Zelda: Feedback-driven Closed-box Fuzzing for Identifying Web Application Vulnerabilities

Soyoung Lee
Korea Advanced Institute of Science and Technology
Daejeon, Republic of Korea
soyoungleell@kaist.ac.kr

Sunnyeo Park
Korea Advanced Institute of Science and Technology
Daejeon, Republic of Korea
psnyeo88@kaist.ac.kr

Yonghwi Kwon
University of Maryland
College Park, MD, USA
yongkwon@umd.edu

Sooel Son
Korea Advanced Institute of Science and Technology
Daejeon, Republic of Korea
sl.son@kaist.ac.kr

Abstract

Despite its practical impact, closed-box fuzzing on web applications remains understudied. This paper investigates two fundamental limitations of closed-box web fuzzing: (1) limited input space exploration due to the lack of a feedback mechanism, and (2) ineffective exploitation strategies caused by the shallow vulnerability identification. We propose Zelda, a novel closed-box web fuzzer designed to address these limitations. Specifically, we infer feedback signals from web responses in a closed-box testing environment, thereby deriving a feedback mechanism to guide the fuzzing process. We then coordinate two distinct input generation strategies for path exploration and exploitation, based on the exploration stage, which facilitates both in-page code coverage and vulnerability identification. Our evaluation across 15 real-world applications and nine benchmark sets demonstrates that Zelda's feedback mechanism and strategies are effective in practical web vulnerability discovery. Zelda uncovered 182 vulnerabilities, outperforming six state-of-the-art web fuzzers. In the wild, Zelda further discovered previously unreported vulnerabilities that received 29 CVE assignments.

CCS Concepts

• **Security and privacy → Web application security**.

Keywords

Web Security; Closed-box Fuzzing; Feedback-driven

ACM Reference Format:
Soyoung Lee, Sunnyeo Park, Yonghwi Kwon, and Sooel Son. 2026. Zelda: Feedback-driven Closed-box Fuzzing for Identifying Web Application Vulnerabilities. In *Proceedings of the ACM Web Conference 2026 (WWW '26), April 13–17, 2026, Dubai, United Arab Emirates.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3774904.3792378

Resource Availability:
The source code of this paper has been made publicly available at https://doi.org/10.5281/zenodo.18333961.

WWW '26, Dubai, United Arab Emirates

ACM ISBN 979-8-4007-2307-0/2026/04
https://doi.org/10.1145/3774904.3792378

1 Introduction

Web fuzzing has gained traction for identifying security flaws in server-side web applications. In particular, prior fuzzing research has demonstrated its efficacy in uncovering various web vulnerabilities, including object injection, unrestricted file upload, cross-site scripting (XSS), and SQL injection [13, 17, 18, 59, 60].

Recent advancements in semi-open-box fuzzing [59, 60] show promising potential for web vulnerability discovery. However, unfortunately, their requirements, such as access to the target program and/or interpreter instrumentation for code coverage and the significant overhead caused by the instrumentation (up to sevenfold as reported in §7), limit its practicality in real deployments.

In contrast, closed-box scanners are widely used in production by over 80,000 organizations to assess their services [43, 47, 48, 57]. However, because closed-box web fuzzers lack source code, they have limited feedback for prioritizing promising inputs. It also affects their strategies, where they choose to inject fixed exploits for all input parameters [5, 14, 20, 38], instead of injecting selectively. While recent penetration testing approaches [1, 25] adopt reinforcement learning for guidance, they target a single vulnerability type, limiting applicability to broader vulnerability types.

Our contributions. We observe that state-of-the-art closed-box web fuzzers employ shallow input-generation strategies (see §6.1). They derive inputs from a small set of known exploit patterns. Without a feedback mechanism, they fail to create diverse inputs that penetrate multiple conditions guarding vulnerable code. Moreover, even when execution reaches vulnerable code, their naive exploitation strategies often impede triggering vulnerabilities, which require injecting exploit payloads into specific input parameters.

To overcome these limitations, we design input-generation strategies that address two technical challenges. (1) In a closed-box setting, we must infer and prioritize promising inputs *solely from observable responses*; this requires careful inference techniques that enable diversifying promising inputs that reach vulnerable code without access to source code. (2) We should infer which parameters should receive exploit payloads to trigger vulnerabilities, rather than relying on each tool's fixed heuristics for exploit injection.

In this paper, we propose Zelda, a novel closed-box web fuzzing framework overcoming the challenges above. Zelda orchestrates input generation strategies through a two-stage testing strategy: it first prioritizes exploring more code blocks (i.e., increasing coverage) and then injects exploits into promising input parameters

likely to trigger vulnerabilities. Specifically, we develop (1) a *seed scheduling algorithm* to prioritize promising inputs based on their selection history, and (2) an *input parameter selection algorithm* that leverages differential analysis of web responses for attempted inputs (e.g., response length and the occurrences of the parameters) for effective input parameter mutation. Last, Zelda parallelizes multiple fuzzing processes with seed input sharing, improving throughput.

Our comprehensive evaluation of Zelda on 15 real-world applications and nine benchmark sets shows that Zelda outperformed state-of-the-art web fuzzers (i.e., Burp, Wapiti, BlackWidow, wfuzz, WebFuzz, and Witcher). Zelda identified a total of 182 vulnerabilities, including 65 that all the other six fuzzers failed to detect. Compared with Burp, the next best-performing tool, Zelda identified 91 more vulnerabilities. Notably, Zelda found *10 multi-conditional vulnerabilities* (i.e., vulnerabilities only triggered when multiple conditions are satisfied) that the other fuzzers missed, demonstrating Zelda's ability to address challenging and complex vulnerabilities. Last but not least, Zelda discovered 29 previously unreported vulnerabilities across seven applications, all of which were assigned 29 CVEs.

2 Background: Web Fuzzing

Web penetration testing [42, 55] detects vulnerabilities by injecting known exploits into input parameters and checking the execution of the payload via heuristics. As semi-open-box techniques require source code or an isolated environment, which is often unavailable in production, closed-box scanners are widely used in practice [43].

However, unfortunately, those web scanners choose to *rapidly inject predefined exploits*, which often fail to supply inputs that trigger vulnerabilities [1, 25]. This stems from a design choice toward fast detection under a limited time budget. To address this, previous studies have proposed various techniques for generating diverse inputs via random mutations to explore larger input spaces [17, 59, 60, 71]. Semi-open-box web fuzzing has effectively achieved deeper code coverage [17, 59, 60], but it typically relies on a feedback channel to obtain coverage signals from the target.

In practice, establishing feedback channels poses significant challenges, not only for closed-box fuzzing but also for semi-open-box fuzzing, as it requires a precise analysis of web applications [60] or modifications to the underlying execution environments (e.g., the PHP virtual machine) [59]. The challenges become more significant due to the diverse web programming languages and platforms.

Despite the current limitations, *closed-box security testing* for web services is widely practiced. Large organizations with over 1,000 employees typically manage 100+ web applications built and tested within diverse environments [6]. Therefore, dependence on specific applications or testing environments inevitably becomes a major factor in scalable vulnerability detection, leading them to choose closed-box scanners. In practice, over 80,000 organizations conduct security testing with four popular closed-box web scanners [43, 47, 48, 57]. This widespread use motivates us to develop a new closed-box web fuzzer that overcomes the limitations of state-of-the-art techniques while preserving practicality and scalability.

Terminology. We define key fuzzing terms used in this paper.

- *Web scanner*: A penetration testing tool (e.g., Wapiti [55]) that injects predefined attack exploits (including random strings) to identify web vulnerabilities.

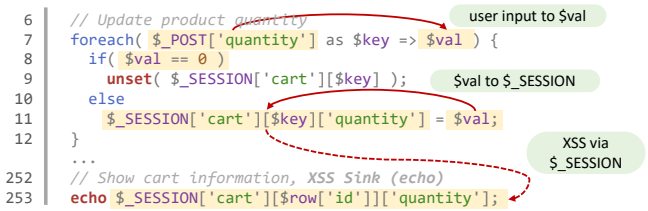


Figure 1: Example of multi-conditional vulnerability.

- *Closed-box web fuzzer*: A fuzzing tool randomly sampling input values to discover new inputs that may trigger vulnerabilities. In this work, we include web scanners under this category as they perform dynamic testing (using exploits and unexpected inputs) without accessing source code or modifying infrastructures.

3 Motivation

Closed-box web fuzzing remains understudied despite its practical importance. Specifically, due to the lack of coverage feedback in the closed-box setting, most prior closed-box web fuzzers [14, 31, 42, 55] focus on penetration testing by injecting predefined exploits [5, 14, 20, 38], prioritizing per-test efficiency rather than systematically exploring promising inputs.

We observe two limitations that closed-box fuzzing struggles to overcome: (1) limited input space exploration that prevents reaching vulnerable code and (2) ineffective exploitation policies that fail to trigger vulnerabilities even when the vulnerable code is reached. In our evaluation (§6.1), these limitations account for 28.05% of false negatives in BlackWidow, 28.22% in Burp, and 12.2% in Wapiti.

First, closed-box fuzzers exhibit limited input space exploration and often miss complex vulnerabilities that require satisfying multiple conditions—*multi-conditional vulnerabilities*. Figure 1 shows a reflected XSS in Shopping Portal [41], which is a multi-conditional vulnerability: (1) negating the condition at Line (Ln) 8 to reach Ln 11; (2) appending an XSS payload to `$val`; (3) invoking `echo` at Ln 253 with valid `$row['id']`. Failing any step prevents the vulnerability from triggering. Semi-open-box fuzzers leverage code coverage feedback to prioritize inputs satisfying such conditions. Unfortunately, code coverage feedback requires access to the target's source code or execution environment (e.g., a PHP virtual machine), which is unavailable for closed-box fuzzers. As a result, without such feedback, closed-box fuzzers often fail to reach vulnerable code due to the inability to satisfy the conditions.

