applied sciences

Article

State-Sensitive Black-Box Web Application Scanning for Cross-Site Scripting Vulnerability Detection

Tianxiang Zhang [1,2], Hui Huang [1,2,*], Yuliang Lu [1,2], Kailong Zhu [1,2] and Jiazhen Zhao [1,2]

[1] College of Electronic Engineering, National University of Defense Technology, Hefei 230037, China; zhangtx21@nudt.edu.cn (T.Z.)
[2] Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation, Hefei 230037, China
* Correspondence: huanghui17@nudt.edu.cn

Abstract: Black-box web application scanning has been a popular technique to detect Cross-Site Scripting (XSS) vulnerabilities without prior knowledge of the application. However, several limitations lead to low efficiency of current black-box scanners, including (1) the scanners waste time by repetitively visiting similar states, such as similar HTML forms of two different products, and (2) using a First-In-First-Out (FIFO) fuzzing order for the collected forms has led to low efficiency in detecting XSS vulnerabilities, as different forms have different potential possibilities of XSS vulnerability. In this paper, we present a state-sensitive black-box web application scanning method, including a filtering method for excluding similar states and a heuristic ranking method for optimizing the fuzzing order of forms. The filtering method excludes similar states by comparing readily available characteristic information that does not require visiting the states. The ranking method sorts forms based on the number of injection points since it is commonly observed that forms with a greater number of injection points have a higher probability of containing XSS vulnerabilities. To demonstrate the effectiveness of our scanning method, we implement it in our black-box web scanner and conduct experimental evaluations on eight real-world web applications within a limited scanning time. Experimental results demonstrate that the filtering method improves the code coverage about 17% on average and the ranking method helps detect 53 more XSS vulnerabilities. The combination of the filtering and ranking methods helps detect 81 more XSS vulnerabilities.

Keywords: black-box scanner; fuzzing; state-sensitive; cross-site scripting (XSS); HTML forms

check for updates

Citation: Zhang, T.; Huang, H.; Lu, Y.; Zhu, K.; Zhao, J. State-Sensitive Black-Box Web Application Scanning for Cross-Site Scripting Vulnerability Detection. *Appl. Sci.* **2023**, *13*, 9212. https://doi.org/10.3390/app13169212

Academic Editor: Christos Bouras

Received: 11 July 2023
Revised: 8 August 2023
Accepted: 10 August 2023
Published: 13 August 2023

Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Web applications have became increasingly popular due to their convenience in use. However, various web vulnerabilities [1–3] also threaten the security of web applications and user information. Cross-Site Scripting (XSS) vulnerability manifests as an injection flaw in web applications, caused by untrusted input propagating to sensitive locations in the web application. It has consistently maintained its position among the top ten vulnerabilities documented by the Open Web Application Security Project (OWASP) since 2003, as also evidenced in their latest report released in 2021 [4]. XSS vulnerabilities have been a longstanding and significant threat to the security of web applications.

Web application scanners play a crucial role in detecting web vulnerabilities. Different from white- [5] and grey-box [6] scanners, black-box scanners do not require prior knowledge of the application, such as source code. They acquire the information for detecting vulnerabilities by interacting with web applications through crawlers. Black-box scanners explore attack surfaces by visiting web pages to discover URLs, HTML forms, and other input fields. Researchers are committed to designing black-box scanners that can cover as many attack surfaces as possible.

In 2012, Doupé et al. [7] proposed modeling the state of applications to achieve better coverage and cover as many attack surfaces as possible. However, modern web applications have rich dynamic characteristics due to the presence of the JavaScript program, client-side events, and server-side statefulness. These dynamic characteristics allow network requests to change the state of the server. For example, clicking a button can change the DOM to generate a new link or field. Modeling the interaction between server and client code with scanners is very challenging. In 2021, Eriksson et al. [8] proposed an effective navigation model (introduced in Section 3.2.1) construction method to model the client- and server-side state to a certain extent in Black Widow, a state-of-the-art black-box web scanner. The model covers multiple methods of application interaction, including GET requests, JavaScript events, HTML forms, and iframes, and can construct dependencies between various states. For example, users may have to hover over the menu before clicking the button, or submit a series of forms in order before purchasing the product.

Similar states often exist in web applications. Especially for XSS, there are often multiple similar HTML forms in an application. For example, in most web applications with product information management functionality, each product will have a corresponding form for modifying information. A common method for restricting repetitive visits to similar states in current black box scanners is to limit the number of visits to URLs with the same path. On the other hand, the fuzzing order of the form in current black-box scanners refers to the sequence in which the scanners discover the forms. However, these straightforward methods suffer from several **limitations: (1)** there are numerous repetitive visits to similar states in the scanning process, especially for the state of HTML forms; and **(2)** they lack an appropriate method for judging the importance of HTML forms to guide the fuzzing order of them. Both of these limitations result in low efficiency of black-box scanners, which further leads to the missing of some XSS vulnerabilities within a limited scanning time.

To overcome these limitations, we propose a state-sensitive black-box web application scanning method, including a filtering method for excluding similar states and a heuristic ranking method for optimizing the the fuzzing order of forms. In our filtering method, we mainly focus on the similarity of HTML forms and judge them based on the feature information of the form and the page on which it is located. It reduces the time spent by the scanner in visiting and fuzzing similar states. In our ranking method, we propose that a higher priority should be assigned to forms with more injection points in the context of fuzzing. To evaluate the effectiveness of our methods, we present our black-box scanner SSBW by incrementally implementing the filtering method and ranking method on Black Widow. Experimental evaluation illustrates that the code coverage on eight applications of the scanner with our filtering method increased by 17% on average, with the highest increase being 57%. The scanner with the ranking method can detect 53 more vulnerabilities in three out of eight applications compared to the scanner without the ranking method. The combination of the filtering and ranking methods helps to get an average increase of about 19% in code coverage and detect 81 more XSS vulnerabilities.

In summary, our contributions are the following:

- We have designed a stricter filtering approach that excludes similar states without the need to actually visit these states. This enhancement makes the scanner more attuned to different states, resulting in fewer repetitive visits to similar states. This improvement, in turn, boosts the code coverage of the black-box web scanner within a limited scanning time.
- In order to fuzz a greater number of injection points within a specific time frame, we have devised a heuristic ranking technique for HTML forms. This approach prioritizes the forms with a higher count of injection points, thereby enhancing the efficiency of the black-box web scanner in identifying XSS vulnerabilities.
- We implement a filtering method and a ranking method in our black-box web scanner SSBW. These methods were experimentally evaluated on eight real-world applications. Our evaluation results show that SSBW can attain an average increase of about 19% in code coverage and detect 81 more XSS vulnerabilities compared to Black Widow within a limited scanning time.

