in the tool's ability to identify unprotected outputs embedded in JSON-formatted storage structures. Table 4.4 summarizes the characteristics of the unprotected outputs identified in the Answer platform.

Table 4.4: Analysis of Unprotected Outputs in Answer Platform.

Field	**Input Source**	**Sanitization**	**Vulnerable**	**New in v2**
answer.parsed_text	Yes	Yes	No	No
question.parsed_text	Yes	Yes	No	No
tag.parsed_text	Yes	Yes	No	No
site_info.content	No	N/A	No	**Yes (JSON)**

The evaluation of the `Ghost` platform yielded a similar outcome to that of `Answer`. Although Spider-Scents v2 identified several unprotected output fields, all of them are associated with user input that is rigorously sanitized by the application before rendering. As a result, none of these outputs were found to be vulnerable, and no verified stored XSS vulnerabilities were discovered in this application.

However, it is worth noting that the specific version of `Ghost` used in our evaluation is known to be affected by a separate file-system-based stored XSS vulnerability, where an attacker may upload a crafted SVG file containing malicious JavaScript. This vulnerability, documented as CVE-2024-23724 [33], enables stored XSS without relying on database storage. Since our experimental framework focuses exclusively on database-stored data flows, this class of vulnerability falls outside the current scope of our evaluation.

4.3 Experiment C

To evaluate the broader utility of the data synthesis algorithm, we examined whether the modified database state influenced the detection capabilities of four external scanners: ZAP, Burp Suite, BlackWidow, and Codename Scnr. The results are summarized in Table 4.5. Unlike Spider-Scents, the other scanners do not report unprotected outputs; instead, they directly identify and report potential XSS candidates based on observed payload execution. Each scanner was executed after data synthesis was applied, and their findings were recorded in terms of R (number of reported XSS candidates) and V (number of verified stored XSS instances confirmed). Based on the results, the data synthesis algorithm demonstrably enhanced the detection capabilities of Burp Suite and Codename Scnr on two target applications.

For Burp Suite, the synthesis process led to a newly reported stored XSS candidate in MyBB (R=1), although this case was not verified (V=0). In contrast, on CMSMS, Burp reported two candidates (R=2), one of which was successfully verified as a stored XSS vulnerability (V=1). These findings suggest that the synthetic payloads introduced new interaction patterns or data traces that made previously hidden injection points detectable.

Codename Scnr, which reported zero findings across all applications prior to synthesis, identified four stored XSS candidates (R=4) in CMSMS after the synthetic modifications. Although none were verified (V=0), this marks a clear increase in scanner visibility, indicating that the synthesized data enabled the scanner to reach previously inaccessible or inactive input flows.

To assess whether the data synthesis algorithm introduced unintended changes to the contents of database, we manually inspected the corresponding backend databases before and after applying the synthesis. The results are summarized in Table 4.6. A value of “yes” indicates that the database contents were modified as a result of the synthetic payloads generated by Spider-Scents, while “no” suggests no detectable change.

As shown in the table, five out of eight applications exhibited database modifica-

Table 4.5: Impact of Data Synthesis on Stored XSS Detection by External Scanners.

Scanner	**ZAP**		**Burp Suite**		**BlackWidow**		**Codename Scnr**	
	R	V	R	V	R	V	R	V
Doctor	0	0	0	0	0	0	0	0
Hospital	0	0	0	0	0	0	0	0
MyBB	0	0	1	0	0	0	0	0
Opencart	0	0	0	0	0	0	0	0
Piwigo	0	0	0	0	0	0	0	0
User Login	0	0	0	0	0	0	0	0
WordPress	0	0	1	0	0	0	0	0
CMSMS	0	0	2	1	0	0	4	0

tions post-synthesis. These include **MyBB**, **Opencart**, **Piwigo**, **WordPress**, and **CMSMS**, suggesting that the synthetic data successfully are inserted into persistent storage. On the other hand, Doctor and User Login showed no evidence of database-level change, potentially due to input sanitization or architectural constraints that prevent state mutation via crafted payloads.

Table 4.6: Impact of Data Synthesis on Application Database State.