Second, even when a fuzzer reaches vulnerable code, its exploitation strategy often hinders triggering vulnerabilities. For instance, wfuzz uses the *same exploit* for all input parameters, and BlackWidow applies identical payloads across POST parameters.[1] This design may increase throughput, but it fails when a vulnerability requires distinct inputs across parameters. Wapiti stops testing when its injected inputs even partially appear in the response, leading to a possible premature labeling of a reflected XSS; continued testing with different inputs may reveal additional vulnerabilities. Together, limited input space exploration and blunt exploitation strategies cause false negatives, motivating us to devise new solutions.

[1]Consider a web page, `page.php`, accepting two input parameters, `name` and `age`. These fuzzers generate inputs by injecting the same exploit into all parameters, e.g., '`name=EXPLOIT&age=EXPLOIT`'.

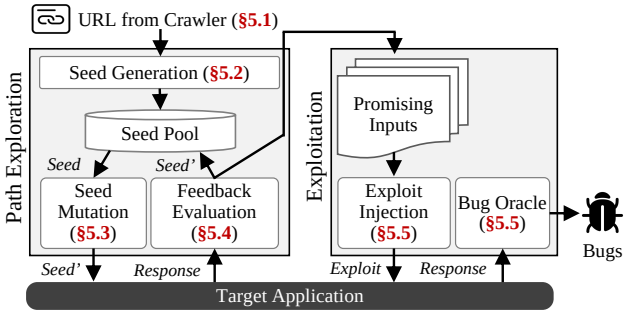


Figure 2: Overview of Zelda.

3.1 Technical Challenges

Reaching multi-conditional vulnerabilities. An effective feedback mechanism is a key missing component in closed-box fuzzers to reach multi-conditional vulnerabilities. To devise effective feedback and guidance mechanisms, it is critical to know (1) which input parameters are relevant to cover vulnerable code and (2) how to obtain a metric that can measure the effectiveness metric of a given input toward the goal. In this context, a fundamental challenge arises from inferring the critical input parameters and their effectiveness without access to the target's internal structure. In other words, We can only use *observable outputs* to infer them.

Effective input exploration and exploitation strategies. To inject functional exploits into proper input parameters (instead of applying the same exploit to all parameters), it is crucial to identify parameters that can affect the internal program state towards successful exploitation. However, closed-box fuzzers do not have access to the internal program state. While feedback mechanisms, such as code coverage, may help, they are vulnerability agnostic (i.e., they aim to cover more code, not vulnerabilities). They are limited in handling vulnerabilities triggered by specific values (e.g., out-of-bounds with an improper value of `$y` in `$x[$y]`).

To this end, we propose the following strategies.

- *Strategy-1. Coverage inference via responses:* As different web responses imply different web application paths covered, we analyze observable web responses (e.g., response status, headers, and body) to determine whether we find a new response. When a mutated input yields a newly observed response, it may indicate increased testing coverage, hinting promising inputs.
- *Strategy-2. Inferring critical input parameters:* To infer critical (or promising) input parameters, we run an iterative process that identifies input parameters inducing changes to the webpage's content, leveraging our feedback mechanism (*Strategy-1*). To further optimize the inference, we employ a frequency-based input parameter weighting method[2] to focus on the input parameters whose names appear frequently in the webpage content.
- *Strategy-3. Separation of input exploration and exploitation:* Unlike typical closed-box fuzzers that directly inject payloads for exploitation, we first identify potentially effective input parameters with a non-exploit payload via *Strategy-2*. Specifically, we mutate and test each parameter individually. We then inject diverse targeted exploit payloads to reveal vulnerabilities.

[2]Intuitively, keywords related to the major functionalities of the page have a higher probability of appearing frequently.

Algorithm 1: Zelda Workflow.

```
1  function ConductFuzzCampaign(start_url)
2      target_set ← Crawler(start_url)
3      for url, param ∈ target_set do
4          cand_vul ← Fuzz(url, param)
5          Exploit(cand_vul)
6  function Fuzz(url, param)
7      S_init ← ParamsToSeed(param)
8      seed_pool ← InitSeedPool(S_init)
9      cand_vul ← []
10     while current_time < timeout do
11         S ← SelectSeed(seed_pool)
12         for i ← 1 to S.energy do
13             param ← SelectParam(param_history)
14             S_mut ← MutateSeed(S, param)
15             resp ← SendRequest(url, param, S_mut)
16             found, param_vul, changes ← Check(resp)
17             if found then
18                 cand_vul.Append(url, param_vul, S_mut)
19             if found or changes or rand() < 0.1 then
20                 seed_pool.AddSeed(S_mut)
21     return cand_vul
22 function Exploit(cand_vul)
23     exploit_dictionary ← GetExploits()
24     for url, param_vul, S_mut ∈ cand_vul do
25         for ex ∈ exploit_dictionary do
26             S_inj ← InjectExploit(S_mut,param_vul,ex)
27             resp ← SendRequest(url,param_vul,S_inj)
28             found ← BugOracle(resp)
29             if found then
30                 ReportBug(url, param_vul, ex)
```

4 Zelda Overview

Zelda is a closed-box web fuzzing tool for detecting web vulnerabilities, including SQL injection, command injection, and XSS. As shown in Figure 2, for each crawled URL, Zelda first conducts the path exploration phase via fuzz testing with our feedback mechanism to assemble promising inputs and input parameters. Zelda then enters the exploitation phase, carefully injecting attack payloads to create functional exploits and confirm vulnerabilities.

Systematized fuzzing. To conduct systematic fuzzing, the following key questions must be addressed [28, 29]: (1) What initial seeds to use (seed generation)? (2) Which seeds to mutate next (seed scheduling)? (3) Which mutated inputs should be selected as new seeds (seed selection)? (4) For each selected seed, how many mutations should be done (seed energy)?

Unlike binary fuzzing techniques [27, 33, 39, 49, 54], which typically operate on a single binary's input channel, web application fuzzing handles multiple web pages, where each page's URL includes multiple input parameters. In addition, in web applications, different types of vulnerabilities can be triggered from the same vulnerable statement depending on the input, further complicating the testing. To this end, we design our algorithm that incorporates key fuzzing components: webpage crawling, seed generation, seed scheduling, seed energy allocation, and bug oracle.

Algorithm. Algorithm 1 details our fuzzing methodology. Given an initial URL of a target website, Zelda crawls reachable URLs and their respective input parameters (Lns 1–2). For each crawled URL, Zelda launches the path exploration phase (Lns 3–4). It creates 10 initial seeds where each seed represents a set of input parameters for the target URL. Then, Zelda assigns default or random values to each input parameter and adds the seed to the seed pool (Lns 7–8).

During the path exploration phase, Zelda creates multiple parallel fuzzing processes[3] over a shared seed pool to improve testing throughput. In each fuzzing process, Zelda selects a seed (Ln 11) based on the *seed score* (§5.2) and chooses *one parameter* via our upper confidence bound (UCB) algorithm. It then mutates the parameter's value with a randomly selected mutation operator (§5.3, Lns 13–14). The fuzzing process then sends a request to the target URL with the mutated input parameters and infers whether it invokes sensitive sink functions, potentially indicating vulnerabilities (Lns 15–16). If successful, the mutated seed is selected as a candidate for the exploitation phase (§5.4, Lns 17–18). The loop repeats until the seed's energy is exhausted or a timeout occurs. Note that if new URLs (not identified during the crawling step) are discovered during testing, they are added to the seed pool as future targets.

For each candidate from path exploration, Zelda conducts the exploitation phase (§5.5, Ln 5) by injecting exploits into the input parameter values of the candidate input (Ln 26). It then checks whether vulnerabilities are triggered by examining the responses with the bug oracle (Ln 28). If vulnerabilities are detected, Zelda reports the corresponding inputs (Lns 29–30).

5 Design

5.1 Crawler

Given an initial URL of a target website, Zelda collects all reachable URLs along with their input parameters. Specifically, we improve Wapiti's crawler [55]. To identify subpages, our crawler inspects URLs from <a>, <link>, and <form> tags, and collects URLs with the same domain name of the initial URL (or relative URLs). The crawling process terminates when no new URLs are identified. We configured the crawler with a maximum crawling depth of 40, following the crawler setting of Wapiti [55]. Note that our crawler uses Selenium [46] to handle dynamically generated URLs by JavaScript.

Input parameter identification. For each crawled URL, Zelda identifies potential input parameters by parsing query strings and relevant HTML tags (e.g., <form>'s inputs and targets). Zelda also detects default values for the identified parameters. The compiled URL list—each containing identified input parameters and their default values—is then passed to the path exploration phase to find effective input values. Path exploration begins after 200 URLs are collected or 90 seconds have elapsed; URLs discovered after that are continuously appended to the crawled list. Crawling and path exploration processes run concurrently for efficiency.

5.2 Seed Generation and Scheduling

Seed. A *seed* is the input unit mutated by Zelda. It consists of a URL, input parameters, cookies, a referrer, a user agent, and a URL-injection string. Each seed is used to compose a fuzzing request with user-provided input values. In the seed, a URL is a fully qualified domain name (FQDN) along with a path. The input parameters are structured as key-value pairs, where a key represents an input parameter name, and the value holds its corresponding value. Note that the seed also includes information regarding the input injection points for testing. Zelda covers a broader range of injection points than other fuzzers (e.g., Wapiti and BlackWidow), including referrer headers, user agent headers, cookies, and URL injection.