2. Background

This section is comprised of three main parts. Firstly, it introduces the principle and three primary classifications of XSS vulnerabilities. Secondly, it discusses three different detection methods. Finally, it presents the challenge addressed in this paper.

2.1. XSS Vulnerability

Cross-Site Scripting (XSS) vulnerabilities arise from flaws in how web applications handle user-generated content. The core principle behind XSS vulnerabilities revolves around the ability for malicious actors to inject unauthorized code or scripts into trusted websites [9]. Utilizing XSS vulnerabilities can obtain user privacy information, spread webworms and cause denial of service. XSS vulnerabilities can be categorized into three primary types [10,11]: reflected XSS, stored XSS, and DOM-based XSS. The primary objective of this paper is to detect reflected and stored XSS vulnerabilities.

- **Reflected XSS** occurs when a malicious script, injected into an HTTP request, is directly included in the response without appropriate sanitization, leading to its execution by the web browser. This vulnerability commonly manifests in scenarios where malicious scripts are embedded within search content, thereby becoming integral components of search results or error messages. Reflected XSS is also referred to as non-persistent XSS since the malicious scripts are not permanently stored on the server.
- **Stored XSS** refers to the practice of storing malicious scripts on the server. When a user requests relevant content, the malicious script is included as part of the response, leading the browser to execute it. A common instance of this vulnerability involves injecting malicious scripts into message boards. Upon viewing the content within this message, the injected script is executed. Since the malicious scripts persistently reside on the server, Stored XSS is alternatively referred to as persistent XSS.
- **DOM-based XSS** occurs when the client-side script alters the Document Object Model (DOM) using a malicious script. Unlike stored and reflected XSS, the malicious script associated with DOM-based XSS can be inserted into the URL as the value of a particular DOM object or HTML element, without ever reaching the server.

2.2. XSS Vulnerability Detection

According to the analysis methodology, the detection of XSS vulnerabilities can be classified into three distinct categories: static analysis, dynamic analysis, and a combination of both known as hybrid analysis. Static analysis [5,12–14] focuses on the analysis of code structures and functions within the source code of web applications, without taking into account the runtime state. While it excels in conducting a thorough examination of the application's source code to minimize false negatives, it is susceptible to generating false positives. Hybrid analysis [3,6,15–17] incorporates a comprehensive evaluation of both the source code and runtime state of the application, thereby mitigating false positives to a certain degree. However, the source code of an application may be unavailable in certain cases. Consequently, dynamic analysis has become a prevalent technique to detect vulnerabilities [18] utilized in such circumstances. A concise overview of several studies on dynamic analysis is presented as follows.

Doupé et al. [7] introduced a methodology for inferring server-side states during browsing, aiming to enhance code coverage. Their approach was implemented within state-aware crawlers, which facilitated the clustering of similar pages. Nevertheless, this crawler solely takes into account static HTML links and forms for detecting state alterations. The failure to accommodate the dynamic attributes inherent in contemporary web applications may result in the omission of dynamically triggered links and forms.

Pellegrino et al. [19] incorporated client events into their methodology to discover additional client-side states. Through dynamic analysis of the client-side JavaScript program, their approach enables the identification of dynamically generated URLs and forms.

However, the support for client-side events is restricted and does not encompass the comprehensive utilization of JavaScript in contemporary applications.

Eriksson et al. [8] proposed an effective navigation model construction method to model the client- and server-side state to a certain extent in Black Widow. This method provides a relatively comprehensive modeling of the dynamic characteristics of web applications. However, it fails to fully utilize the modeled state information, resulting in low efficiency in vulnerability detection. Our work is directly related to Black Widow. And we have evaluated our method based on it. Prior to this, Black Widow had been evaluated based on open-source tools developed by the community [20–22] and academic black-box scanners [7,19]. The overall performance of Black Widow in code coverage and vulnerability detection is superior to these scanners.

From another perspective, Clustering similar pages can reduce the access to forms with similar functionality and improve the efficiency of vulnerability detection. Existing research mainly uses visual analysis [23–26] or webpage text content [27–29] to determine similar pages. Although these methods are effective in their application scenarios, they are not suitable for our research objectives.

The work that best matches our research objectives is ReScan [30], a middleware framework that can be used for black-box web application scanners. Unfortunately, ReScan cannot be directly applied to Black Widow, which hinders our experimental evaluation.

2.3. Challenges

The state-of-the-art black-box web scanner, Black Widow, fully models the dynamic characteristics of web programs to a certain extent. The strategy taken by Black Widow is to build a navigation model for the web application, a transformation graph between client-side states. State refers to the content of the client page. It models the client-side state to further map the server state. The transformation between the state mainly includes the GET requests, form submissions, iframes and JavaScript events.

Challenge 1: There are numerous repetitive visits to similar states in the scanning process, especially for the state of HTML forms. Based on Black Widow's excellent modeling of web applications, there are certain flaws in its use of the model. Especially due to insufficient restrictions on similar states, the scanner spends a lot of time on repetitive visits to similar states. For example, every product page http://example.com/product.php?id=X contains a form http://example.com/product.php?id=X&action=edit for modifying information. X represents the number of the product. Ideally, a form for editing the product only needs to appear once in the navigation model. However, insufficient restrictions may cause undesirable repetition. It will significantly reduce the efficiency of the scanner. This inspired us to design a filtering method for similar states.

Challenge 2: Current black-box web scanners lack an appropriate method for judging the importance of HTML forms to guide the fuzzing order of them. In Black Widow and other black-box scanners [7,19], there is only a limited connection between HTML forms and XSS vulnerabilities. The fuzzing order of the form in current black-box scanners refers to the sequence in which the scanners discover the forms. They lack an appropriate method for judging the importance of HTML forms. This also leads to low efficiency of detecting XSS vulnerabilities, further resulting in missing some XSS vulnerabilities within a limited scanning time. It is inspired by previous research [31–33] on fuzzing of other types of applications. We attempt to explore information in the state that is closely related to XSS vulnerabilities. Based on this information, a ranking method is designed to provide priority to fuzzing specific HTML forms.

