Effective Directed Fuzzing with Hierarchical Scheduling for Web Vulnerability Detection

Zihan Lin, Yuan Zhang, Jiarun Dai, Xinyou Huang, Bocheng Xiang, Guangliang Yang, Letian Yuan, Lei Zhang, Fengyu Liu, Tian Chen, and Min Yang

Fudan University

Abstract

Java web applications play a pivotal role in the modern digital landscape. Due to their widespread use and significant importance, Java web applications have been one prime target for cyber attacks. In this work, we propose a novel directed fuzzing approach, called WDFUZZ, that can effectively vet the security of Java web applications. To achieve this, we address two main challenges: (1) efficiently exploring numerous web entries and parameters, and (2) generating structured and semantically constrained inputs. Our WDFUZZ approach is two-fold. First, we develop a semantic constraint extraction technique to accurately capture the expected input structures and constraints of web parameters. Second, we implement a hierarchical scheduling strategy that evaluates the potential of each seed to trigger vulnerabilities and prioritizes the most promising seeds. In our evaluation against 15 real-world Java web applications, WDFUZZ achieved a 92.6% recall rate in the known vulnerability dataset, finding 3.2 times more vulnerabilities and detecting them 7.1 times faster than the state-of-the-art web fuzzer. We also identified 92 previously unknown vulnerabilities, with 4 CVE IDs and 15 CNVD IDs assigned to date.

1 Introduction

Java web applications play a pivotal role in the modern digital landscape. They can facilitate convenient interaction and provide seamless access to diverse functionalities in various areas, such as business, education, healthcare, and entertainment. However, web applications can often be vulnerable to various security threats, including SQL injection, command injection, server-side request forgery (SSRF), arbitrary file read/write, and server-side template injection (SSTI). Recent statistics showed that over 75% of web applications have at least one defect, with 24% being regarded as serious flaws [37]. Given the extensive deployment of web applications and the potential impact of such security flaws, it is crucial to design an effective method for the security vetting of web applications.

Fuzzing is one of the most effective security analysis techniques, contributing to the discovery of the most dangerous vulnerabilities [17, 36]. It has been widely applied in several fields, including desktop software, network protocols, and Linux kernel. Despite its success, it is a challenging task to apply fuzzing to the security analysis of Java web applications. Two main challenges are faced and must be carefully dealt with.

- **Challenge-1: Efficiently exploring numerous web entries and parameters.** Web applications often have large code spaces that contain a large number of entry points and parameters. In practice, such multi-dimensional large search spaces make it difficult to efficiently pinpoint the vulnerabilities.
- **Challenge-2: Generating structured and semantically constrained inputs.** Web applications often require complex, structured inputs with specific semantic constraints for thorough testing. These inputs are difficult to generate using conventional fuzzing methods, making it challenging to effectively uncover vulnerabilities.

There are a few efforts made to address these challenges. However, they faced weaknesses that hindered their effectiveness in identifying vulnerabilities in Java web applications. Black-box web scanners [2, 5, 23, 24] often struggle with low coverage rates due to their reliance on the increasingly complex web front-end interactions to obtain feedback, failing to explore deeper vulnerabilities adequately. Grey-box web fuzzers [27, 28, 38, 42, 45, 49] also exhibit critical limitations. They often explore a substantial amount of code unrelated to vulnerabilities due to the adoption of coverage-based guidance. Meanwhile, their simplistic scheduling strategies fail to identify and prioritize the most promising web entry points for triggering vulnerabilities, instead wasting time and budget on less likely candidates, which ultimately reduces the overall efficiency of vulnerability detection. Additionally, these approaches typically rely on incomplete black-box crawlers to extract web parameters, overlooking the structured nature of inputs and their semantic constraints, further limiting their ability to generate valid test cases.

In this paper, we present a novel fuzzer, WDFUZZ, that can effectively detect 6 types of common critical vulnerabilities such as command injection, SQL injection, and arbitrary file read/write in Java web applications. Our approach is motivated by several key insights that help address the inherent challenges of exploring extensive code spaces and generating structured inputs with semantic constraints. First, although there are numerous entry points in a web application, our experiment result shows that only 4.8% of the web entries involve security-sensitive operations and have the potential to trigger web vulnerabilities. A natural idea arises that directed fuzzing can be utilized to shrink the code space to be analyzed. To achieve this, it is critical to guide fuzzing on the critical paths that connect user inputs (via entry points) to sensitive operations. Second, we observe that the difficulty of triggering vulnerabilities varies across different web entry points and sink locations due to the varying levels of input validation, sanitization, and processing logic employed at different parts of the application. This variance motivates us to design a novel hierarchical scheduling strategy that prioritizes testing resources toward the most promising paths, maximizing the likelihood of successful exploitation.