Zelda prepares 10 seed inputs and inserts them into the shared seed pool. For each seed, Zelda randomly decides to assign an initial value to each input parameter. If selected and a default value exists, Zelda assigns it. If no default value exists, Zelda creates a 10-character alphanumeric string, consisting of five random letters combined with five random digits, and assigns it to the parameter. Otherwise, no value is assigned.

Seed pool. The seed pool is a priority queue that ranks seeds by the seed score (see Equation 1), shared among parallel processes. Each fuzzing process dequeues the top-ranked seed, mutates it, and requests the target URL. Zelda then checks the response and enqueues the mutated seed if the feedback is positive. Specifically, Zelda checks for error messages and reflections of attempted input—closed-box signals of potential vulnerabilities—facilitating exploration of code blocks. To diversify exploration, Zelda also adds every mutated seed with a 10% probability, regardless of their responses.

Seed score. As shown in Equation 1, we devise the *seed score* to prioritize promising seeds. Specifically, $N(s)$ denotes the number of seed selections (s), and $ln(t)$ denotes the natural logarithm of the total fuzzing attempts. The score is higher for a seed that is less frequently selected for mutation. A new mutated seed that is enqueued due to positive feedback (i.e., induced new webpage content) will be highly ranked, as it has never been selected.

$$Seed\ Score = \sqrt{ln(t) \cdot N(s)^{-1}} \quad (1)$$

5.3 Seed Mutation

Seed energy. For each selected seed, deciding how many mutations to conduct is critical. Intuitively, starting from a small number of mutations, if the mutated seeds are showing promising results, it is wise to focus on those rather than trying to mutate the seed aggressively. On the other hand, if the mutated seeds are not effective, additional mutations will help determine whether they are promising or not. If the mutations are repeatedly ineffective, the seed score will eventually deprioritize them.

$$Seed\ Energy = 2^{N(s)} \quad (2)$$

To this end, we introduce *seed energy* shown in Equation 2, which represents a desirable budget for the seed mutation. It employs an exponential increase based on the seed's selection count ($N(s)$), inspired by AFLFast [7]. Intuitively, the energy remains stable for successful seed mutations, while increasing for unsuccessful mutations, thereby promoting more mutations.

Parameter selection for mutation. Under the seed energy (the mutation budget), Zelda needs to select an input parameter. As there are often multiple input parameters, selecting a promising input parameter is critical. To facilitate the process, we propose a *parameter score* function, leveraging the upper confidence bound (UCB) algorithm [3], which is widely used for reward maximization and fuzzing scheduling [61, 72]. Equation 3 shows the definition.

$$Parameter\ Score = \beta \cdot \sqrt{ln(t) \cdot N(p)^{-1}} + \gamma \cdot Imp(p) + \delta \cdot \Delta Content \quad (3)$$

The first term corresponds to input space exploration, which promotes the selection of less frequently chosen parameters. Here,

[3]The number of fuzzing processes is configurable.

t denotes the total number of parameter selections, and $N(p)$ represents the selections of a specific parameter p. As $N(p)$ increases, this term decreases, encouraging the selection of less frequently chosen parameters in subsequent iterations.

The second term prioritizes parameters likely tied to core functionality. To infer the importance of a parameter, we leverage a hint that if a parameter's name appears frequently on webpage content, it implies higher relevance. For example, on a login page, the keyword "login" might appear frequently, prioritizing mutation of the input parameter named "login." To quantify this impact, we introduce the term *Imp(p)*, representing "importance" and computed by counting the occurrences of the parameter name in the HTTP content in the initial response (before any mutations).

The third term reflects how much a parameter affects the response, favoring parameters inducing significant changes in the HTTP content. We measure impact as the difference in the `Content-Length` header between the initial response and the response obtained after mutation. A larger difference indicates a greater impact on the webpage's content, resulting in a higher score.

Note that these terms are normalized using three hyperparameters: β, γ, and δ. We set $\beta = 1.0$, $\gamma = 0.6$, and $\delta = 0.1$ based on empirical effectiveness for vulnerability discovery (§A.6).

Input mutation. We probabilistically select a mutation operation for the selected parameter's value from seven distinct mutations:

1. *Random generation*: Generate a value that combines both integer numbers and alphabetic characters.
2. *Clear input*: Clear the value of the parameter.
3. *Byte mutation*: Randomly alter each byte.
4. *Special character injection*: Insert special characters that cause syntax errors. (e.g., `/`, `"`, `(`, `'`)
5. *Name prediction*: Generate values based on expected formats associated with the parameter name. (e.g., for the '`email`' parameter, generate `abcd@efg.com`)
6. *Number mutation*: Generate a random number.
7. *String mutation*: Generate a random alphabetic string.

"Random generation" is performed when no input is available for mutation. We assign a lower probability to the "clear input", favoring modification over removal, and split the remaining probability mass equally among the other five operations.

5.4 Determining Inputs for Exploitation

Zelda inspects responses to detect if mutated inputs reach sensitive sinks. For XSS, Zelda checks the presence of any string in the mutated input from the response content. Such reflections suggest the invocation of functions like `echo` or `print`, which can lead to XSS. For SQL/command injection, Zelda checks for 160 error message patterns in the response, indicating database-related failures or command execution issues. Inputs triggering these conditions are marked as exploitation candidates and added to the candidate pool.

Marking injection positions. When Zelda determines that a mutated seed may invoke a sensitive sink, it captures additional information related to this mutated seed. If a substring of an input parameter is found in the response, Zelda marks its position for the exploit injection in the subsequent exploitation phase. For SQL/-command injection, where errors often do not appear directly in responses, Zelda marks all parameters as potential injection points, ensuring that potential vulnerabilities are not overlooked.

5.5 Exploitation

In the exploitation phase, Zelda confirms the presence of vulnerabilities. It constructs input-exploit combinations using promising inputs identified during the exploration phase and an attack dictionary of known exploits to trigger XSS, SQL injection, and command injection. Based on the injection positions marked in (§5.4), Zelda injects each exploit in the dictionary into these positions, generating a new input with the exploit in place. It then requests the target URL and forwards its response to the bug oracle for verification. Zelda reports the combinations that trigger vulnerabilities.

Exploits. For XSS, Zelda utilizes 55 exploits sourced from Wapiti [55] and three additional exploits created by inserting a generated input from the exploration phase. For SQL injection, Zelda uses two strategies: error-based and time-based exploits. Zelda first attempts error-based exploitation. If it does not detect a vulnerability (e.g., failed to analyze error logs in web responses), it proceeds with time-based exploits containing `sleep` functions and escape characters (e.g., single quotes, double quotes, and parentheses). When successfully exploited, the injected `sleep` function will be executed, exhibiting the presence of the vulnerability. For command injection, Zelda uses 13 error-based and 4 time-based exploits. Error-based exploits use special characters and OS shell commands to cause error messages. Zelda also uses time-based exploits containing `sleep` and shell command escapes.

Bug oracle. The bug oracle determines whether an injected exploit was executed. For XSS, we parse web responses with BeautifulSoup [51] and verify that the injected exploits appear in an executable HTML/DOM context. For error-based SQL injection, Zelda matches a predefined dictionary of database error messages against the response. For command injection vulnerabilities, it checks for expected patterns resulting from command execution.

Additionally, Zelda supports a time-based oracle to detect blind SQL/command injection vulnerabilities. When using time-based exploits, Zelda compares response time differences between injected and benign requests; a significant slowdown flags a vulnerability.

We emphasize that Zelda is able to exhaustively try many exploits because it can focus on mutating a small number of promising inputs (not mutating all inputs).

6 Evaluation

We evaluate the effectiveness of Zelda in detecting reflected XSS, SQL injection (SQLi), and command injection (CMDi) by comparing against state-of-the-art fuzzers; we compare the number of detected vulnerabilities, execution time, request count, and coverage.

6.1 Vulnerability Detection Capabilities

6.1.1 Experimental Setup. For hosting the benchmark applications, we used an Ubuntu 22.04 machine with two Intel Xeon CPUs and 128 GB RAM. To run the web fuzzers, we used six Ubuntu 22.04 machines, each with an Intel Core i7-8700 (6 cores) and 32 GB RAM.

Benchmarks. We compiled a set of benchmarks containing vulnerabilities, resulting in nine benchmarks and 15 real-world applications (Table 1; details in §A.2).

Table 1: Evaluation of Zelda and six web fuzzers comparing the number of reported vulnerabilities (TP / FP / FN).