3. Approach

Motivated by the challenges of Section 2.3, this section aims to introduce our approach for black-box web application scanning. We implement our approach as a black-box scanner SSBW that can be divided into two modules: a state-sensitive crawler, responsible for crawling web application information, and a fuzzer, dedicated to XSS vulnerability verification. The two key components of our approach are the filtering method of the navigation model construction process in the crawler and the heuristic ranking method in the traversal process. Figure 1 provides an overview of our scanner and the positions of the two key components of our approach.

The goal of the crawler is to obtain the information from the web application quickly, comprehensively and accurately. The most important information for detecting XSS vulnerabilities is the form in the web application. Thus, in scenarios where the ability to identify new forms is limited, we believe that optimizing the efficiency of XSS vulnerability detection relies on two crucial factors: minimizing the fuzzing of similar forms and prioritizing the fuzzing of forms with more injection points. We employ a filtering method in the crawler to exclude similar forms and the fuzzing order of the forms is determined by the traversal order of the forms. In the fuzzing module of this paper, we maintain consistency with the fuzzer implemented by Black Widow.

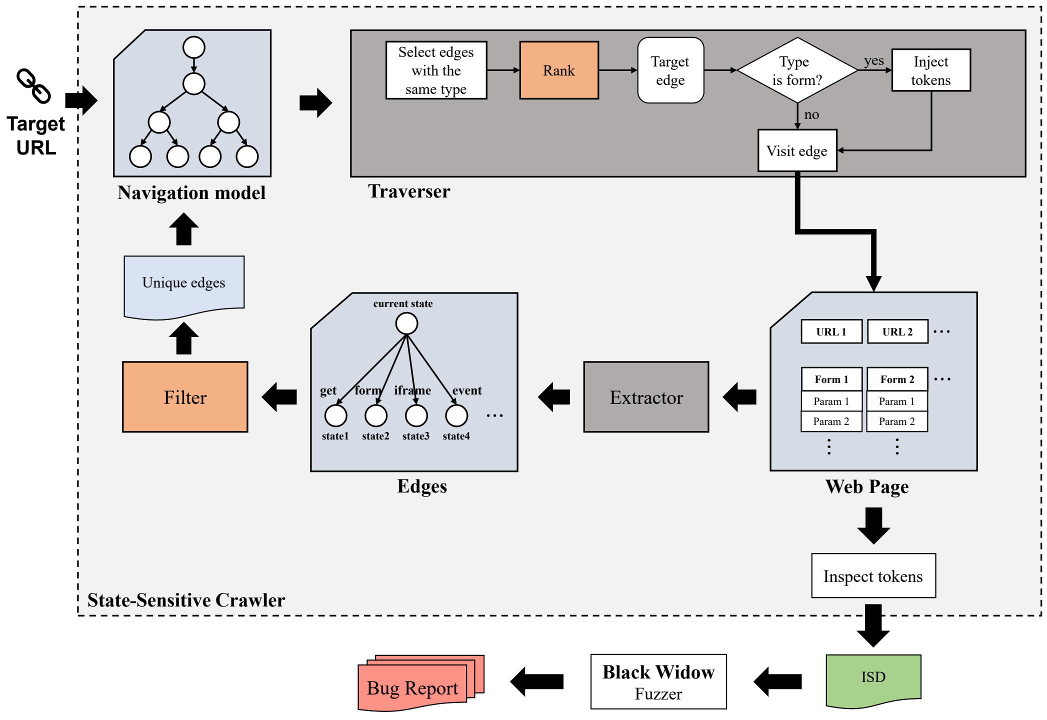

Figure 1. Architecture of SSBW.

3.1. Overview

SSBW is divided into two modules: crawler and fuzzer. The input of the crawler is the URL of the website to be tested. First, an empty node and the URL is constructed to an initial edge, which is the original navigation model graph. Next, we select the only edge of the graph to visit and get the web page corresponding to the initial URL. The web page contains a variety of elements that may find the new state, i.e., GET requests, form submissions, iframes, JavaScript events [8]. The methods for transformation between states are categorized into "get", "form", "iframe", and "event". Thus, an association relationship between the current state (that is, the web page visited by the initial URL) and the elements in the page that may reach new states can be built. Each association relationship is a new edge, the starting node of the edge is the current state, the end node of the edge is the new state that may be reached, and the information required for the transition between the states is recorded. To improve the efficiency of the crawler, it is necessary to delete similar edges (introduced in Section 3.2.4) among all the new edges, in order to avoid repeated visits to the same page or similar forms. Finally, the filtered unique edges are added to the navigation model to create a new navigation model. There are many edges in the new navigation model graph. In order to achieve a balance between the coverage of injection points and the exploration of new states, a proper traversal strategy is required for determining which edge to visit next. We designed different ranking methods for edges with different transformation methods. For edges with a transformation method of "form", their visit order determines the order of fuzzing. Therefore, we believe that their ranking methods play a crucial role in vulnerability detection. The parts of Crawler are detailed in Section 3.2.

During the crawling process, when an edge with the transformation method "form" is visited, tokens with sufficiently high entropy will be injected to map the Inter-state Dependency (ISD), which helps in identifying the mapping relationship between potential source and sink points. The scanner will check if the previously injected token reappears when visiting the new page. Finally, the scanner will submit all the ISD to the fuzzer for vulnerability verification. This is covered in Section 3.3.

3.2. State-Sensitive Crawler

To improve the efficiency of the black-box scanner, the crawler in SSBW crawls the same page and records similar states as little as possible. This relies on the precise identification and filtering of similar states through the crawler. Therefore, our crawler is state-sensitive.

3.2.1. Navigation Model

Inspired by the approach of Eriksson et al. [8], our navigation model is defined in the same way as same as Black Widow. The navigation model is implemented as a directed graph, each node represents the state of the application in terms of unique URLs, and the edges describe the method and necessary information required for the transformation between the states. The transformation methods between the states are divided into "get", "form", "iframe" and "event". Every time we visit a page, we build an association between that page and elements within the page that may reach a new state. Each association represents a new edge, with the starting node being the current state (representing the currently visited page). The ending node of the edge represents a possible new state. The edge itself records the transition information between states and its parent edge.

Visiting states are represented in the navigation model as visiting edges. To properly visit the new state, we need to recursively build the workflow by tracing the parent edge until we find a safe state that represents a GET request. This is based on the HTTP RFC [34], which states that requests of type GET are not considered state-changing and can be safely executed as a starting point.