Based on these insights, we design WDFUZZ with two main phases: 1) static preparation, which understands and extracts necessary semantics, including sensitive paths to sinks and path constraints, for further guiding fuzzing; 2) dynamic fuzzing, which explores the paths with a hierarchical scheduling strategy. Specifically, in the static preparation phase, WDFUZZ performs entry and sink discovery to pinpoint critical code areas. Following this, WDFUZZ conducts vulnerable path extraction for guiding fuzzing with identifying potential exploitation paths and semantic constraint extraction. Specifically, WDFUZZ gathers necessary information on input parameters, ultimately constructing a request tree that accurately reflects the structures and constraints of input data.

In the second fuzzing phase, we propose a hierarchical scheduling strategy to evaluate and prioritize the most promising seeds for fuzzing. The selected seeds are then mutated to generate web requests, which are executed against the web application to collect distance feedback to assess the exploration of vulnerable paths. Any requests triggering the bug oracle are documented in a comprehensive vulnerability report. As a result, WDFUZZ strategically targets high-risk code areas, significantly enhancing the efficiency of vulnerability detection in Java web applications.

Last, we evaluate the effectiveness and efficiency of WDFUZZ on a diverse dataset consisting of 15 widely used popular Java web applications (including both open-source applications and closed-source commercial ones) and 68 known vulnerabilities. WDFUZZ achieved a remarkable 92.6% (63/68) recall rate for known vulnerabilities, which is 3.2 times more than the state-of-the-art web fuzzer, Witcher [42]. Additionally, WDFUZZ demonstrated a significant reduction in the time-to-exposure (TTE) for vulnerabilities, with an 87.69% decrease compared to Witcher. In terms of vulnerability detection efficiency, WDFUZZ can identify 59.39 vulnerabilities per hour – 7.1 times faster than Witcher. Moreover, WDFUZZ successfully identified 92 previously unknown vulnerabilities, all of which were responsibly reported to the relevant developers. Among these, 19 have been confirmed and fixed by vendors, with 4 CVE IDs and 15 CNVD IDs assigned to date. These vulnerabilities consist of high-risk vulnerabilities such as SQL injection, arbitrary file read/write, and SSRF. The security impact is significant, as it can lead to unauthorized access, data breaches, disruption of web services, and potential system takeover. Ablation studies further proved that each component of WDFUZZ contributed positively to its overall performance, validating the effectiveness of our approach in enhancing web application security.

In summary, in this paper, we make the following contributions:

- We propose the first directed fuzzing approach specifically designed for web applications, integrating a novel parameter structure and constraint extraction scheme along with an advanced hierarchical scheduling strategy to enhance vulnerability detection efficiency.
- We develop and release an open-source prototype fuzzer, WDFUZZ, which detects vulnerabilities in Java web applications effectively and efficiently.
- Experimental results demonstrate that WDFUZZ significantly outperforms existing state-of-the-art web fuzzers, while identifying 92 previously unknown web vulnerabilities.

2 Background and Motivation

In this section, we first discuss the background of web application security. Subsequently, we review existing web vulnerability detection approaches and their limitations, which are the motivation of our research.

2.1 Web Application Security

Java web applications play a pivotal role in the modern digital ecosystem, offering flexible user access and diverse functionalities. Due to insecure development practices and the ever-evolving nature of cyber threats, modern Java web applications inevitably suffer from various vulnerabilities, including SQL injection, command injection, server-side request forgery (SSRF), arbitrary file read/write, and server-side template injection (SSTI).

Detecting vulnerabilities in Java web applications remains an open challenge due to several factors. First, the extensive codebase of these applications often contains numerous functionality entries and execution paths, making it impractical to explore the entire code space, especially given the low throughput of web applications. Second, the prevalence of customized protocols with complicated input formats and composite semantic constraints further hinders the effectiveness of testing. As thoroughly discussed in §2.2, existing approaches often find it difficult to effectively address these challenges, which limits their ability to uncover vulnerabilities within a limited time budget. This highlights the need for innovative approaches that can effectively address the unique challenges posed by Java web applications.

2.2 Existing Work and Limitations

In recent years, the detection of vulnerabilities in web applications has predominantly followed three lines of methodologies, namely static analysis, black-box scanning, and grey-box fuzzing. Although these works have made nice first cuts in this problem domain, they still have non-negligible limitations that significantly hurt the effectiveness and efficiency of vulnerability detection (detailed as follows).

Static analysis techniques (e.g., TChecker [33], ANTaint [46], and JackEE [9]) have been developed to automatically analyze source code for potential security flaws. The main idea is to track malicious user input from the web entry point to some security-sensitive operations to find potentially vulnerable execution paths in web applications. A common known issue of these techniques is the high false positive rates [20, 30, 44], which can cost a lot of manual efforts to analyze the bug reports to find genuine security threats. Furthermore, static methods struggle to generate practical proof-of-concept (PoC) exploits, limiting their usability in real-world scenarios like verifying and fixing the detected vulnerabilities.