	Application	Zelda	Burp	Wapiti	BW	WebFuzz
Benchmark	DVWA	8 / 0 / 7	10 / 1 / 5	6 / 2 / 9	3 / 0 / 12	0 / 0 / 15
	XVWA	6 / 0 / 2	6 / 0 / 2	5 / 0 / 3	0 / 0 / 8	2 / 0 / 6
	bWAPP	53 / 0 / 15	32 / 0 / 36	34 / 0 / 34	8 / 0 / 60	11 / 0 / 57
	WackoPicko	2 / 0 / 5	3 / 0 / 4	2 / 1 / 5	2 / 0 / 5	1 / 0 / 6
	DVNA	4 / 1 / 4	4 / 1 / 4	0 / 0 / 8	5 / 1 / 3	N/A
	NodeGoat	10 / 0 / 7	5 / 0 / 12	7 / 1 / 10	0 / 0 / 17	N/A
	JuiceShop	0 / 0 / 4	0 / 0 / 4	0 / 0 / 4	0 / 0 / 4	N/A
	WebGoat	12 / 4 / 16	0 / 0 / 28	0 / 0 / 28	13 / 6 / 15	N/A
	MiniBench	9 / 0 / 0	2 / 0 / 7	1 / 0 / 8	1 / 0 / 8	0 / 0 / 9
	TP	104	62	55	32	14
	FP	5	2	4	7	0
	FN	60	102	109	132	93
Real-world	HMS	26 / 0 / 16	7 / 0 / 35	10 / 0 / 32	1 / 3 / 41	0 / 0 / 42
	Doctor	5 / 0 / 1	1 / 0 / 5	0 / 0 / 6	0 / 0 / 6	0 / 0 / 6
	Login Sys	4 / 0 / 2	4 / 0 / 2	5 / 0 / 1	0 / 0 / 6	0 / 0 / 6
	OpenEMR	0 / 0 / 5	0 / 0 / 5	0 / 0 / 5	0 / 0 / 5	N/A
	CE-Phoenix	0 / 0 / 3	0 / 0 / 3	0 / 0 / 3	3 / 0 / 0	0 / 0 / 3
	Joomla	0 / 0 / 1	0 / 0 / 1	0 / 0 / 1	0 / 0 / 1	N/A
	WeBid	10 / 0 / 1	11 / 0 / 0	10 / 0 / 1	0 / 0 / 11	N/A
	Lodel	26 / 0 / 34	3 / 1 / 57	1 / 0 / 59	13 / 0 / 47	1 / 0 / 59
	phpwcms	0 / 0 / 1	0 / 0 / 1	0 / 0 / 1	0 / 0 / 1	0 / 0 / 1
	EspoCRM	0 / 0 / 1	0 / 5 / 1	0 / 0 / 1	0 / 0 / 1	N/A
	Alto CMS	3 / 0 / 1	1 / 0 / 3	0 / 0 / 4	0 / 0 / 4	N/A
	copyparty	1 / 0 / 1	0 / 0 / 2	0 / 0 / 2	0 / 1 / 2	N/A
	Graphite	1 / 0 / 0	0 / 0 / 1	1 / 0 / 0	1 / 0 / 0	N/A
	Emby	1 / 0 / 0	1 / 0 / 0	0 / 0 / 1	0 / 0 / 1	N/A
	Cronicle	1 / 0 / 0	1 / 0 / 0	1 / 0 / 0	0 / 0 / 1	N/A
	TP	78	29	28	18	1
	FP	0	6	0	4	0
	FN	67	116	117	127	117

* wfuzz and Witcher did not report any True Positives (TP).

A. Benchmarks: We select six benchmarks that have been used to evaluate web fuzzers in previous studies [17, 59, 60]: DVWA [56], XVWA [58], bWAPP [32], WackoPicko [11], JuiceShop [35], and WebGoat [37]. Additionally, we include DVNA [2] and NodeGoat [36] to broaden Node.js coverage. To evaluate multi-conditional vulnerabilities, we introduce *MiniBench* (nine applications), each with an XSS vulnerability guarded by conditions. The nine benchmarks contain 164 vulnerabilities: 114 reflected XSS, 43 SQLi, and 7 CMDi.

B. Real-world applications: Among 23 real-world applications used for evaluation from existing fuzzing literature [17, 59, 60], we exclude 12 (10 moved to §A.4 because all state-of-the-art fuzzers found none; two deprecated and infeasible to install) and retain 11. We then add four non-PHP applications (Python, JS, and C#) to diversify platforms. In total, we evaluate 15 applications containing 145 vulnerabilities (86 XSS and 59 SQLi). To our knowledge, our evaluation covers the largest collection of web applications assembled to date for a web fuzzing study [5, 13, 17, 59, 60, 71].

Web fuzzers. We compared the performance of Zelda to four state-of-the-art closed-box web fuzzers (Burp Suite [42] (v2023.10.2.4), Wapiti [55] (v3.1.5), BlackWidow (BW) [14], and wfuzz [31] (v3.1.0)) and two semi-open-box web fuzzers (WebFuzz [60] and Witcher [59]).

Burp and Wapiti are popular web scanners for detecting various vulnerabilities. BW is a scanner designed to improve crawling capability. wfuzz is a closed-box fuzzer that employs brute-force to inject exploits across various injection points. Regarding semi-open-box fuzzers, WebFuzz uses coverage feedback to detect XSS, while Witcher leverages AFL [70] for its fuzzing algorithm.

All fuzzers were configured with their default settings. As WebFuzz requires code coverage, we instrumented all benchmark applications. For WebFuzz and Zelda, we allocated five workers and five processes, respectively, for multi-process fuzzing. Witcher was evaluated using its released artifact [52] with the maximum number of processes. All fuzzers were limited to three hours, except for Witcher, which used its default configuration (four hours for crawling and a 20-minute fuzzing timeout per URL).

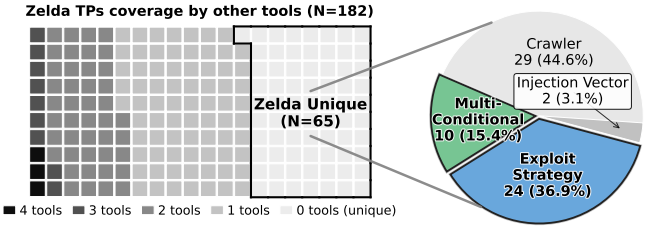


Figure 3: Zelda TP coverage analysis: (left) waffle chart showing coverage by other tools; (right) breakdown of reasons why only Zelda found these unique vulnerabilities.

6.1.2 Effectiveness in Detecting Vulnerabilities. We evaluated all fuzzers on 24 applications (nine benchmarks and 15 real-world applications), covering 309 vulnerabilities. Due to the inherent randomness of fuzz testing, we conducted five fuzzing campaigns and reported the median results for Zelda, WebFuzz, and Witcher, while performing single runs for other fuzzers. Table 1 reports true positives, false positives, and false negatives (TP/FP/FN) per target.

From the nine benchmarks, Zelda identified 104 vulnerabilities, significantly outperforming the other state-of-the-art fuzzers. It reported five FPs and missed 60 vulnerabilities. In the 15 real-world applications, Zelda also outperformed all other tools by discovering 78 vulnerabilities while producing no FPs and 67 FNs.

TPs. Zelda successfully identified 182 vulnerabilities in 18 applications, outperforming the other fuzzers by an average of 78.11%. Specifically, Zelda found 91 (50%), 99 (54.4%), 132 (72.53%), 182 (100%), 167 (91.76%), and 182 (100%) more vulnerabilities than Burp, Wapiti, BW, wfuzz, WebFuzz, and Witcher, respectively. Table 4 (§A.3) breaks down TPs by vulnerability type (i.e., XSS, SQLi, and CMDi). Zelda reported the highest number of TPs for every vulnerability type, detecting 126 XSS, 50 SQLi, and 6 CMDi.

In Figure 3, we summarize the overlap between Zelda's TPs and other tools, and categorize the cases found only by Zelda. We emphasize that Zelda discovered 65 unique vulnerabilities that none of the other fuzzers found (24 from benchmarks and 41 from real-world applications). Our analysis demonstrates that Zelda's feedback-driven strategy—decoupling path exploration and exploitation—significantly improves vulnerability detection effectiveness. Among the 65 unique TPs, 10 cases involved multi-conditional vulnerabilities that other fuzzers failed to trigger due to their fixed inputs. An additional 24 cases (36.92%) stemmed from the other fuzzers' insufficient exploitation strategies (e.g., injecting the same exploit into all input parameters). For instance, Wapiti and Burp stop if no reflection is observed, while Zelda explores diverse inputs to trigger reflection hints that guide exploitation. The remaining 31 unique cases were attributable to Zelda's crawler capability and its support for additional injection vectors (29 and two cases, respectively).

FPs. Zelda exhibited the *second-fewest* false positives, reporting five FPs: one in DVNA and four in WebGoat. All FPs stemmed from the limitations in Zelda's bug oracle. In DVNA, Zelda tried time-based

SQLi and CMDi payloads, and both payloads triggered timeouts; in a closed-box setting, Zelda is unable to distinguish whether the payloads were executed as SQL queries or shell commands, resulting in reporting both SQLi and CMDi vulnerabilities. As the errors were actually caused by SQLi, the CMDi report is an FP. In WebGoat, Zelda identified four FPs where the responses contained non-executable XSS payloads in server error messages.

FNs. Although Zelda outperformed the other tools, it missed 127 vulnerabilities. Note that a single FN may have multiple causes.

The major cause of FNs is limited page discovery (88 FNs; 58.28%). However, improving page discovery is beyond our scope, as we focus on effective input generation. Moreover, Zelda can work on URLs provided by *other crawlers*. To quantify this, we manually provided 88 missed URLs and parameters; Zelda then successfully identified 48 vulnerabilities, while the remaining 40 cases require further improvements in the crawler (e.g., user interactions).

An additional 27 FNs (17.88%) stemmed from closed-box bug oracle limits, as server error logs are inaccessible. Unsupported injection vectors, such as file uploads and custom headers, accounted for 10.6% of the FNs. Missing functional exploits in Zelda's attack dictionary, which require target-specific knowledge for successful exploitation, contributed to 3.97% of the FNs.

We observed five non-deterministic cases where identical inputs do not reliably reach vulnerable paths. For instance, in DVNA's product addition function, the initial request succeeds but a subsequent identical request fails once the item already exists. In such a case, even if Zelda identifies an input triggering a vulnerability during the path exploration phase, replaying the identified input during the exploitation phase does not trigger the vulnerability.

We found five multi-conditional vulnerabilities missed by Zelda. Our manual analysis reveals that Zelda did not generate string values requiring a specific special character at a particular position, showing the limitation of Zelda in producing inputs that meet application-specific conditions. The remaining four vulnerabilities were missed due to limited time resources. When we extended the timeout to 10 seconds per URL, Zelda was able to detect them.