3.2.2. Traverser

Firstly, we select one of the four transformation methods from the navigation model. For XSS vulnerability detection, the key injection point is the parameters of forms. Thus, we increase the probability of the "form" being selected to 50%, following the same approach as Black Widow. Secondly, we design a unique ranking method for edges with the "form" transformation method, giving priority access to forms with the highest number of input parameters. The edges of the other three transformation methods are ranked in terms of their discovery time through breadth-first traversal. Finally, the edge with the highest priority is selected and visited according to the recursive construction workflow (constructed in navigation model). If the transformation method of the selected edge is "form", an additional token consisting of eight random lowercase characters is injected into the parameters in the form.

Our ranking method for the edges with the "form" transformation method is based on the principle that granting priority visits to forms with more input parameters can potentially detect more inter-state dependencies within a given time period. It is inspired by the observation that forms with a greater number of injection points have a higher probability of containing XSS vulnerabilities. Each input parameter represents a potential injection point. The number of input parameters for each form is recorded by the extractor when it discovers the form submission. We believe that injecting and inspecting tokens constitute the preparation stage of fuzzing. It means that the order in which we visit the forms essentially determines the order in which they are subjected to fuzzing.

3.2.3. Extractor

The function of the extractor is to extract potential new states that can be navigated to from a visited web page, and build edges between the current state and new states. The current state represents the currently visited page. Possible operations on the page include GET requests, form submissions, iframes and JavaScript events. In order to facilitate the detection of XSS vulnerabilities, the GET request here excludes the form submission using "get". The new state is accessed by performing operations within the page.

3.2.4. Filter

For all edges obtained by the extractor, there may be repeated GET requests, as well as edges related to similar form submissions. The filtering algorithm is shown in Algorithm 1. In the navigation model graph, a list of unique URLs, denoted URL, is maintained. This helps filter the construction of repeated GET requests. Alternatively, we have defined two functions (similarUrl and similarForm) to filter similar form submissions. After filtering out similar new edges, we add all the unique edges to the navigation model.

Repeated GET requests: If the URL (only including GET requests) of page C exists on page A, and the URL of page C exists on page B, then only one edge of page C needs to be built to visit it.

Edges of similar form submissions: We will illustrate similar form submissions through the following two examples. First, every product page http://example.com/product.php?id=X contains a form with action http://example.com/product.php?id=X&action=edit for modifying information. We then only need to build a corresponding edge (X represents the number of the product). Second, those two URLs can be http://example.com/product.php/X and http://example.com/product.php/X/edit after URL rewriting.

Function similarUrl: The function is used to determine whether the URLs of the page where the form is located are similar. It determines similarity through the protocol of the URL, domain name, path and parameter list. The parameter list consists of parameter names passed through the "get" method. Except for one item in the path that can be changed, the protocol, domain name, and parameter list should be consistent, e.g., http://example.com/product.php?id=1 and http://example.com/product.php?id=2 are similar, http://example.com/product.php/1/edit and http://example.com/product.php/2/edit are similar.

Function similarForm: This function is used to determine whether the forms are similar. It is mainly judged by the similar action attribute and the same parameter list of the form. The similarity judgment of the action attribute still adopts a function similar Url.

Algorithm 1: Function filter

```
Input:
The list of possible new edges, $E_n$
The list of unique URLs, $URL$
The dictionary of forms that have been visited, $D$
Output:
The list of unique edges, $U_e$
// Every edge contains two nodes and necessary information.
for $e \in E_n$ do
    $similarEdge = False$
    if type(e) = get then
        if $e.node2.url \in URL$ then
            $similarEdge = True$
        end
    end
    if type(e) = form then
        for $url \in D$ do
            if simliarUrl(e.node1.url,url) then
                // $e.value.form$ contains the form information we care
                   about;$D[url]$ is a list.  It represents all the forms
                   in the web page whose URL is equal to $url$.
                if similarForm(e.value.form,D[url]) then
                    $similarEdge = True$
                    break
                end
            end
        end
        if not similarEdge then
            $D[url].append(e.value.form)$
        end
    end
    if not similarEdge then
        $U_e.append(e)$
    end
end
return $U_e$
```

3.3. Fuzzer

The stored XSS vulnerability may not necessarily trigger on the landing page after submitting the payload, such as modifying the payload submitted on the user information page, which can only be triggered on the page where the user information is viewed. Inter-state Dependency (ISD) maps the connection between user input and the states of a web application. Scanners can discover the dependencies between these states by injecting tokens when accessing the form, and checking whether the previously injected tokens appear on the new page. The tokens injected are introduced in the Traverser module. Token inspection is completed in two stages: one is when accessing a new state, and the other is to check all states after completing the crawling process. Finally, the ISD is submitted to the fuzzer for vulnerability verification.

The fuzzer of SSBW adopts Black Widow's fuzzer, which injects a carefully prepared XSS payload dictionary at the injection point and checks all corresponding sink pages for triggering vulnerabilities. By detecting the successful execution of Javascript functions defined in the payload to determine whether the vulnerabilities have been triggered, no false positives will be generated.

4. Evaluation

To evaluate the effectiveness of our approach, we implement it based on Black Widow and compare with Black Widow on a set of eight different web applications. We primarily compare the crawling capabilities and vulnerability detection capabilities.

4.1. Implementation

We have incrementally implemented the filtering and ranking methods based on Black Widow, resulting in the development of two scanners. The first scanner, named S1, applies the filtering method on Black Widow. The second scanner, SSBW, combines both the filtering and ranking methods simultaneously. Black Widow controls mainstream web browsers (chrome) through Python and selenium, which can handle the dynamic characteristics of modern web applications. SSBW and S1 also adopt related technologies. Next, we will discuss the specific experimental subjects and comparative content.

4.2. Experimental Setup

We evaluate our approach on a set of eight different real-world web applications and answer the following questions.

- **RQ1.** How effective is the filtering method in the crawler?
- **RQ2.** Is the ranking method beneficial for vulnerability detection?
- **RQ3.** How effective is SSBW in vulnerability detection?

To make the comparison as fair as possible, we repeated all experiments three times. As in previous studies, we imposed a maximum time limit of eight hours for each scanner to perform a single scan. We take the average of the experimental results for code coverage and the union set of the experimental results for XSS vulnerabilities. Experiments were conducted on a desktop with a 16-core Intel Core i7-10700 CPU 2.90 GHz and 32 GB of RAM.