Black-box scanners (e.g., Burp Suite [2], Wapiti [7], and OWASP ZAP [5]) focus on testing running web applications without access to the underlying code. While these scanners are useful for identifying vulnerabilities from an attacker's perspective, they generally suffer from low coverage and fail to explore deeper vulnerabilities within the application [23, 31]. Their effectiveness is further limited by the reliance on complex front-end interactions to gather feedback, which may not adequately represent all potential attack vectors.

Grey-box fuzzing has emerged as a promising approach by combining dynamic and static analysis techniques to leverage internal information of programs to enhance vulnerability detection efficiency. It presents significant advantages over traditional static and black-box methods, making it an appealing option for testing web applications. Various techniques, such as webFuzz [45], Witcher [42], Atropos [28], and CeFuzz [49], have been developed within this domain. However, these approaches still fail to address key challenges, limiting their effectiveness in detecting vulnerabilities. Existing methods mainly rely on black-box crawling to generate initial seeds, which often results in incomplete attack surface discovery and high false negative rates. Additionally, they typically use coverage-guided feedback, which is inefficient and wastes time on branches that do not lead to vulnerabilities. The nested structures and complex semantic constraints of web application parameters also make it difficult for these methods to generate valid test cases. These limitations highlight the pressing need for a more effective fuzzing methodology to enhance vulnerability detection in web applications, which we will explore in this paper.

3 WDFuzz Overview

To enhance the security of web applications, we must address a critical question: *How can we effectively fuzz web applications to uncover vulnerabilities*? As discussed in the previous sections, due to the complexity and vast code space of web applications, coverage-guided fuzzing approaches are insufficient. This necessitates the adoption of directed fuzzing, which allows us to strategically prune the search space. Furthermore, unlike memory corruption vulnerabilities in binary programs, where memory operations are ubiquitous and finding potentially vulnerable operations can be challenging, web vulnerabilities are strongly related to specific security-sensitive operations, such as database manipulations and network requests. These operations can be more readily identified through static analysis, making them suitable targets for directed fuzzing. Therefore, utilizing directed fuzzing is helpful and essential for improving vulnerability detection efficiency in web applications. While directed fuzzing is not a novel concept and has been successfully applied to fuzz binary software, it still encounters unique challenges in the context of web application vulnerability detection.

3.1 Challenges

Fuzzing is one of the most effective vulnerability detection approaches. To apply fuzzing to the security of web applications, two primary challenges must be addressed, i.e., 1) how to efficiently explore the web entries and parameters under the throughput limitations, and 2) how to address the nested structure and complex semantic constraints posed on web application input.

Challenge-1: Exploring numerous web entries and parameters with low throughput. The first challenge is exploring the extensive number of entry points and parameters within web applications, coupled with the low testing throughput. Web applications typically expose hundreds and thousands of entry points, with each entry point often requiring multiple parameters for testing. For instance, a single commercial Java web application may present over 27,000 accessible entry points, each having an average of 10 input parameters. Even if we use static analysis to prune the entry points that are unlikely to trigger vulnerabilities, there still remain over 1,200 entries, representing a large exploration space.

Additionally, the throughput for fuzzing web applications is very low, frequently dropping below 15 execs/s, in contrast to the over 500 execs/s typically achieved in binary program fuzzing [1]. Existing research also suggests that the throughput of web applications can decline to as low as 1.25 execs/s [28]. This significant reduction in fuzzing throughput limits the ability to explore and test web applications within a reasonable time budget. As a result, efficiently navigating this complex landscape of entries and parameters is a critical challenge for effective fuzzing.

Challenge-2: Generating structured and semantically constrained inputs to test web parameters. Many web applications necessitate inputs in complex formats, such as JSON or XML, which are composed of nested key-value pairs. This structural complexity presents significant challenges for fuzzing, as it is difficult to infer appropriate keys and values, as well as the overall structure that meets the program's requirements. Consequently, generating well-formed requests that can be accurately processed by the application becomes a hard task.

Furthermore, web applications often impose strict semantic constraints on these inputs, such as the specific formats like dates and email addresses, as well as the constraints within the application code. These constraints further complicate the input generation process, as inputs must not only adhere to the required structure but also have the expected values of the application. The inability to generate valid and meaningful inputs significantly limits the effectiveness of a fuzzer in uncovering vulnerabilities within web applications.

Motivating Example. Figure 1a shows an example involving multiple web entry points and two sink locations (i.e., the code locations performing security-sensitive operations) within a Java web application. The Java methods `deptMapper.updateDept` and `deptMapper.deleteDept` in line 10 and line 15 are potential SQL injection sinks, since the configuration of the underlying database operations uses the insecure `${params.dataScope}` placeholder which directly injects the parameter values into the SQL statements (line 3 in Figure 1b).