Runtime performance. We compared Zelda against other fuzzers by execution time and the number of requests sent to identify vulnerabilities, excluding crawling time and requests. For WebFuzz, we report the total number of fuzzing and crawling requests since it performs both concurrently, making it infeasible for us to distinguish them. For each tool, we computed the average number of requests and execution time across all benchmark applications.

Table 2 summarizes the results. On average, Zelda required 54,938 requests and took 37 minutes 20 seconds per application. Burp and Wapiti required 40,267 and 33,548 requests, taking 12 minutes 10 seconds and 23 minutes 38 seconds, respectively. Considering that Zelda performs additional exploration and detects more vulnerabilities, it demonstrated competitive performance.

BW sent 774 requests (the second fewest) but its execution time (56 minutes 31 seconds) was longer than Zelda's, as it renders webpages in a browser and sends attack requests one by one. While BW required only 1.41% of Zelda's requests, it detected 27.5% of the vulnerabilities and took 151.4% longer testing time. wfuzz required 26,445 requests and took only 2 minutes 46 seconds via brute-force exploitation without a bug oracle, making it fast but ineffective. WebFuzz sent 965,232 requests and took 2 hours and 49 minutes

Table 2: Runtime performance of Zelda and six web fuzzers (average values reported).

	Zelda	Burp	Wapiti	BW	wfuzz	WebFuzz	Witcher
# of requests	54,938	40,267	33,548	774	26,445	965,232	0
Execution time	37m 20s	12m 10s	23m 38s	56m 31s	2m 46s	168m 52s	174m 35s

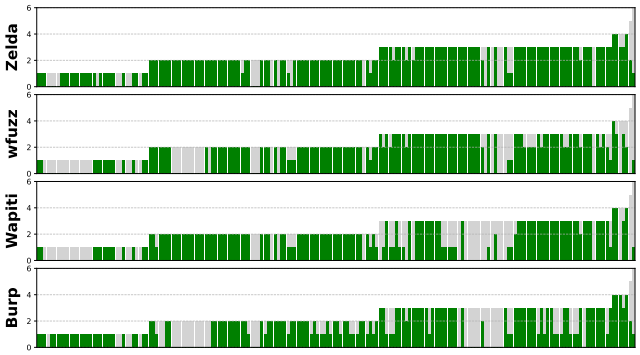


Figure 4: Path coverage of the top four fuzzers that reach in-depth code blocks. Gray shows the depth of code blocks containing vulnerable sinks from the entry points, while green represents the depth of covered blocks. Other fuzzer results are provided in Appendix (Figure 9).

with a 4.85% detection rate—falls significantly behind Zelda. In the case of Witcher, we observed that the fuzzer malfunctioned after crawling, thus reporting zero requests and finding no vulnerabilities. Overall, Zelda achieved a higher detection rate without imposing significant overhead, even with its path exploration phase.

6.1.3 Finding Vulnerabilities in the Wild. We also evaluated the capability of Zelda to find publicly unreported vulnerabilities in the latest versions of 15 real-world applications from our benchmarks and eight PHP applications from open PHP projects [22, 40]. Zelda discovered 29 previously unreported vulnerabilities—eight XSS and 21 SQLi—from seven applications. We reported all vulnerabilities to the corresponding vendors, and *29 CVEs* were assigned.

6.2 In-Depth Analysis with Instrumentation

To evaluate Zelda's ability to reach vulnerable execution paths, we measured code block coverage on instrumented benchmarks. For each vulnerability, we computed the fraction of triggered blocks (§6.2.2) and the cumulative block coverage over time (§A.8).

6.2.1 Experimental Setup. To measure precise block coverage, we instrumented code to log covered basic blocks and inserted canaries at each sink to verify reachability. We instrumented PHP applications in our benchmark— five benchmarks and 11 real-world applications—excluding three real-world applications (Lodel, EspoCRM, and Alto CMS) that failed due to pre-existing syntax issues. In total, we evaluated 13 instrumented applications with 182 vulnerabilities. The setup follows §6.1, except we extend the timeout for closed-box fuzzers and WebFuzz from 3 to 5 hours to offset logging overhead. For Witcher, we applied its default configuration, since it already includes sufficient crawling time (4 hours) and fuzzing timeout (20 minutes per URL).

6.2.2 Path coverage. Zelda detected 107 vulnerabilities, achieving the highest TP rate (58.79%). To analyze these TPs, we examined the path coverage from the entry point of the target page to each sink. Figure 4 illustrates the required block depth to trigger each vulnerability (gray) and the depth of covered blocks (green) across the evaluated fuzzers. Zelda successfully reached the sink functions for 150 vulnerabilities (82.42%), largely outperforming the other fuzzers (mean 54.12%). Overall, Figure 4 shows that Zelda covered the most extensive code blocks required to trigger vulnerabilities.

6.3 Ablation Study

We conducted an ablation study to understand the impact of each component of Zelda in identifying vulnerabilities. We prepared four different versions of Zelda: (1) the full implementation of Zelda; (2) $Zelda_{w/o\_paramselect}$, which replaces the parameter selection algorithm with random selection; (3) $Zelda_{random}$, which replaces both the parameter and seed selection algorithms with random selections; and (4) $Zelda_{w/o\_pathexplore}$, which omits the path exploration phase and only performs the exploitation phase by injecting predefined exploits into each input parameter. For the ablation study, we used bWAPP, DVWA, XVWA, and MiniBench, with a three-hour time limit. We conducted five campaigns per Zelda variant and compared the median number of detected vulnerabilities.

Zelda outperformed the other versions, identifying 76 vulnerabilities and achieving the highest number of TPs. It underscores the effectiveness of each technical component, which collectively contributes to the overall effectiveness of Zelda. $Zelda_{w/o\_paramselect}$ and $Zelda_{random}$ detected 72 and 69 vulnerabilities, respectively. On the other hand, $Zelda_{w/o\_pathexplore}$ shows a significant performance decrease, detecting only 27 vulnerabilities. This performance gap underscores the crucial role of the path exploration phase. The shortfall in $Zelda_{w/o\_pathexplore}$ is primarily due to its inability to generate appropriate inputs for multiple input parameters. Note that Zelda filters out input parameters that are not effective in triggering vulnerabilities during its fuzzing phase. This allows Zelda to allocate more time and resources to the promising parameters when attempting attack exploits. In contrast, $Zelda_{w/o\_pathexplore}$ fails to prioritize these promising parameters since limited fuzzing time is spent trying attack exploits across all input parameters rather than focusing on the most promising ones.

7 Discussion

Coverage feedback. Recent studies have demonstrated the effectiveness of code coverage in semi-open-box fuzzing [17, 59]. However, obtaining code coverage often suffers from incompatibility or excessive overhead on real-world applications. We investigated the overheads of state-of-the-art code coverage tools by measuring the loading time of each web application index page across three popular PHP code coverage libraries, i.e., PCOV [65], Xdebug [50], and WebFuzz [60], with and without the code coverage computation.

When measuring the differences in page loading times with and without each code coverage library, the average overheads were 76.8% (PCOV), 72.8% (Xdebug), and 226.49% (WebFuzz instrumentation); see Appendix (§A.7) for details. The maximum overheads reached 267.4%, 181.5%, and 748.75% in WordPress, respectively. Moreover, WebFuzz instrumentation is incompatible with OpenEMR and EspoCRM. We also note that different server-side programming languages require different code coverage tools. These tools often introduce side effects due to instrumented code, thereby impeding functional services or fuzz testing [60, 62, 69].

8 Related Work

The identification of web vulnerabilities has been a long-standing research area. Static approaches have entailed conducting a series of data flow analyses on source code, including string analysis [53, 64], capturing data flows [19, 34, 53, 67], simulating built-in features [9, 10], and leveraging code property graphs [4, 66]. While these methods offer high coverage on suspicious flows, they suffer from false positives due to the dynamic nature of web applications. Dynamic approaches, on the other hand, have involved instrumenting the browser to track data flow at runtime [8, 21, 26, 30, 63].

Recently, a new research trend focusing on semi-open-box web fuzzing has emerged. WebFuzz [60] is the first semi-open-box web fuzzer designed to identify XSS vulnerabilities. It performs mutation using a predefined grammar based on coverage collected by instrumenting target PHP applications. Witcher [59] is a semi-open-box fuzzer designed to detect SQLi and CMDi vulnerabilities, employing coverage-guided fuzzing through a revised PHP interpreter. Atropos [17] is another semi-open-box fuzzer for server-side vulnerability detection, including remote code execution, file inclusion, and PHP object injection vulnerabilities. It leverages algorithms used in AFL++ [16] and collects web application feedback by revising a PHP interpreter. In contrast, there have been a few studies on closed-box web fuzzing. Most prior efforts in closed-box fuzzing have focused on penetration testing [12, 14, 15, 20, 68]. These approaches limit the exploration of extensive input spaces in web applications by overlooking the prioritization of promising inputs.

9 Conclusion

We propose Zelda, a novel closed-box fuzzing framework that overcomes the limitations of closed-box fuzzers. Zelda employs an input parameter selection algorithm to systematically select promising input parameter values for further mutation. For this, we design a feedback method that conducts differential analysis of web responses to identify promising input parameters. Zelda also focuses on exploiting promising inputs by separating the exploration and exploitation phases, uncovering more vulnerabilities. Zelda has successfully identified 182 vulnerabilities, outperforming the state-of-the-art web fuzzers, and discovered 29 previously unreported vulnerabilities. These findings demonstrate Zelda's effectiveness in web vulnerability identification and highlight its potential in advancing closed-box web fuzzing.