Web Applications: The eight applications we selected are sourced from previous studies [8,35], including reference applications with known vulnerabilities: WackoPicko (2018), phpBB (2.0.23) and Vanilla (2.0.17.10); and modern production-grade applications: osCommerce (2.3.4.1), Joomla (3.9.6), WordPress (5.1), PrestaShop (1.7.5.1), and ImpressCMS (1.4.4).

Code coverage: To compare the crawling capabilities of the scanner, we calculate the Lines of Code (LoC) executed by web applications during scanning as a metric to measure code coverage. We implement this by enabling Xdebug [36] in each application. During the crawling process, we add functionalities such as version upgrade, user deletion, and password modification to the crawler's blacklist. This ensures the normal operation of the crawler and the correct deployment of the application. Although these pages may also contain vulnerabilities, they should be scanned separately due to their greater impact on crawlers and applications.

Vulnerability detection: Our rank method aims to crawl more attack surfaces in a shorter time, so we also compare the scanner's ability in vulnerability detection. To be consistent with the terminology used in previous works [7,8,19], we also define an XSS vulnerability as any injected JavaScript code that results in execution. Unlike the previous clustering method of XSS vulnerabilities, we believe that independent injection points are a unique vulnerability, while previous research has clustered the injections with the same SQL query into a set of vulnerabilities, since developers only need to change one line of server code to fix the vulnerability. Our method is based on the fact that injection points from different pages may call the same SQL query encapsulated in the program; however, clustering them together cannot accurately evaluate the vulnerability detection ability of scanners. Believing that different injection points are a unique vulnerability is more conducive to a fair comparison of the scanner's vulnerability detection capabilities.

4.3. Analysis of Results

We present our experimental results in this section and analyze them around the three questions raised in the previous section. In addition, we analyze some surprising experimental results in the next section.

4.3.1. Filtering Method (RQ1)

To evaluate the effectiveness of our filtering method, we present the code coverage and detected XSS vulnerabilities on the eight applications by Black Widow and the scanner with the filtering method added to Black Widow (S1).

As shown in Table 1, we count the unique LoC executed by web applications during the operation of Black Widow and S1. Compared with Black Widow, S1 performs better in code coverage on all eight applications. S1 increases the coverage by about 57% in PrestaShop. Due to the small number of lines of code in WackoPicko, the improvement in coverage is less than 1%. Overall, S1 increases the code coverage by approximately 17% in the eight applications.

Table 1. LoC executed by web applications during the operation of SSBW, S1 and Black Widow.

APP/Scanner	SSBW	S1	Black Widow
WackoPicko	1035	1035	1025
Vanilla	15,713	15,337	14,135
osCommerce	17,691	15,764	13,059
Joomla	44,539	44,561	43,947
WordPress	42,820	42,814	41,118
ImpressCMS	21,395	21,353	19,005
phpBB	11,935	11,900	8971
PrestaShop	146,674	147,132	93,556

Through further analysis of the crawling paths, we found that there were duplicate GET requests and similar form submissions in Black Widow's crawling paths. The reason for duplicate GET requests is the presence of similar administrator consoles, where there are many GET request pages that can jump to each other. Black Widow lacks strict filtering methods for this. The reason for similar submissions is that Black Widow does not fully define the filtering method for edges of "form". Consequently, it will perform multiple rounds of crawling on similar forms with the same functionality, thus wasting time. Repeated crawls in these two types of similar states reduces the efficiency of Black Widow.

To further explore the effectiveness of the filtering method in reducing visiting similar states during the scanning process, Figure 2 depicts the temporal evolution of code coverage. At the same time of execution, both SSBW and S1 can almost have higher code coverage than Black Widow. We believe that this is a positive effect of filtering similar states. Sometimes the code coverage of SSBW is lower than that of Black Widow (e.g., during the 2–4 h period of Joomla). We believe this is because SSBW spends the necessary fuzzing time on forms with more input parameters.

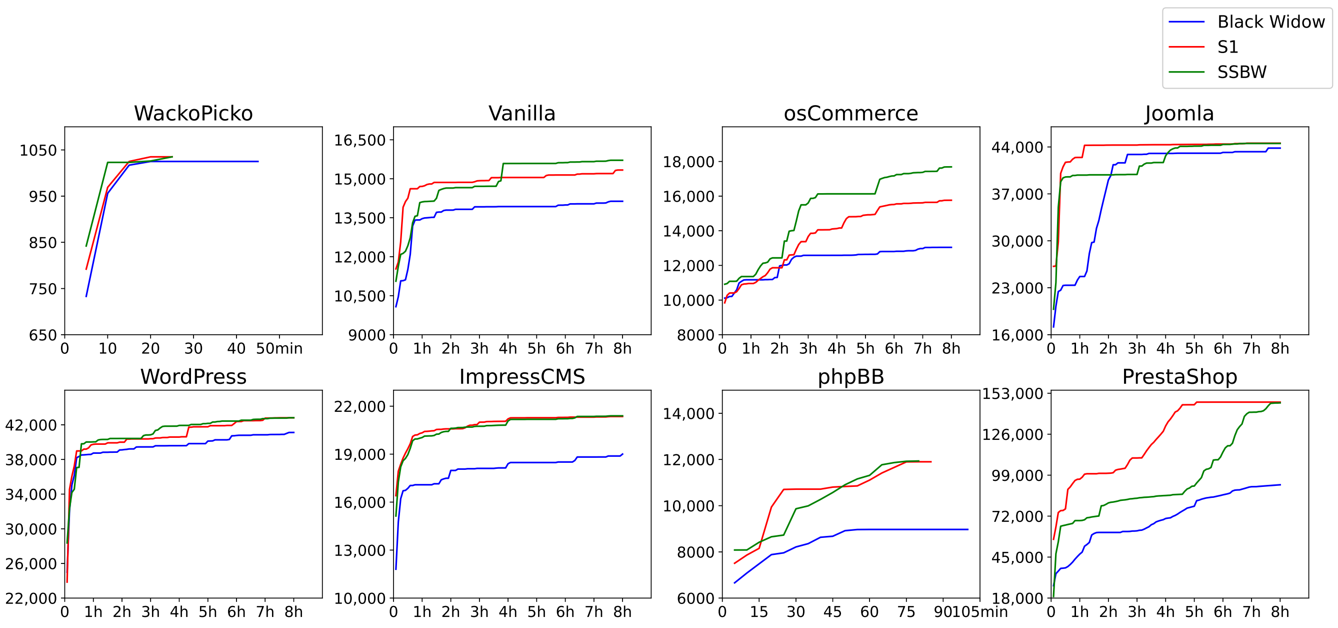

Figure 2. The relationship between LoC and time in eight applications.