Application	**Database Modified**
Doctor	No
Hospital	No
MyBB	Yes
Opencart	Yes
Piwigo	Yes
User Login	No
WordPress	Yes
CMSMS	Yes

5

Conclusion

In this chapter, we will discuss Spider-Scents v2 from a overall perspective, highlighting its current achievements and identifying potential areas for future work.

5.1 Discussion

Although only a small number of new vulnerabilities were confirmed, this result still demonstrates the potential of our improvements, given the known difficulty of detecting stored XSS in complex data flows. In this project, we have focused on a limited set of open-source web applications and scanners, which substantially restricts the scope of the experiments. Including more web applications and additional scanning tools in future evaluations would allow for a broader assessment and more conclusive comparisons.

In the following sections, we will discuss the project more thoroughly. We begin by reflecting on the key contributions and methods of Spider-Scents v2, followed by an overview of the implementation and evaluation results. Finally, we discuss some of the general limitations of this project and explore opportunities for future work.

5.1.1 Summary of Current Work

Spider-Scents v2 extends the database table traversal algorithm of its predecessor, Spider-Scents v1. In v2, the traversal order of tables is determined by analyzing the foreign key relationships between tables in the database schema. Specifically, the schema is abstracted as a directed graph, where nodes represent tables and edges represent foreign key constraints. We then apply both BFS and DFS algorithms on this graph to generate meaningful traversal orders for table iteration. This innovative approach overcomes the limitations of the random and dictionary-based traversal orders used in v1, thereby increasing the likelihood of identifying vulnerabilities.

Another key enhancement in Spider-Scents v2 is the customization of payloads based on the data format of each cell. The main idea is to identify the structure of data stored in database tables and to generate tailored payloads accordingly. In particular, we focused on customizing payloads for JSON-formatted structured data, which has become increasingly common in modern web applications. This approach increases the scanner's ability to detect vulnerabilities in contemporary application scenarios, where traditional generic payloads might be less effective.

In the subsequent evaluation phase, we evaluate Spider-Scents v2 on the original PHP applications, then we conducted two complementary sets of experiments. First, to assess the generalizability, we selected a group of highly popular and actively maintained non-PHP web applications. We deliberately chose their most recent versions to ensure our testing was relevant to contemporary deployment environments. Second, we evaluated whether Spider-Scents v2 could be seamlessly integrated with other widely used vulnerability scanners to enhance their ability to detect stored XSS vulnerabilities. These two complementary evaluations allowed us to demonstrate both the standalone effectiveness and the integrability of Spider-Scents v2 in real-world web security testing scenarios.

Based on the final experimental results, we can draw the following conclusions. First, Spider-Scents v2 identified a larger number of unprotected outputs in the original programs compared to its predecessor, and it also uncovered two confirmed stored XSS vulnerabilities. This demonstrates that our improved database table traversal algorithm enhances the scanner's ability to detect potential vulnerabilities. Second, although we did not find any confirmed stored XSS vulnerabilities in the selected non-PHP applications, we still observed a substantial number of unprotected outputs. These unprotected outputs indicate that the applications have not fully adopted the best practice of output encoding to prevent XSS, suggesting potential avenues for future exploitation. Notably, in one of these applications, our structure-aware, customized payload strategy successfully reported an unprotected output, further demonstrating its practical effectiveness. Finally, in the integration experiments, we observed that incorporating Spider-Scents v2 with other vulnerability scanners enabled them to discover additional unprotected outputs and even verified vulnerabilities that they would not have detected on their own.

5.1.2 Limitations

While Spider-Scents v2 demonstrates significant improvements in stored XSS vulnerability detection, there are some limitations to our current work that should be acknowledged.

First, the effectiveness of Spider-Scents v2's database traversal algorithm depends on the availability of explicit foreign key relationships in the database schema. However, many modern web applications deliberately avoid or minimize the use of foreign keys in their schema design to improve database performance. As a result, identifying accurate foreign key relationships in such environments often requires manual inference or additional analysis, which may introduce uncertainty or incompleteness into the traversal graph. Additionally, our current evaluation has been limited to applications using MySQL-based relational databases. We have not yet tested Spider-Scents v2 on other types of databases, particularly NoSQL systems, and it is unclear how well the tool would perform in those contexts.