Detecting the vulnerability illustrated in Figure 1 is challenging. Firstly, among all the code paths in this example, only the path highlighted in red – from entry point 1 (the `edit` method in line 2) to sink location 1 (the `deptMapper.updateDept` method in line 10) – is exploitable. Existing work typically applies a simple fixed scheduling strategy that uniformly explores all web entries, wasting a lot of time on unpromising entries that cannot trigger vulnerabilities. For instance, in this example, entry 2 is not vulnerable due to the external configuration `config.allowDelete` is set to false. As a result, a fixed scheduling strategy treats the two entry points equally, leading to insufficient exploration of the promising entry point 1 while wasting time on the unpromising entry point 2.

Moreover, existing solutions may fail to generate valid and meaningful inputs. In this example, the payload that triggers the vulnerability must satisfy the following structural and semantic constraints: ❶ *Input semantics*: The input must comply with complex validations like email and date formats to ensure the web application accepts the provided input. ❷ *Input structure*: The parameters must adhere to the nested structures. For instance, the `params` parameter must be a map that contains a key named `action`, whose value must be a structured JSON string. This JSON must have nested keys for `modify` or `delete`, both of which must be boolean true. ❸ *Application checks*: The `status` field must be "open" for the editing process to proceed past the necessary checks.

```
1  @PostMapping("/edit")     /* web entry 1 */
2  void edit(@Validated SysDept dept) {
3    if ("open".equals(dept.getStatus()))
4      updateDeptStatus(dept);
5    checkAndDeleteDept(dept);
6  }
7  void updateDeptStatus(SysDept dept) {
8    Map action = JSONObject.parse(dept.getParams().get("action"));
9    if (action.getBoolean("modify")) {
10     deptMapper.updateDept(dept); // sink location 1
11 }}
12 void checkAndDeleteDept(SysDept dept) {
13   // not triggerable due to configuration
14   if (action.getBoolean("delete") && config.allowDelete) {
15     deptMapper.deleteDept(dept); // sink location 2
16 }}
17 @PostMapping("/delete")   /* web entry 2 */
18 void delete(@Validated SysDept dept) {
19   checkAndDeleteDept(dept);
20 }
21 /* a lot of other web entries */
22 @PostMapping("/other")
23 void otherWebEntries() { ... }
24 /* explicit parameter definition */
25 class SysDept {
26   Long deptId;
27   @Email String email;
28   String status;
29   @JsonFormat(pattern="yyyy-MM-dd HH:mm") Date updateTime;
30   Map<String, Object> params;
31 }
```

(a) Vulnerable web application.

```
1 <update id="updateDept">   <!-- SQL statement configuration -->
2   UPDATE sys_dept SET status = #{status}
3   WHERE dept_id IN ${params.dataScope}
4 </update>
```

(b) Database operation configuration for `deptMapper.updateDept`.

```
1 status=open
2 &deptId=101
3 &email=fuzz@example.com
4 &updateTime=2024-01-01 08:00
5 &params[action]={"modify":true}
6 &params[dataScope]='payload
```

(c) Payload to exploit the above web vulnerability.

Figure 1: Motivating example from a real world vulnerability.

Existing fuzzers can barely extract these semantic constraints, and are generally incapable of generating inputs with valid nested structures such as maps and JSON, which are prevalent in modern web applications.

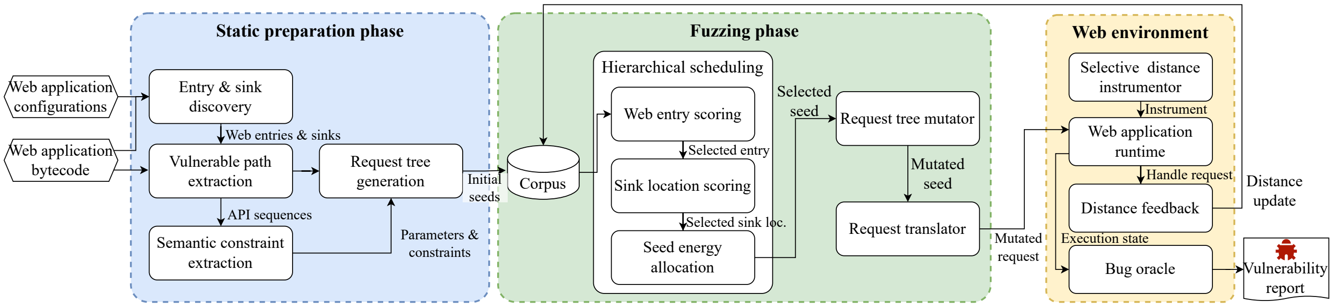


Figure 2: The system architecture of WDFUZZ.