Table 2 shows all the XSS vulnerabilities found by Black Widow and S1 on all the applications. In total, S1 finds 81 unique XSS vulnerabilities, and more vulnerabilities are detected in three applications compared to Black Widow. Especially in osCommerce and Vanilla, the total number of XSS vulnerabilities detected by S1 is three times as many as Black Widow. It is more important that S1 does not miss any XSS vulnerabilities compared to Black Widow. This indicates that our filtering method helps improve the crawling capabilities of the scanner and thus enhance vulnerability detection capabilities.

Table 2. Number of XSS vulnerabilities detected by SSBW, S1, Black Widow and the classification of them.

Scanner	SSBW		S1		Black Widow	
Type	R-XSS	S-XSS	R-XSS	S-XSS	R-XSS	S-XSS
WackoPicko	2	2	2	2	2	2
Vanilla	-	11	-	7	-	1
osCommerce	1	68	1	31	-	12
Joomla	-	-	-	-	-	-
WordPress	-	-	-	-	-	-
ImpressCMS	-	3	-	3	-	1
phpBB	-	45	-	33	-	33
PrestaShop	2	-	2	-	2	-

In summary, the experimental results of code coverage and vulnerabilities detection illustrate that our filtering method is effective.

4.3.2. Ranking Method (RQ2)

In this section, we demonstrate that our ranking method is beneficial for vulnerabilities detection in two aspects. On the one hand, our ranking method can increase the number of vulnerabilities detected by the scanner. On the other hand, our ranking method has little negative impact on code coverage.

Table 2 shows all the XSS vulnerabilities found by SSBW and the scanner with filtering method added to Black Widow (S1) on all the applications. SSBW is S1 with the ranking method applied. Compared with S1, SSBW detected 4, 37, 12 more vulnerabilities in Vanilla, osCommerce, and phpBB. This indicates that SSBW has a stronger vulnerability detection capabilities than S1.

The unique LoC executed by web applications during the operation of SSBW and S1 are listed in Table 1. Except in osCommerce, there are subtle code coverage changes in other applications. In the PrestaShop with the largest reduction, the reduction in code coverage was only about 0.31%. This illustrates that the negative impact on code coverage is very little.

Therefore, the experimental results illustrate that the ranking method is beneficial for detecting vulnerabilities.

4.3.3. SSBW in Vulnerability Detection (RQ3)

In this section, the results of the comparison of XSS vulnerabilities detected by various scanners are presented. As shown in Table 2, we list the number of XSS vulnerabilities detected by SSBW and Black Widow on the eight applications. Overall, 134 and 53 vulnerabilities were detected by SSBW and Black Widow, respectively. SSBW detected 81 more vulnerabilities than Black Widow, representing an increase of about 150%.

In Table 3, a deeper analysis is conducted to examine the relationship between the sets of XSS vulnerabilities detected by the three scanners. A, B and C represent the sets of XSS vulnerabilities detected by SSBW, S1 and Black Widow. With the exception of Vanilla and osCommerce, SSBW successfully detected all vulnerabilities that were detected by S1, and S1 detected all the vulnerabilities that were detected by Black Widow on the other six applications. Compared with S1, SSBW detected a total of 44 new vulnerabilities on Vanilla and osCommerce, although it did miss three vulnerabilities. Similarly, compared with Black Widow, SSBW detected additional 58 new vulnerabilities on osCommerce, even though it missed two vulnerabilities. This indicates that SSBW demonstrates superior vulnerability detection capability compared to S1 and Black Widow.

Table 3. Number of XSS vulnerabilities missed by SSBW compared to S1, S1 compared to Black Widow and SSBW compared to Black Widow.

Set	B-A		C-B		C-A	
Type	R-XSS	S-XSS	R-XSS	S-XSS	R-XSS	S-XSS
WackoPicko	-	-	-	-	-	-
Vanilla	-	1	-	-	-	-
osCommerce	-	2	-	-	-	2
Joomla	-	-	-	-	-	-
WordPress	-	-	-	-	-	-
ImpressCMS	-	-	-	-	-	-
phpBB	-	-	-	-	-	-
PrestaShop	-	-	-	-	-	-

On the other hand, Table 1 shows the code coverage of SSBW and Black Widow, respectively. The code coverage of SSBW has increased by an average of about 19%. This indicates that the crawling capability of SSBW is also better than that of Black Widow.

From these two perspectives, SSBW is more effective than Black Widow.

4.4. Case Studies

The results in the Section 4.3 indicate that SSBW has good code coverage and excellent XSS vulnerability detection capabilities. This section provides an in-depth analysis of some surprising experimental results.

Vulnerabilities Missed by SSBW: Throughout the entire experiment, we found that SSBW misses a total of three XSS vulnerabilities detected by S1 or Black Widow on Vanilla and osCommerce. This is caused by our heuristic filtering method. S1 and Black Widow perform fuzzing according to the order in which the form is discovered. Our ranking method modifies the fuzzing order of the form. In the scenario where it is not feasible to perform fuzzing on all forms, this will inevitably lead to some forms not being tested. Therefore, SSBW misses the three vulnerabilities. This is also a limitation of our method.

Crash on Web Applications: We also found that both Black Widow and SSBW may cause server crashes in some applications during the scanning process. The above situation has occurred in Vanilla, osCommerce, phpBB, and ImpressCMS. Unfortunately, we were unable to successfully analyze the more detailed reasons for the server crash. We speculate that the partial payload injected during the fuzzing process has affected the successful parsing of the web page, or that operations during the crawling process have incorrectly configured the website's parsing method. We believe that this is not a flaw only found in SSBW and Black Widow, but rather a common problem with black-box scanners. Therefore, in order to avoid unfair comparisons caused by accidental factors such as website crashes as much as possible, we repeated all experiments three times.

Surprisingly, the scanning process for phpBB always ended within eight hours; however, SSBW can detect XSS vulnerabilities that Black Widow cannot. In theory, the end of the scan represents that the scanner has fuzzed all the forms that have been crawled. Black Widow should be able to detect all vulnerabilities detected by SSBW. We further analyze the execution path of phpBB and the forms that have been fuzzed. The conclusion is that SSBW's ranking method enables some forms to be fuzzed before the server crashes. The lack of ranking method in Black Widow resulted in some forms not being fuzzed before the server crashed.