Acknowledgments

We thank the anonymous reviewers for their concrete feedback. This work was supported by (1) the National Research Foundation of Korea (NRF) grant (No. RS-2023-NR076965), (2) the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant (No. RS-2020-II200153, Penetration Security Testing of ML Model Vulnerabilities and Defense) funded by the Korea government (MSIT), and (3) the NSF CAREER Award 2427783.

References

[1] Salim Al Wahaibi, Myles Foley, and Sergio Maffeis. 2023. SQIRL: Grey-Box Detection of SQL Injection Vulnerabilities Using Reinforcement Learning. In *Proceedings of the USENIX Security Symposium*.

[2] Appsecco. 2024. DVNA. https://github.com/appsecco/dvna.

[3] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning* 47 (2002), 235–256.

[4] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and flexible discovery of php application vulnerabilities. In *Proceedings of the IEEE European Symposium on Security and Privacy*.

[5] Enrico Bazzoli, Claudio Criscione, Federico Maggi, and Stefano Zanero. 2016. XSS PEEKER: Dissecting the XSS exploitation techniques and fuzzing mechanisms of blackbox web application scanners. In *Proceedings of the Information Security Conference and Privacy*.

[6] BetterCloud. 2023. 2023 State of SaaSOps. https://pages.bettercloud.com/rs/719-KZY-706/images/2023-StateofSaaSOps-report-final.pdf.

[7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the ACM Conference on Computer and Communications Security*.

[8] Quan Chen and Alexandros Kapravelos. 2018. Mystique: Uncovering information leakage from browser extensions. In *Proceedings of the ACM Conference on Computer and Communications Security*.

[9] Johannes Dahse and Thorsten Holz. 2014. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *Proceedings of the Network and Distributed System Security Symposium*.

[10] Johannes Dahse and Thorsten Holz. 2014. Static Detection of Second-Order Vulnerabilities in Web Applications. In *Proceedings of the USENIX Security Symposium*.

[11] Adam Doupe. 2024. WackoPicko. https://github.com/adamdoupe/WackoPicko.

[12] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2012. Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. In *Proceedings of the USENIX Security Symposium*.

[13] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. 2014. KameleonFuzz: evolutionary fuzzing for black-box XSS detection. In *Proceedings of the ACM Conference on Data and Application Security and Privacy*.

[14] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. 2021. Black Widow: Blackbox data-driven web scanning. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[15] Damiano Esposito, Marc Rennhard, Lukas Ruf, and Arno Wagner. 2018. Exploiting the potential of web application vulnerability scanning. In *Proceedings of the International Conference on Internet Monitoring and Protection*.

[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: combining incremental steps of fuzzing research. In *USENIX Workshop on Offensive Technologies*.

[17] Emre Güler, Sergej Schumilo, Moritz Schloegel, Nils Bars, Philipp Görz, Xinyi Xu, Cemal Kaygusuz, and Thorsten Holz. 2024. Atropos: Effective Fuzzing of Web Applications for Server-Side Vulnerabilities. In *Proceedings of the USENIX Security Symposium*.

[18] Jin Huang, Junjie Zhang, Jialun Liu, Chuang Li, and Rui Dai. 2021. Ufuzzer: Lightweight detection of php-based unrestricted file upload vulnerabilities via static-fuzzing co-analysis. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses*.

[19] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. 2004. Securing web application code by static analysis and runtime protection. In *Proceedings of the Web Conference*.

[20] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. 2006. Secubat: a web vulnerability scanner. In *Proceedings of the Web Conference*.

[21] Zifeng Kang, Song Li, and Yinzhi Cao. 2022. Probe the Proto: Measuring Client-Side Prototype Pollution Vulnerabilities of One Million Real-world Websites.. In *Proceedings of the Network and Distributed System Security Symposium*.

[22] kashipara Group. 2024. Kashipara. https://www.kashipara.com.

[23] Anuj Kumar. 2024. Hospital Management System In PHP. https://phpgurukul.com/hospital-management-system-in-php.

[24] Anuj Kumar. 2024. User Registration & Login and User Management System With admin panel. https://phpgurukul.com/user-registration-login-and-user-management-system-with-admin-panel.

[25] Soyoung Lee, Seongil Wi, and Sooel Son. 2022. Link: Black-Box Detection of Cross-Site Scripting Vulnerabilities Using Reinforcement Learning. In *Proceedings of the Web Conference*.

[26] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: large-scale detection of DOM-based XSS. In *Proceedings of the ACM Conference on Computer and Communications Security*.

[27] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the International Symposium on Foundations of Software Engineering*.

[28] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218.

[29] V. M. Manes, H. Han, C. Han, S. Cha, M. Egele, E. J. Schwartz, and M. Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* (2021), 2312–2331.

[30] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out DOMsday: Towards detecting and preventing dom cross-site scripting. In *Proceedings of the Network and Distributed System Security Symposium*.

[31] Xavi Mendez. 2024. wfuzz. https://github.com/xmendez/wfuzz.

[32] MME. 2024. bWAPP. http://www.itsecgames.com.

[33] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. 2020. Binary-level directed fuzzing for use-after-free vulnerabilities. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses*.

[34] Paulo Jorge Costa Nunes, José Fonseca, and Marco Vieira. 2015. phpSAFE: A security analysis tool for OOP web application plugins. In *Proceedings of the International Conference on Dependable Systems and Networks*. IEEE.

[35] OWASP. 2024. Juice Shop. https://github.com/juice-shop/juice-shop.

[36] OWASP. 2024. NodeGoat. https://github.com/OWASP/NodeGoat.

[37] OWASP. 2024. WebGoat. https://github.com/WebGoat/WebGoat.

[38] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. 2015. jäk: Using dynamic analysis to crawl and test modern web applications. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses*.

[39] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2016. Model-based whitebox fuzzing for program binaries. In *Proceedings of the International Conference on Automated Software Engineering*.

[40] PHPGurukul. 2024. PHPGurukul. https://phpgurukul.com.

[41] PHPGurukul. 2024. Shopping Portal. https://phpgurukul.com/shopping-portal-free-download.

[42] PortSwigger. 2024. Burp Suite - Cybersecurity Software from PortSwigger. https://portswigger.net/burp.

[43] PortSwigger. 2024. Burp Suite Customers. https://portswigger.net/customers.

[44] Projectworlds. 2024. Online Doctor Appointment Booking System PHP and Mysql. https://projectworlds.in/free-projects/php-projects/online-doctor-appointment-booking-system-php-and-mysql.

[45] PyPI. 2024. requests. https://pypi.org/project/requests.

[46] PyPI. 2024. selenium. https://pypi.org/project/selenium.

[47] Qualys. 2024. Customers. https://www.qualys.com/customers.

[48] Rapid7. 2024. Customers. https://www.rapid7.com/customers.

[49] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium*.

[50] Derick Rethans. 2024. Xdebug. https://xdebug.org.

[51] Leonard Richardson. 2024. Beautiful Soup. https://www.crummy.com/software/BeautifulSoup.

[52] SEFCOM. 2023. Experiments from the Witcher paper. https://github.com/sefcom/Witcher-experiment.

[53] Antonin Steinhauser and François Gauthier. 2016. JSPChecker: Static detection of context-sensitive cross-site scripting flaws in legacy web applications. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*.

[54] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium*.

[55] Nicolas Surribas. 2024. Wapiti. https://wapiti.sourceforge.io.

[56] DVWA team. 2024. DVWA. https://github.com/digininja/DVWA.

[57] Tenable. 2024. Customers. https://www.tenable.com/customers.

[58] Sanoop Thomas. 2024. XVWA. https://github.com/s4n7h0/xvwa.

[59] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. 2023. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[60] Orpheas van Rooij, Marcos Antonios Charalambous, Demetris Kaizer, Michalis Papaevripides, and Elias Athanasopoulos. 2021. webfuzz: Grey-box fuzzing for web applications. In *Proceedings of the European Symposium on Research in Computer Security*.

[61] Jinghan Wang, Chengyu Song, and Heng Yin. 2021. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In *Proceedings of the Network and Distributed System Security Symposium*.

[62] Meng Wang, Chijung Jung, Ali Ahad, and Yonghwi Kwon. 2021. Spinner: Automated Dynamic Command Subsystem Perturbation. In *Proceedings of the ACM Conference on Computer and Communications Security*.

[63] Ran Wang, Guangquan Xu, Xianjiao Zeng, Xiaohong Li, and Zhiyong Feng. 2018. TT-XSS: A novel taint tracking based dynamic detection framework for DOM Cross-Site Scripting. *J. Parallel and Distrib. Comput.* 118 (2018), 100–106.

[64] Gary Wassermann and Zhendong Su. 2008. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the International Conference on Software Engineering*.

[65] Joe Watkins. 2024. PCOV - CodeCoverage compatible driver for PHP. https://github.com/krakjoe/pcov.
[66] Seongil Wi, Sijae Woo, Joyce Jiyoung Whang, and Sooel Son. 2022. HiddenCPG: large-scale vulnerable clone detection using subgraph isomorphism of code property graphs. In *Proceedings of the Web Conference*.
[67] Yichen Xie and Alex Aiken. 2006. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the USENIX Security Symposium*.
[68] Zijing Yin, Yiwen Xu, Fuchen Ma, Haohao Gao, Lei Qiao, and Yu Jiang. 2023. Scanner++: Enhanced vulnerability detection of web applications with attack intent synchronization. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–30.
[69] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. 2007. JavaScript instrumentation for browser security. *ACM SIGPLAN Notices* (2007).
[70] Michal Zalewski. 2024. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl.
[71] Jiazhen Zhao, Yuliang Lu, Kailong Zhu, Zehan Chen, and Hui Huang. 2022. Cefuzz: An directed fuzzing framework for php rce vulnerability. *Electronics*.
[72] Yiru Zhao, Xiaoke Wang, Lei Zhao, Yueqiang Cheng, and Heng Yin. 2022. Alphuzz: Monte carlo search on seed-mutation tree for coverage-guided fuzzing. In *Proceedings of the Annual Computer Security Applications Conference*.