Second, although Spider-Scents v2 introduces a structure-aware payload generation strategy, its current implementation primarily targets JSON-formatted data structures. While this represents a significant step towards handling modern structured data, it does not comprehensively address other popular or complex formats such

as XML, YAML, or deeply nested objects. These alternative data formats are increasingly used in contemporary web applications, and their structural complexity may limit the scanner's effectiveness in detecting stored XSS vulnerabilities when relying solely on the current payload customization strategy.

Finally, it is important to note that our experimental evaluations were conducted in controlled environments using containerized open-source applications(Docker). While this approach ensures reproducibility and comparability of results, it does not fully capture the complexity and variety of real-world production systems. In actual deployment scenarios, factors such as custom configurations, access control mechanisms, and integration with proprietary services may influence the tool's effectiveness and introduce additional challenges that were beyond the scope of this study.

5.1.3 Future Work

One promising direction for future work is to integrate static analysis techniques to further enhance the detection capabilities of Spider-Scents v2. By combining dynamic database-based scanning with static source code analysis, it may be possible to achieve more comprehensive identification of stored XSS vulnerabilities, especially in cases where certain data flows remain hidden during dynamic testing.

Another important avenue for future research is to conduct experiments in settings that more closely resemble real-world deployment environments. This includes using proprietary web applications, incorporating complex authentication and authorization mechanisms, and evaluating Spider-Scents v2 within modern CI/CD pipelines. Such experiments would help validate the tool's robustness and practical applicability in diverse production scenarios.

Additionally, extending Spider-Scents v2's scope to include vulnerabilities introduced by modern front-end frameworks and client-side rendering logic would help address an increasingly important class of security issues. Many stored XSS vectors manifest primarily through dynamic rendering on the client side, and incorporating detection for these scenarios would further enhance the tool's overall coverage.

5.2 Conclusion

Spider-Scents v2 is an important extension of the original Spider-Scents gray-box scanner, which was designed specifically to detect stored XSS vulnerabilities. The enhancements we introduced — most notably, advanced table traversal strategies and payload customization for JSON — enable Spider-Scents v2 to identify more unprotected outputs and stored XSS vulnerabilities than its predecessor. Our evaluations further confirm the tool's generalizability across web applications written in different programming languages, showcasing its versatility in real-world scenarios. Finally, our integration experiments suggest that Spider-Scents v2 can be incorporated into other vulnerability scanners to improve their ability to uncover stored

XSS vulnerabilities and unprotected outputs, highlighting its potential as a complementary tool in comprehensive security testing workflows.

Bibliography

[1] Abdullah Alsharif, Gregory M. Kapfhammer, and Phil McMinn. "DOMINO: Fast and Effective Test Data Generation for Relational Database Schemas". In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 2018, pp. 12–22. DOI: 10.1109/ICST.2018.00012.

[2] Muzun Althunayyan, Neetesh Saxena, Shancang Li, and Prosanta Gope. "Evaluation of Black-Box Web Application Security Scanners in Detecting Injection Vulnerabilities". In: *Electronics* 11.13 (2022). ISSN: 2079-9292. DOI: 10.3390/electronics11132049. URL: https://www.mdpi.com/2079-9292/11/13/2049.

[3] Jeroen Castelein, Maurício Aniche, Mozhan Soltani, Annibale Panichella, and Arie van Deursen. "Search-based test data generation for SQL queries". In: *Proceedings of the 40th international conference on software engineering*. 2018, pp. 1220–1230.

[4] An Chen, JiHo Lee, Basanta Chaulagain, Yonghwil Kwon, and Kyu Hyung Lee. "Synthdb: Synthesizing database via program analysis for security testing of web applications". In: *Network and Distributed System Security Symposium*. 2023.

[5] Cobalt. *How to Fix the Top 5 Web App Vulnerabilities*. 2024. URL: https://www.cobalt.io/blog/top-web-application-vulnerabilities.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 3rd. MIT Press, 2009.

[7] Douglas Crockford. *The JSON Data Interchange Format*. Accessed: 2025-04-15. n.d. URL: https://www.json.org/json-en.html.

[8] Johannes Dahse and Thorsten Holz. "Static detection of Second-Order vulnerabilities in web applications". In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, pp. 989–1003.