5. Conclusions and Future Work

In this work, we present a state-sensitive black-box web application scanning method, including a filtering method for excluding similar states and a heuristic ranking method for optimizing the fuzzing order of forms. The filtering method reduces repetitive visits to similar states in the scanning process, especially for the state of HTML forms, which further improves the code coverage of the black-box web scanner within a limited scanning time. The ranking method helps to prioritize the fuzzing order of forms with more injection points, which further improves the efficiency of the black-box web scanner in detecting XSS vulnerabilities. We implemented our method in our black-box scanner SSBW and evaluated it on a set of eight real-world web applications. Compared with Black Widow, the state-of-the-art black-box scanner, SSBW had an average increase of about 19% in code coverage, and the maximum increase was about 57% on PrestaShop. In terms of XSS vulnerabilities detection, SSBW detected a total of 81 more XSS vulnerabilities than Black Widow within eight hours on four applications.

This paper introduces four types of states: GET requests, form submissions, iframes, and Javascript events. It provides a detailed introduction to the similar state filtering method for GET requests and form submissions. The method of judging similar states between iframes and GET requests is also similar. However, due to the dynamic characteristic of Javascript events, determining the similarity of their states without executing events is currently challenging. Exploring methods to avoid repeated visits to similar states of Javascript events is a topic worthy of future study. Additionally, current black box scanners still extensively rely on prepared payload dictionaries for fuzzing. The utilization of artificial intelligence techniques, such as reinforcement learning, is also worth exploring as a way to efficiently generate payloads based on the information detected by black box scanners.

Author Contributions: Conceptualization, T.Z.; Data curation, T.Z., H.H. and K.Z.; Investigation, T.Z.; Methodology, T.Z., K.Z. and J.Z.; Project administration, Y.L.; Resources, T.Z.; Software, T.Z. and J.Z.; Supervision, Y.L.; Validation, T.Z. and H.H.; Writing—original draft, T.Z. and H.H.; Writing—review & editing, T.Z., H.H., Y.L. and K.Z. All authors have read and agreed to the published version of the manuscript.

Funding: The research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data are available on request to the authors.

Acknowledgments: The authors sincerely appreciate Eriksson et al. for their open-source black-box scanner Black Widow, which has helped us a lot.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Khodayari, S.; Pellegrino, G. JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals. In Proceedings of the 30th USENIX Security Symposium, USENIX Security 2021, Virtual, 11–13 August 2021; Bailey, M., Greenstadt, R., Eds.; USENIX Association: Berkeley, CA, USA, 2021; pp. 2525–2542.
2. Trickel, E.; Pagani, F.; Zhu, C.; Dresel, L.; Vigna, G.; Kruegel, C.; Wang, R.; Bao, T.; Shoshitaishvili, Y.; Doupé, A. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities. In Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 22–25 May 2023; IEEE Computer Society: Los Alamitos, CA, USA, 2023; pp. 116–133.
3. Song, X.; Zhang, R.; Dong, Q.; Cui, B. Grey-Box Fuzzing Based on Reinforcement Learning for XSS Vulnerabilities. *Appl. Sci.* **2023**, *13*, 2482. [CrossRef]
4. OWASP. Top 10 Web Application Security Risks. 2021. Available online: https://owasp.org/www-project-top-ten/ (accessed on 30 June 2023).
5. Luo, C.; Li, P.; Meng, W. TChecker: Precise Static Inter-Procedural Analysis for Detecting Taint-Style Vulnerabilities in PHP Applications. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, 7–11 November 2022; Yin, H., Stavrou, A., Cremers, C., Shi, E., Eds.; ACM: New York, NY, USA, 2022; pp. 2175–2188. [CrossRef]
6. van Rooij, O.; Charalambous, M.A.; Kaizer, D.; Papaevripides, M.; Athanasopoulos, E. webFuzz: Grey-Box Fuzzing for Web Applications. In *Lecture Notes in Computer Science, Proceedings of the Computer Security—ESORICS 2021—26th European Symposium on Research in Computer Security, Darmstadt, Germany, 4–8 October 2021*; Bertino, E., Shulman, H., Waidner, M., Eds.; Proceedings, Part I; Springer: Berlin/Heidelberg, Germany, 2021; Volume 12972, pp. 152–172. [CrossRef]
7. Doupé, A.; Cavedon, L.; Kruegel, C.; Vigna, G. Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. In Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, 8–10 August 2012; Kohno, T., Ed.; USENIX Association: Berkeley, CA, USA, 2012; pp. 523–538.
8. Eriksson, B.; Pellegrino, G.; Sabelfeld, A. Black Widow: Blackbox Data-driven Web Scanning. In Proceedings of the 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24–27 May 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 1125–1142. [CrossRef]
9. Gupta, S.; Gupta, B.B. Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art. *Int. J. Syst. Assur. Eng. Manag.* **2017**, *8*, 512–530. [CrossRef]
10. Rodríguez, G.E.; Torres, J.G.; Flores, P.; Benavides, D.E. Cross-site scripting (XSS) attacks and mitigation: A survey. *Comput. Netw.* **2020**, *166*, 106960. [CrossRef]
11. Hannousse, A.; Yahiouche, S.; Nait-Hamoud, M.C. Twenty-two years since revealing cross-site scripting attacks: A systematic mapping and a comprehensive survey. *CoRR* **2022**, arXiv:2205.08425.
12. Jovanovic, N.; Krügel, C.; Kirda, E. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *Computer Society, Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P 2006), Berkeley, CA, USA, 21–24 May 2006*; IEEE: Piscataway, NJ, USA, 2006; pp. 258–263. [CrossRef]
13. Dahse, J.; Holz, T. Simulation of Built-in PHP Features for Precise Static Code Analysis. In Proceedings of the 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, CA, USA, 23–26 February 2014; The Internet Society: Reston, VA, USA, 2014.
14. Backes, M.; Rieck, K.; Skoruppa, M.; Stock, B.; Yamaguchi, F. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In Proceedings of the 2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, 26–28 April 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 334–349. [CrossRef]
15. Balzarotti, D.; Cova, M.; Felmetsger, V.; Jovanovic, N.; Kirda, E.; Kruegel, C.; Vigna, G. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In Proceedings of the 2008 IEEE Symposium on Security and Privacy (S&P 2008), Oakland, CA, USA, 18–21 May 2008; IEEE Computer Society: Los Alamitos, CA, USA , 2008; pp. 387–401. [CrossRef]
16. Pan, J.; Mao, X. Detecting DOM-Sourced Cross-Site Scripting in Browser Extensions. In Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, 17–22 September 2017; IEEE Computer Society: Los Alamitos, CA, USA , 2017; pp. 24–34. [CrossRef]