3.2 Our Main Idea

Upon the above challenges, our main idea is to apply directed fuzzing to focus limited testing resources on high-risk code areas within web applications, thereby maximizing the efficiency of vulnerability detection. However, as discussed in §3.1, there remain several difficulties to solve. Through our observations of the characteristics of web applications, we summarize several key insights below that help us design an effective directed fuzzer.

Observation #1: *The difficulties for triggering vulnerabilities differ across various web entry points and sink locations.* Some entry points and their associated sink locations may present easier conditions for exploitation than others. By recognizing this variance, we can allocate more testing resources to those locations that are more likely to yield successful exploitation. We design a novel hierarchical scheduling technique to identify and prioritize the most promising entries, sink locations, and seeds, which maximizes the chances of discovering vulnerabilities in web applications.

Observation #2: *The web application development commonly involves using well-known frameworks and libraries to configure accessible entry points and handle user inputs.* For example, statistics indicate that the Spring framework holds 81% of the market share for Java server-side web applications [6]. As a result, this prevalent development pattern enables the effective extraction of web entry points and API sequences that process inputs, which helps understand the expected inputs of a web application. Importantly, these API sequences contain extensive information about inputs, including data types, nested relationships, and validation rules, thereby providing insights into the relationships between parameters and their expected formats.

Based on these observations, we propose a novel directed fuzzer, WDFUZZ, to tackle the challenges of exploring expansive web application codebases and generating structured, semantically constrained inputs.

3.2.1 Workflow

As illustrated in Figure 2, the overall process for WDFUZZ involves two main phases: the static preparation phase and the fuzzing phase.

Static Preparation Phase. WDFUZZ takes the bytecode and configuration files of the target web application as the input. The static preparation phase begins with *entry and sink discovery* to identify all the entry points and sink call sites in the web application. WDFUZZ then performs *vulnerable path extraction* through static analysis to identify potentially vulnerable paths from entries to sinks. These paths can later be used to precisely extract constraints and construct initial seeds.

Next, *semantic constraint extraction* is conducted to gather information on the structure and constraints of input parameters based on the API sequences on the vulnerable paths. By leveraging web application development patterns, we can systematically develop a precise semantic constraint extraction technique, enabling the inference of the structure (e.g., nested JSON objects), and semantic constraints (e.g., emails, date formats, and checks in application code) of web parameters.

This information collectively contributes to the *request tree generation*, where initial seeds for fuzzing are generated with a tree-based representation, accurately reflecting the structure and constraints of input parameters. Additionally, the static preparation phase also provides information for the distance instrumentation on the web application runtime.

Fuzzing Phase. After the static preparation phase, the generated initial seeds are put into a *corpus*. WDFUZZ employs the *hierarchical scheduling* technique, which evaluates the potential of web entries, sink locations, and seeds based on multiple factors to prioritize the most promising seeds for uncovering vulnerabilities. This targeted approach ensures that WDFUZZ focuses on the most promising seeds first, improving the efficiency and effectiveness of our fuzzing process.

The selected seeds are then mutated using the *request tree mutator* and translated into mutated requests to test the web application. During runtime, these mutated requests are executed, and *distance feedback* is collected to evaluate the fuzzing progress in exploring the vulnerable paths. This feedback helps determine if a seed is "interesting" enough to warrant further mutations and provides information for the future evaluation of seeds. Finally, any request triggering the *bug* *oracle* will be reported, and a comprehensive *vulnerability report* is generated, detailing the discovered issues.

4 Design and Implementation

In this section, we present the design of WDFuzz and deploy a prototype specifically targeting Java web applications. Note that WDFuzz only requires the bytecode of target applications as input. In the paper, we use source code in the examples for illustrative purposes and readability.

4.1 Static Preparation Phase

In this phase, our goal is to identify the vulnerable code areas within web applications and to extract the corresponding structures and constraints of web parameters. This workflow includes the discovery of entry points and sinks, the extraction of vulnerable paths, and the identification of semantic constraints associated with input parameters.

4.1.1 Entry and Sink Discovery

In order to statically find vulnerable paths within the web applications, the first step is to define and detect the feasible entries and sinks in the applications, which is challenging due to the complexity and variability of web application architectures. Our primary insight stems from the observation that the majority (over 81%) of Java web applications are developed using well-established web frameworks such as Servlet and Spring framework. By harnessing this insight, WDFuzz incorporates a method that systematically matches common development patterns inherent to these frameworks to effectively identify web entry points and sink locations, which are crucial for directed fuzzing.