A Appendix

A.1 Implementation

We implemented Zelda with 10.0K lines of code (LoC) in Python. For the baseline crawler, we used Wapiti [55] and revised it to support Markdown document parsing and browser rendering. To establish sessions with web applications under fuzz testing, we leveraged the Python library *requests* [45] for HTTP communication and Selenium [46] for browser rendering. To support open science, we release Zelda at `https://github.com/WSP-LAB/Zelda`.

Table 3: Benchmark web applications.

	Application	Lang	LoC	GitHub ★	# of Vulnerabilities		
					XSS	SQLi	CMDi
Benchmark	DVWA	PHP	9.74k	8.8k	11	3	1
	XVWA	PHP	27.51k	1.7k	4	3	1
	bWAPP	PHP	31.87k	-	49	16	3
	WackoPicko	PHP	3.75k	317	4	2	1
	DVNA	JS	11.36k	648	5	2	1
	NodeGoat	JS	40.08k	1.8k	14	3	0
	JuiceShop 8.1.0	JS	122.99k	9.1k	2	2	0
	WebGoat 8.1	Java	84.40k	6.2k	16	12	0
	MiniBench	PHP	533	-	9	0	0
Real-world	HMS [23]	PHP	140.38k	-	3	39	0
	Doctor [44]	PHP	53.90k	-	0	6	0
	Login Sys [24]	PHP	17.24k	-	0	6	0
	OpenEMR 5.0.1.7	PHP	2.38M	2.5k	0	5	0
	CE-Phoenix 1.0.8.20	PHP	50.14k	19	3	0	0
	Joomla 3.8.8	PHP	547.21k	4.6k	1	0	0
	WeBid 1.2.2	PHP	86.70k	115	10	1	0
	Lodel 1.0.5	PHP	290.63k	46	59	1	0
	phpwcms 1.9.26	PHP	163.89k	90	0	1	0
	EspoCRM 5.6.4	PHP	805.66k	1.4k	1	0	0
	Alto CMS 1.1.31	PHP	426.01k	108	4	0	0
	copyparty 1.8.4	Python	54.76k	532	2	0	0
	Graphite 1.1.8	JS/Python	82.56k	5.9k	1	0	0
	Emby 4.5.4.0	C#	1.15M	4k	1	0	0
	Cronicle 0.9.46	JS	21.85k	3.6k	1	0	0

A.2 Benchmark Statistics

Table 3 shows statistics of the 24 applications on which we evaluated Zelda in §6. For each application, the table shows its programming languages, LoC, the number of stars on its GitHub repository, and the number of known vulnerabilities.

Benchmark. We selected six benchmarks used in previous studies [17, 59, 60] and included two benchmarks in Node.js. These benchmarks are implemented in various languages, including PHP, JS, and Java, with LoCs ranging from 3.75K to 122.99K. Additionally, we developed MiniBench, including nine multi-conditional vulnerabilities. The nine benchmarks collectively contain 164 vulnerabilities, including 114 reflected XSS, 43 SQLi, and 7 CMDi.

Real-world. We selected 11 PHP applications used in evaluating state-of-the-art web fuzzing approaches [17, 59, 60]. Additionally, we selected four real-world applications that use other web development languages, such as Python, JS, and C#. Each application has at least one known vulnerability, totaling 145 vulnerabilities (86 XSS and 59 SQLi).

A.3 TPs for Each Vulnerability Type

In §6.1, we evaluated Zelda and six existing fuzzers using 24 benchmark applications. Table 4 shows the number of detected vulnerabilities for each vulnerability type, including XSS, SQLi, and CMDi. Among the tools, Zelda, Burp, and Wapiti handle all three vulnerability types, while BlackWidow and WebFuzz support only XSS, wfuzz supports XSS and SQLi, and Witcher supports SQLi and CMDi vulnerabilities. We denote unsupported vulnerability types as *N/S*. Since WebFuzz is a semi-open-box tool requiring target application instrumentation, we excluded eight non-PHP applications and five applications where instrumentation failed or runtime errors occurred. Although Witcher is a semi-open-box tool supporting multiple languages, including PHP, Python, Java, Node.js, and Ruby, it lacks support for C# and specific Node.js versions. Consequently, we excluded eight non-PHP applications and four PHP applications where runtime errors occurred. Applications excluded from the evaluation are denoted as *N/A*.

Table 4: The number of vulnerabilities reported by Zelda and six other web fuzzers (XSS / SQLi / CMDi).

Application	Zelda	Burp	Wapiti	BW	wfuzz	WebFuzz	Witcher
DVWA	4 / 3 / 1	7 / 2 / 1	3 / 2 / 1	3 / 0 / 0	-	-	-
XVWA	3 / 2 / 1	3 / 2 / 1	4 / 0 / 1	-	-	2 / 0 / 0	-
bWAPP	40 / 10 / 3	25 / 6 / 1	23 / 9 / 2	8 / 0 / 0	-	11 / 0 / 0	-
WackoPicko	1 / 1 / 0	2 / 1 / 0	1 / 0 / 1	2 / 0 / 0	-	1 / 0 / 0	-
DVNA	2 / 1 / 1	2 / 1 / 1	-	5 / 0 / 0	-	N/A	N/A
NodeGoat	10 / 0 / 0	5 / 0 / 0	7 / 0 / 0	-	-	N/A	N/A
JuiceShop	-	-	-	-	-	N/A	N/A
WebGoat	12 / 0 / 0	-	-	13 / 0 / 0	-	N/A	N/A
MiniBench	9 / 0 / 0	2 / 0 / 0	1 / 0 / 0	1 / 0 / 0	-	-	-
XSS	81	46	39	32	0	14	N/S
SQLi	17	12	11	N/S	0	N/S	0
CMDi	6	4	5	N/S	N/S	N/S	0
HMS	3 / 23 / 0	1 / 6 / 0	0 / 10 / 0	1 / 0 / 0	-	-	-
Doctor	0 / 5 / 0	0 / 1 / 0	-	-	-	-	-
Login Sys	0 / 4 / 0	0 / 4 / 0	0 / 5 / 0	-	-	-	-
OpenEMR	-	-	-	-	-	N/A	N/A
CE-Phoenix	-	-	-	3 / 0 / 0	-	-	-
Joomla	-	-	-	-	-	N/A	-
WeBid	10 / 0 / 0	10 / 1 / 0	10 / 0 / 0	-	-	N/A	-
Lodel	25 / 1 / 0	3 / 0 / 0	1 / 0 / 0	13 / 0 / 0	-	1 / 0 / 0	N/A
phpwcms	-	-	-	-	-	-	-
EspoCRM	-	-	-	-	-	N/A	N/A
Alto CMS	3 / 0 / 0	1 / 0 / 0	-	-	-	N/A	N/A
copyparty	1 / 0 / 0	-	-	-	-	N/A	N/A
Graphite	1 / 0 / 0	-	1 / 0 / 0	1 / 0 / 0	-	N/A	N/A
Emby	1 / 0 / 0	1 / 0 / 0	-	-	-	N/A	N/A
Cronicle	1 / 0 / 0	1 / 0 / 0	1 / 0 / 0	-	-	N/A	N/A
XSS	45	17	13	18	0	1	N/S
SQLi	33	12	15	N/S	0	N/S	0
CMDi	0	0	0	N/S	N/S	N/S	0

A dash (–) represents 0 / 0 / 0
N/A: Not available for the evaluation, N/S: Non-supporting vulnerability types

Zelda reported the highest number of TPs for every vulnerability type. Specifically, Zelda detected 126 XSS, 50 SQLi, and 6 CMDi vulnerabilities. On average, Zelda identified 90 (71.43%) more XSS, 38 (75%) more SQLi, and 3 (50%) more CMDi vulnerabilities than the other tools.

Table 5: Evaluation of Zelda on 10 real-world applications.

Application	Lang	LoC	GitHub ★	Zelda		
				TP	Requests	Time (s)
Atropim 1.9.22	PHP	32.45K	123	0	1,083	2m 36s
Down52 2.1.3	PHP	14.09K	2.4K	0	0	0
nextCloud 29.0.0	PHP	941.34K	24.8K	0	19,703	25m 43s
phpBB 3.3.3	PHP	812.63K	1.8K	0	25,675	15m 30s
FlaskBB 2.0.0	Python	33.84K	2.5K	0	5,491	4m 42s
Invoice Ninja 3.6.1	PHP	1.89M	7.5K	0	3,083	21m 20s
Maxsite 108	PHP	101.49K	142	0	0	0
Iubenda 3.3.2	PHP	16.74K	0.1M+*	0	335	27
osCommerce 2.3.4	PHP	67.18K	279	0	1,011	27
Wordpress 5.7.1	PHP	748.15K	18.4K	0	102,780	2h 58m 15s

*Active installations.

A.4 Evaluation on Another Benchmark Set

In addition to the 15 real-world applications with known vulnerabilities, we evaluated Zelda on 10 other applications used in prior web fuzzing studies [17, 59]. These applications have no known vulnerabilities, and neither of the state-of-the-art fuzzers (i.e., Atropos [17] nor Witcher [59]) identified any vulnerabilities.

Table 5 provides details of these applications and summarizes the results. The first four columns present information about each application, including its programming language, LoC, and the number of GitHub stars. The last three columns show the performance of Zelda, including the number of detected vulnerabilities, the number of requests sent to identify vulnerabilities, and execution time. Similar to Atropos and Witcher, Zelda did not detect any vulnerabilities in these 10 applications.