17. Alhuzali, A.; Gjomemo, R.; Eshete, B.; Venkatakrishnan, V.N. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In Proceedings of the 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, 15–17 August 2018; Enck, W., Felt, A.P., Eds.; USENIX Association: Berkeley, CA, USA, 2018, pp. 377–392.
18. Petukhov, A.; Kozlov, D. *Detecting Security Vulnerabilities in Web Applications Using Dynamic Analysis with Penetration Testing*; Computing Systems Lab, Department of Computer Science, Moscow State University: Moscow, Russia, 2008; pp. 1–120.
19. Pellegrino, G.; Tschürtz, C.; Bodden, E.; Rossow, C. jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications. In *Lecture Notes in Computer Science, Proceedings of the Research in Attacks, Intrusions, and Defenses—18th International Symposium, RAID 2015, Kyoto, Japan, 2–4 November 2015*; Bos, H., Monrose, F., Blanc, G., Eds.; Springer: Berlin/Heidelberg, Germany, 2015; Volume 9404, pp. 295–316. [CrossRef]
20. Riancho, A. w3af—Open Source Web Application Security Scanner. 2007. Available online: https://w3af.org/(accessed on 30 November 2020)
21. LLC, S. Framework—Arachni—Web Application Security Scanner Framework. 2019. Available online: http://www.arachni-scanner.com/(accessed on 30 November 2020)
22. OWASP. Owasp zed Attack Proxy (zap). 2020. https://www.zaproxy.org/(accessed on 30 November 2020)
23. Liu, R.; Lin, Y.; Yang, X.; Ng, S.H.; Divakaran, D.M.; Dong, J.S. Inferring Phishing Intention via Webpage Appearance and Dynamics: A Deep Vision Based Approach. In Proceedings of the 31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, 10–12 August 2022; Butler, K.R.B., Thomas, K., Eds.; USENIX Association: Berkeley, CA, USA, 2022; pp. 1633–1650.
24. Lin, Y.; Liu, R.; Divakaran, D.M.; Ng, J.Y.; Chan, Q.Z.; Lu, Y.; Si, Y.; Zhang, F.; Dong, J.S. Phishpedia: A Hybrid Deep Learning Based Approach to Visually Identify Phishing Webpages. In Proceedings of the 30th USENIX Security Symposium, USENIX Security 2021, Virtual, 11–13 August 2021; Bailey, M., Greenstadt, R., Eds.; USENIX Association: Berkeley, CA, USA, 2021; pp. 3793–3810.
25. Abdelnabi, S.; Krombholz, K.; Fritz, M. VisualPhishNet: Zero-Day Phishing Website Detection by Visual Similarity. In Proceedings of the CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual, 9–13 November 2020; Ligatti, J., Ou, X., Katz, J., Vigna, G., Eds.; ACM: New York, NY, USA, 2020; pp. 1681–1698. [CrossRef]
26. Dalgic, F.; Bozkir, A.; Aydos, M. Phish-IRIS: A New Approach for Vision Based Brand Prediction of Phishing Web Pages via Compact Visual Descriptors. In Proceedings of the 2018 2nd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT), Ankara, Turkey, 19–21 October 2018; pp. 1–8. [CrossRef]
27. Verma, R.M.; Dyer, K. On the Character of Phishing URLs: Accurate and Robust Statistical Learning Classifiers. In Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, CODASPY 2015, San Antonio, TX, USA, 2–4 March 2015; Park, J., Squicciarini, A.C., Eds.; ACM: New York, NY, USA, 2015, pp. 111–122. [CrossRef]
28. Li, Y.; Yang, Z.; Chen, X.; Yuan, H.; Liu, W. A stacking model using URL and HTML features for phishing webpage detection. *Future Gener. Comput. Syst.* **2019**, *94*, 27–39. [CrossRef]
29. Lee, J.; Ye, P.; Liu, R.; Divakaran, D.M.; Chan, M.C. Building Robust Phishing Detection System: An Empirical Analysis. In Proceedings of the NDSS MADWeb 2020, San Diego, CA, USA, 23 February 2020.
30. Drakonakis, K.; Ioannidis, S.; Polakis, J. ReScan: A Middleware Framework for Realistic and Robust Black-box Web Application Scanning. In Proceedings of the 30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, CA, USA, 27 February–3 March 2023; The Internet Society: Reston, VA, USA, 2023.
31. Liang, Y.; Liu, S.; Hu, H. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In Proceedings of the 31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, 10–12 August 2022; Butler, K.R.B., Thomas, K., Eds.; USENIX Association: Berkeley, CA, USA, 2022, pp. 4309–4326.
32. Zhang, L.; Lian, K.; Xiao, H.; Zhang, Z.; Liu, P.; Zhang, Y.; Yang, M.; Duan, H. Exploit the Last Straw That Breaks Android Systems. In Proceedings of the 43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, 22–26 May 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 2230–2247. [CrossRef]
33. Lee, G.; Shim, W.; Lee, B. Constraint-guided Directed Greybox Fuzzing. In Proceedings of the 30th USENIX Security Symposium, USENIX Security 2021, Online, 11–13 August 2021; Bailey, M., Greenstadt, R., Eds.; USENIX Association: Berkeley, CA, USA, 2021, pp. 3559–3576.
34. Fielding, R.; Reschke, J. RFC 7231—Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. 2014. Available online: https://datatracker.ietf.org/doc/html/rfc7231#section-4.2.1 (accessed on 30 June 2023).
35. Su, H.; Xu, L.; Chao, H.; Li, F.; Yuan, Z.; Zhou, J.; Huo, W. A Sanitizer-centric Analysis to Detect Cross-Site Scripting in PHP Programs. In Proceedings of the IEEE 33rd International Symposium on Software Reliability Engineering, ISSRE 2022, Charlotte, NC, USA, 31 October–3 November 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 355–365. [CrossRef]
36. Rethans, D. Xdebug—Debugger and Profiler Tool for PHP. 2021. Available online: https://xdebug.org/ (accessed on 20 April 2023).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.