Entry Point Identification. The goal of detecting web application entry points is to identify as many attack vectors as possible, which is crucial for discovering more vulnerabilities. We comprehensively construct development patterns for five popular Java web frameworks based on their documentation, including Spring[1], Struts2[2], Servlet[3], JSP[4], and Javax-standard REST APIs[5]. As for the example in Figure 1a, WDFuzz is able to identify the `edit` and `delete` as web entry points, since its route is explicitly defined by the `@PostMapping` (line 1 and line 17) annotation from the Spring framework.

Sink Identification. WDFuzz employs two primary strategies to identify sinks and find sink locations. Firstly, WDFuzz relies on a manually predefined sink list, including common security-sensitive operations like SQL execution, command execution, SSRF, etc. These predefined sinks, as listed in Appendix A, are derived from well-known patterns and practices in web application security [47], ensuring that our analysis covers the most critical areas where web vulnerabilities are likely to occur.

Secondly, WDFuzz also incorporates a complementary sink identification strategy to accurately identify the sinks introduced during runtime. The necessity lies in that, some security-sensitive operations in Java web applications are merely observed during runtime. For example, frameworks like MyBatis[6] for database operations and Thymeleaf[7] for template rendering can bind sensitive operations to user-defined Java methods during the application's execution. To be specific, WDFuzz relies on a set of static detection rules derived from the framework documentation for identifying these runtime sinks. For instance, WDFuzz can identify `deptMapper.updateDept` as a sink that performs SQL operations by applying a static detection rule that analyzes the SQL mapping configurations in Figure 1b.

4.1.2 Vulnerable Path Extraction

This module is designed to efficiently identify and extract vulnerable paths through taint analysis. The process of extracting vulnerable paths involves not only tracing paths from web inputs to sink locations but also tracking the user-input web parameters along these paths to determine whether these inputs can influence any sinks. WDFuzz extracts those paths that lead to sinks potentially influenced by user inputs, which is significant indications of vulnerabilities.

Java web applications utilize two distinct types of web parameters: ➊ *Data-binding parameters*, which are directly bound from the request to the arguments of entry methods by web frameworks; and ➋ *Runtime-fetched parameters*, which are dynamically retrieved and processed through APIs such as `request.getParameter`. WDFuzz begins by tainting the data-binding parameters of the web entry methods. For the parameters of class types, WDFuzz recursively taints their fields, including those inherited from superclasses. Additionally, when WDFuzz identifies call sites related to the retrieval of runtime-fetched parameters, it taints the corresponding return values. Then WDFuzz propagates the taint through the inter-procedural control flow graph (ICFG), seeking to find taint paths from the web parameters to the sink locations. When a taint path is found, WDFuzz can also determine a path from an entry to a sink location, where the sink is potentially controllable by the web parameters.

To enhance the precision of our taint analysis, we support dependency injection features commonly used in Java web frameworks [19]. WDFuzz generates singleton heap objects for injected class fields and connects corresponding call edges

[1] https://spring.io/

[2] https://struts.apache.org/

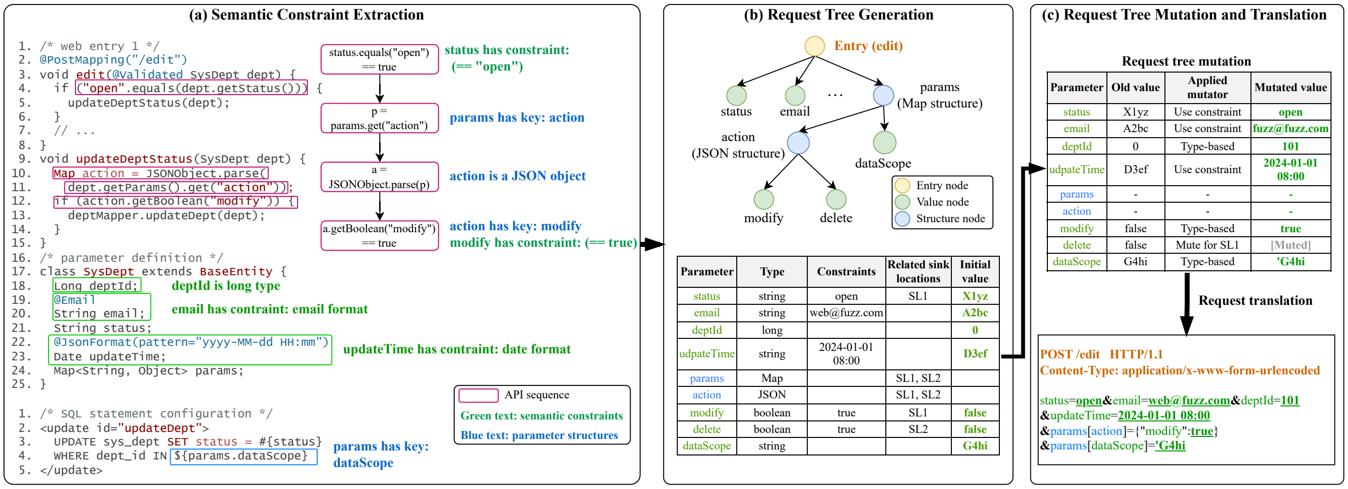
[3] https://docs.oracle.com/javaee/7/tutorial/servlets.htm