A.5 Case Studies

Figure 5: Example of MiniBench.

MiniBench. Figure 5 shows an example from the MiniBench benchmark. It has a reflected XSS vulnerability involving simple branch conditions that compare the input value with a number (100) and a string ('auth'). Among all tested tools, only Zelda successfully detected this vulnerability. Burp attempted identical values for all parameters, and Wapiti assigned predefined values to all parameters and tried an alphanumeric string once for one parameter. These strategies contributed to the failure to reach the vulnerable sink `echo`. BlackWidow and wfuzz injected predefined exploits into all parameters, but their inputs failed to pass the branch conditions. WebFuzz also failed to generate inputs that reached the sink, due to limited mutation and input generation capabilities. In contrast, Zelda randomly mutated and assigned different values to each parameter, allowing it to satisfy the conditions and reach the vulnerable sink.

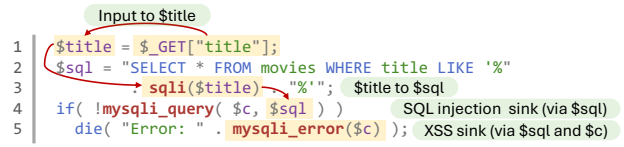


Figure 6: Example of bWAPP.

bWAPP. Figure 6 describes a multi-conditional vulnerability in the bWAPP application, where the *injected input* (via `$sql`) is reflected in SQL *error messages* at Ln 5. Exploiting this vulnerability requires the injected input to both provoke an SQL error and inject functional XSS payloads. Most fuzzers, except Zelda and WebFuzz, failed to detect this vulnerability. Burp and Wapiti failed for the same reasons described in the previous example. BlackWidow prioritized crawling and skipped injecting exploits into input parameters, thus not attempting any exploits. In contrast, both Zelda and WebFuzz successfully generated inputs that induced SQL errors, subsequently triggering the XSS vulnerability.

A.6 Hyperparameters

To determine optimal hyperparameters, we evaluated the performance of Zelda with varying hyperparameters. We used bWAPP, DVWA, XVWA, and MiniBench for this evaluation. For each hyperparameter set, we ran Zelda five times and recorded the maximum, minimum, and median number of detected vulnerabilities. Figure 7 shows the results.

Input parameter selection. When selecting an input parameter to mutate, Zelda computes a parameter score using Equation 3 (§5.3). This score consists of the exploration term, the importance term, and the content change term, controlled by β, γ, and δ.

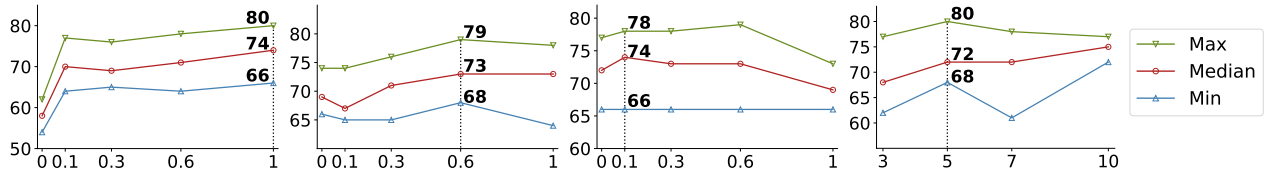
To determine the optimal hyperparameter values, we conducted experiments by varying one hyperparameter while keeping the others fixed. Specifically, we adjusted the value of β to 0, 0.1, 0.3, 0.6, and 1.0, while maintaining γ and δ at 0.1 to ensure all terms were considered (with non-zero values). Figure a shows that Zelda detected the highest number of vulnerabilities when β was set to 1.0.

Similarly, we tested different values of γ (0, 0.1, 0.3, 0.6, and 1.0) while keeping β and δ at 1.0 and 0.1, respectively. Figure b shows that Zelda achieved the best performance when γ was set to 0.6.

Finally, with β and γ fixed at 1.0 and 0.6, we explored δ values of 0, 0.1, 0.3, 0.6, and 1.0. Figure c reveals that Zelda achieved the best results when δ was set to 0.1, and the performance declined as δ increased. It demonstrates that each term in Equation 3 affects detection performance, and we set β, γ, and δ to 1.0, 0.6, and 0.1 in all experiments.

Fuzzing timeout. Another hyperparameter affecting the performance of Zelda is the fuzzing timeout, which is assigned for a fuzzing attempt on each URL. Zelda does not have specific termination conditions for each fuzzing process. Therefore, assigning a longer timeout allows Zelda to spend more time sending fuzzing requests for each crawled URL.

We varied the timeout values to 3, 5, 7, and 10 seconds, and Figure d shows the change in detection performance according to



(a) **Exploration (β)** (b) **Importance (γ)** (c) **Content Changes (δ)** (d) **Timeout**

Figure 7: The number of detected vulnerabilities while varying the hyperparameters.

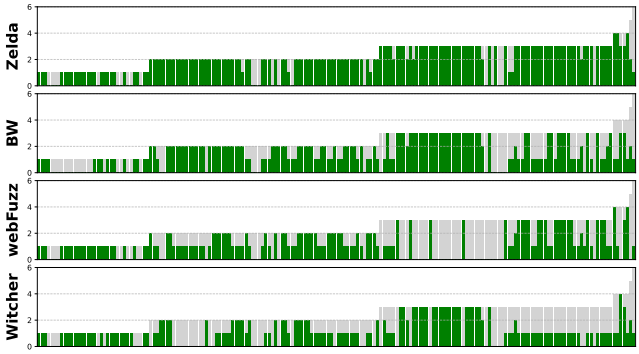


Figure 9: Path coverage of Zelda and other three fuzzers that trigger in-depth code blocks. Gray indicates the depth of code blocks containing vulnerable sinks from the entry points, while green represents the depth of covered blocks.

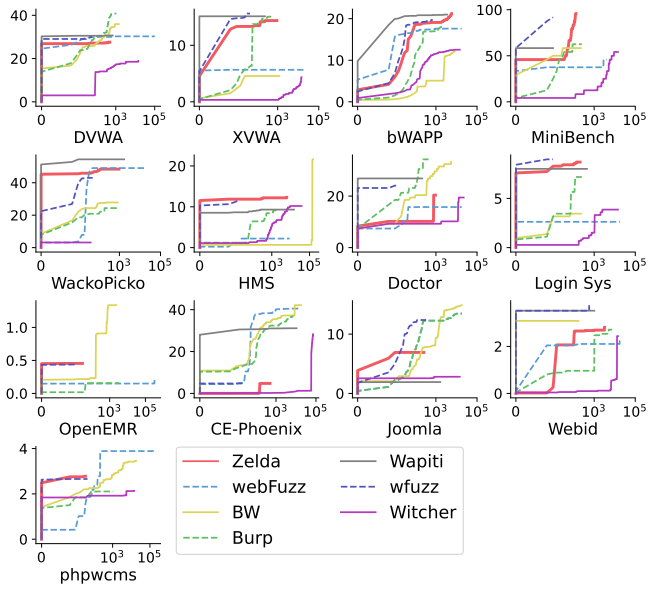


Figure 10: Cumulative coverage (%) over elapsed time (sec).

the timeout. Increasing the timeout from 3 to 5 seconds resulted in a 3.7% increase in the maximum number of detected vulnerabilities. However, further increasing the timeout from 5 to 10 seconds did not result in any additional increase in the maximum number of detected vulnerabilities. Therefore, we set the fuzzing timeout to 5 seconds.

A.7 Coverage Overhead

We investigated the overheads of state-of-the-art code coverage tools by measuring the loading time of each web application index page across three code coverage libraries (i.e., PCOV [65], Xdebug [50], and WebFuzz [60]), with and without code coverage computation. Note that PCOV and Xdebug are PHP code coverage libraries commonly used for PHP unit tests. WebFuzz provides an instrumentation library supporting real-time coverage collection. Figure 8 shows the differences in page load times with and without each code coverage library. The average overheads for PCOV, Xdebug, and WebFuzz instrumentation are 76.8%, 72.8%, and 226.49%, respectively. The maximum overheads reached 267.4%, 181.5%, and 748.75% in WordPress, respectively.

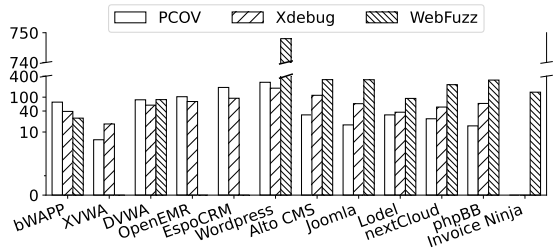


Figure 8: Loading time overhead of three different PHP coverage libraries.

A.8 Coverage

Figure 10 presents the cumulative coverage of the instrumented benchmark set. Among the 13 applications, BlackWidow achieved the highest coverage in four, making it the most effective overall, while wfuzz ranked second in three applications. Both Zelda and Burp attained the highest coverage in two applications.

BlackWidow demonstrated superior crawling capabilities by incorporating user interactions, allowing it to cover more pages than other tools. wfuzz relied on large input dictionaries and brute-force techniques, testing all injection points without analyzing target responses. As a result, while wfuzz explored a broad target space, it failed to reach many vulnerabilities, as shown in Figure 4.

Zelda achieved the highest coverage in the two applications and ranked second in three. Since Zelda does not prioritize crawling strategies, it struggled with overall coverage. Nonetheless, in terms of path coverage for vulnerabilities (Figure 4), Zelda proved more effective at identifying and reaching vulnerable pages compared to other tools. These results indicate that increasing overall coverage expands the target space but does not necessarily lead to better identification of vulnerable entry points.