[9] David Dittrich and Erin Kenneally. *The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research*. Tech. rep. https://www.dhs.gov/sites/default/files/publications/CSD-MenloPrinciplesCORE-20120803_0.pdf. US Department of Homeland Security, Aug. 2012.

[10] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. "KameleonFuzz: evolutionary fuzzing for black-box XSS detection". In: *Proceedings of the 4th ACM conference on Data and application security and privacy*. 2014, pp. 37–48.

[11] Ecsypno. *Codename SCNR - A Modern Web Application Security Scanner*. https://ecsypno.com/pages/codename-scnr. Accessed: 2025-06-05. 2025.

[12] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. "Black Widow: Blackbox Data-driven Web Scanning". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 1125–1142. DOI: 10.1109/SP40001.2021.00022.

[13] ESET Research. *Winter Vivern attacks Roundcube webmail servers of governments in Europe through zero-day vulnerability*. Accessed: 2025-06-01. 2023. URL: https://www.eset.com/int/about/newsroom/press-releases/research/eset-research-winter-vivern-attacks-roundcube-webmail-servers-of-governments-in-europe-through-zero/.

[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*. 2021. DOI: 10.48550/arXiv.2108.08455. arXiv: 2108.08455 [cs.CR].

[15] Google Project Zero. *Vulnerability Disclosure Policy*. https://googleprojectzero.blogspot.com/p/vulnerability-disclosure-policy.html. Accessed: 2025-06-01. 2021.

[16] Shashank Gupta and B. B. Gupta. "Evaluation and monitoring of XSS defensive solutions: a survey, open research issues and future directions". In: *Journal of Ambient Intelligence and Humanized Computing* 10.11 (2019), pp. 4377–4405. DOI: 10.1007/s12652-018-1118-3.

[17] Behnaz Hassanshahi, Hyunjun Lee, and Paddy Krishnan. "Gelato: Feedback-driven and guided security analysis of client-side web applications". In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2022, pp. 618–629.

[18] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. "Precise alias analysis for static detection of web application vulnerabilities". In: *Proceedings of the 2006 workshop on Programming languages and analysis for security*. 2006, pp. 27–36.

[19] Rody Kersten, Kasper Luckow, and Corina S. Psreanu. "POSTER: AFL-based Fuzzing for Java with Kelinci". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2511–2513. ISBN: 9781450349468. DOI: 10.1145/3133956.3138820. URL: https://doi.org/10.1145/3133956.3138820.

[20] Tasos Laskos. *The Arachni Chronicles*. https://ecsypno.com/blogs/articles/the-arachni-chronicles. Accessed: 2025-04-12. 2023.

[21] Sean McAllister, Engin Kirda, and Christopher Kruegel. "Leveraging user interactions for in-depth testing of web applications". In: *Recent Advances in Intrusion Detection: 11th International Symposium, RAID 2008, Cambridge, MA, USA, September 15-17, 2008. Proceedings 11*. Springer. 2008, pp. 191–210.

[22] MDN Web Docs. *Content Security Policy (CSP)*. https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP. Accessed: 2025-06-01. 2023.

[23] Ali Mesbah, Engin Bozdag, and Arie van Deursen. "Crawling AJAX by Inferring User Interface State Changes". In: *2008 Eighth International Conference on Web Engineering*. 2008, pp. 122–134. DOI: 10.1109/ICWE.2008.24.

[24] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. "Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes". In: *ACM Trans. Web* 6.1 (Mar. 2012). ISSN: 1559-1131. DOI: 10.1145/2109205.2109208. URL: https://doi.org/10.1145/2109205.2109208.

[25] Eric Olsson, Benjamin Eriksson, Adam Doupé, and Andrei Sabelfeld. "SpiderScents: Grey-box Database-aware Web Scanning for Stored XSS". In: *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, 2024, pp. 6741–6758. ISBN: 978-1-939133-44-1. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/olsson.

[26] OWASP Foundation. *Cross Site Scripting Prevention Cheat Sheet*. Accessed 2025-05-30. 2024. URL: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html.

[27] OWASP Foundation. *OWASP Zed Attack Proxy (ZAP)*. https://www.zaproxy.org/. Accessed: 2025-04-12. 2024.