[4] https://docs.oracle.com/javaee/5/tutorial/doc/bnajo.html

[5] https://docs.oracle.com/javaee/7/tutorial/jaxrs.htm

[6] https://mybatis.org/mybatis-3/

[7] https://www.thymeleaf.org/


(a) Semantic Constraint Extraction
status.equals("open") == true
status has constraint: (== "open")
p = params.get("action")
params has key: action
a = JSONObject.parse(p)
action is a JSON object
a.getBoolean("modify") == true
action has key: modify
modify has constraint: (== true)
deptId is long type
email has contraint: email format
updateTime has contraint: date format
params has key: dataScope
API sequence
Green text: semantic constraints
Blue text: parameter structures
(b) Request Tree Generation
Entry (edit)
status
email
...
params (Map structure)
action (JSON structure)
dataScope
modify
delete
Entry node
Value node
Structure node
(c) Request Tree Mutation and Translation
Request tree mutation
Request translation

```
/* web entry 1 */
@PostMapping("/edit")
void edit(@Validated SysDept dept) {
  if ("open".equals(dept.getStatus())) {
    updateDeptStatus(dept);
  }
  // ...
}
void updateDeptStatus(SysDept dept) {
  Map action = JSONObject.parse(
    dept.getParams().get("action"));
  if (action.getBoolean("modify")) {
    deptMapper.updateDept(dept);
  }
}
/* parameter definition */
class SysDept extends BaseEntity {
  Long deptId;
  @Email
  String email;
  String status;
  @JsonFormat(pattern="yyyy-MM-dd HH:mm")
  Date updateTime;
  Map<String, Object> params;
}
```

```
/* SQL statement configuration */
<update id="updateDept">
  UPDATE sys_dept SET status = #{status}
  WHERE dept_id IN ${params.dataScope}
</update>
```

Parameter	Type	Constraints	Related sink locations	Initial value
status	string	open	SL1	X1yz
email	string	web@fuzz.com		A2bc
deptId	long			0
udpateTime	string	2024-01-01 08:00		D3ef
params	Map		SL1, SL2	
action	JSON		SL1, SL2	
modify	boolean	true	SL1	false
delete	boolean	true	SL2	false
dataScope	string			G4hi

Parameter	Old value	Applied mutator	Mutated value
status	X1yz	Use constraint	open
email	A2bc	Use constraint	fuzz@fuzz.com
deptId	0	Type-based	101
udpateTime	D3ef	Use constraint	2024-01-01 08:00
params	-	-	-
action	-	-	-
modify	false	Type-based	true
delete	false	Mute for SL1	[Muted]
dataScope	G4hi	Type-based	'G4hi

```
POST /edit   HTTP/1.1
Content-Type: application/x-www-form-urlencoded

status=open&email=web@fuzz.com&deptId=101
&updateTime=2024-01-01 08:00
&params[action]={"modify":true}
&params[dataScope]='G4hi
```

Figure 3: An illustration of the process of detecting the vulnerability in the motivating example in Figure 1.

in the call graph to accurately track tainted data, thereby increasing the reliability of the vulnerable path extraction.

4.1.3 Semantic Constraint Extraction

The main idea of this module is to extract the semantic constraints based on the API operation sequences in which the parameters are processed.

It first extracts the parameter names depending on the type of web parameters. For data-binding parameters, which are directly mapped to the parameters of entry methods, WDFUZZ processes all method parameters and extracts their names following the Java Bean conventions[8]. To extract the names of runtime-fetched parameters, we use a backward data-flow tracking approach on the arguments of parameter retrieval methods (e.g., `request.getParameter`). This technique allows us to trace back and identify the parameter names being passed and processed. As illustrated in Figure 3, WDFUZZ can identify that the web entry `edit` expects 5 data-binding parameters within the `SysDept` class, i.e., `deptId`, `email`, `status`, `updateTime` and `params`.

After WDFUZZ identifies all the parameters, it traces the taint paths of the parameters along the vulnerable paths found in §4.1.2. This tracing allows WDFUZZ to capture the API sequences that process these parameters and extract semantic constraints, including structural information, type constraints, and value constraints.

❶ *Structural Information.* Structural information is extracted by analyzing APIs involved in parameter structure processing within the API sequences. For instance, if a parameter is processed using the `JSON.parseObject` method, WDFUZZ can infer the parameter as a string representing a JSON object. Furthermore, if the following API call is `obj.getString("some_key")`, WDFUZZ can infer that this object contains a key named `some_key` with a string value.