[28] Jihyeok Park, Hongki Lee, and Sukyoung Ryu. "A Survey of Parametric Static Analysis". In: *ACM Comput. Surv.* 54.7 (July 2021). ISSN: 0360-0300. DOI: 10.1145/3464457. URL: https://doi.org/10.1145/3464457.

[29] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. "jäk: Using dynamic analysis to crawl and test modern web applications". In: *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015. Proceedings 18*. Springer. 2015, pp. 295–316.

[30] PortSwigger Ltd. *Burp Suite Web Vulnerability Scanner*. https://portswigger.net/burp. Accessed: 2025-04-12. 2024.

[31] PortSwigger Web Security Academy. *Cross-site scripting (XSS)*. Accessed: 2025-04-07. 2023. URL: https://portswigger.net/web-security/cross-site-scripting.

[32] Positive Technologies. *What is a cross-site scripting (XSS) attack?* 2023. URL: https://global.ptsecurity.com/analytics/knowledge-base/what-is-a-cross-site-scripting-xss-attack.

[33] Rhino Security Labs. *CVE-2024-23724: Stored XSS in Ghost via Malicious SVG Upload*. Accessed: 2025-05-02. URL: https://github.com/RhinoSecurityLabs/CVEs/tree/master/CVE-2024-23724.

[34] Orpheas van Rooij, Marcos Antonios Charalambous, Demetris Kaizer, Michalis Papaevripides, and Elias Athanasopoulos. "webfuzz: Grey-box fuzzing for web applications". In: *Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I 26*. Springer. 2021, pp. 152–172.

[35] Xuyan Song, Ruxian Zhang, Qingqing Dong, and Baojiang Cui. "Grey-box fuzzing based on reinforcement learning for XSS vulnerabilities". In: *Applied Sciences* 13.4 (2023), p. 2482.

[36] Antonín Steinhauser and Petr Tma. "Database Traffic Interception for Graybox Detection of Stored and Context-sensitive XSS". In: *Digital Threats* 1.3 (Aug. 2020). DOI: 10.1145/3399668. URL: https://doi.org/10.1145/3399668.

[37] He Su, Feng Li, Lili Xu, Wenbo Hu, Yujie Sun, Qing Sun, Huina Chao, and Wei Huo. "Splendor: Static Detection of Stored XSS in Modern Web Applications". In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2023. Seattle, WA, USA: Association for Computing Machinery, 2023, pp. 1043–1054. ISBN: 9798400702211. DOI: 10.1145/3597926.3598116. URL: https://doi.org/10.1145/3597926.3598116.

[38] Yi Wang, Zhoujun Li, and Tao Guo. "Program Slicing Stored XSS Bugs in Web Application". In: *2011 Fifth International Conference on Theoretical Aspects of Software Engineering*. 2011, pp. 191–194. DOI: 10.1109/TASE.2011.43.

A

Appendix 1

A.1 Scanner Configuration

Spider-Scents v2: Spider-Scents v2 shares the same prerequisites and environment setup as the original Spider-Scents scanner, which are detailed in the projects repository at `https://github.com/Spider-Scents/dbfuzz`. The scanning process is initiated using the following command:

```
pipenv run script  --config 'webapp_config.ini' --insert-empty
↪  --reset-fuzzing --reset-scanning --sensitive-rows --primary-keys
↪  --traversal column --traversal-order dfs
```

This introduces a new `-traversal-order` option, allowing users to select between BFS and DFS traversal orders.

Burp Suite: Start by launching the Burp Suite application and navigating to the target web applications login page using the embedded browser within Burp Suite. After logging in, initiate an active scan on the target application to comprehensively identify potential security vulnerabilities.

ZAP: To scan vulnerabilities with ZAP, configure your browser to proxy through ZAP, establish an authenticated session in your browser and define ZAP's authentication settings for your application, then spider your target with both traditional spider and ajax spider, and finally run an active scan as the authenticated user.

Scnr: The following command is used to scan the application with Scnr:

```
scnr scan --url [url] --authentication-type form  --login-url
↪  [login_url] --username-field []  --password-field []--username []
↪  --password [] --report-format html --output report.html
```

Black Widow: The following command was used to run the Black Widow scanner:

```
python3 crawl.py
```