❷ *Type Constraints.* Type constraints are derived based on analyzing APIs that imply type-specific operations in the sequences. For example, the presence of `Integer.parseInt` in the API sequence indicates that the parameter should be an integer. Additionally, if parameters are bound from class objects, WDFUZZ infers the parameter types based on the declaring types of the fields.

❸ *Value Constraints.* Value constraints are derived by analyzing APIs that compare or validate parameter values. For instance, if a parameter is subjected to validation through functions like `String.equals`, WDFUZZ will solve the constant values and extract them as value constraints.

As for the example in Figure 3, WDFUZZ first analyzes the data-binding web parameters to extract semantic constraints. Specifically, it identifies that the parameter `deptId` must be a *long* type value. Additionally, the parameters `email` and `updateTime` are found to have the *email* and *date* format validations. WDFUZZ also examines the configuration file to derive structural information about the Map-type parameter `params`. It reveals that there exists a key named `dataScope` within the `params` parameter.

WDFUZZ further performs taint tracking on the `dept` parameter and its fields, which allows it to extract a sequence of API calls that the web application performs on this parameter, as illustrated in the stage (a) of Figure 3 in magenta color. From this API sequence, WDFUZZ can uncover a series of structural and semantic constraints associated with the web parameters, e.g., `status` must equal "open"; the map `params` contains a key named `action`; the `action` key itself is a nested JSON object containing a key named `modify`, whose value must be true.

[8] https://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/

4.1.4 Request Tree Generation

WDFuzz employs a tree-based structure, i.e., request tree, to represent the initial seeds for each entry point of the web application due to the tree-like nature of web parameters, such as nested JSON and Map structures. Such structure can include all the information about extracted entry points, potential sink locations, parameter names, and semantic constraints. A request tree includes three different node types, each serving a distinct purpose:

- *Entry node* contains information related to the entry point, including the URL, HTTP method, and overall parameter passing method (URL-encoded form, JSON, XML, etc.) of the request.
- *Structure node* holds the structural information of parameters, such as the key names in a Map or a JSON object. Structure nodes do not contain concrete values but serve as hierarchical placeholders.
- *Value node* stores key-value pairs where the key is the parameter name and the value is the concrete value of the parameter in this seed. It also includes the semantic constraints extracted by the previous static analysis modules.

The design of the request tree ensures that each tree can be uniquely mapped to a specific HTTP request.

The stage (b) of Figure 3 shows the request tree generated for the motivating example. The structure of the request tree is mapped from the structural information extracted in the previous step. Each value node in the tree also retains the information about the constraints and the related sink locations, which can be used for later mutations.

4.2 Fuzzing Phase

The fuzzing phase is designed to actively engage with the web application, utilizing the previously identified entry points, sinks, and parameter constraints to uncover vulnerabilities through a series of targeted mutations and requests. We design a hierarchical scheduling to prioritize the most promising entry points and corresponding sink location, and utilize a set of mutation operators to mutate the request trees. The following sections delve into the specific methodologies. The entire fuzzing loop is summarized in Appendix B.

4.2.1 Hierarchical Scheduling

WDFuzz leverages hierarchical scheduling to assess and select the most promising entry points and corresponding sink locations that could trigger vulnerabilities in web applications. The hierarchical scheduling strategy follows a comprehensive evaluation process to prioritize the most promising seeds. Initially, each entry point is assessed for its potential to trigger vulnerabilities, prioritizing those with higher potential. Subsequently, after the web entry is selected, the potential of the selected entry to reach specific sink locations is also evaluated, giving precedence to locations that are easier to reach. Once both the targeted entry and sink location are determined, fuzzing energy (i.e., the number of mutations) is allocated to the seed from the corpus of the selected entry and sink location.

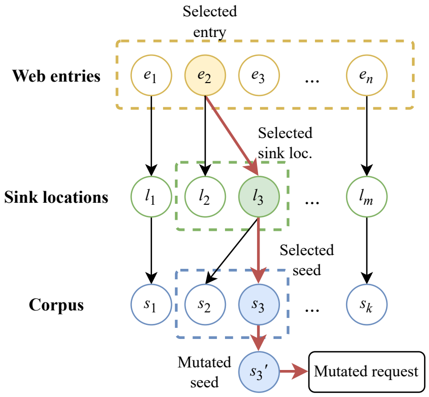


Figure 4: An illustration of the hierarchical scheduling process.

To facilitate this, a *scoring algorithm* is designed to calculate a score based on the potential of an entry point, taking into account multiple factors such as the distances to the sink locations, the number of times the entry point has been scheduled, the number of times the call site has been triggered, and the complexity of the entry point parameters. The scores of sink locations are calculated similarly.

We visualize this scheduling process in Figure 4. First, we sample a target entry point from the pool of all entry points based on their scores, for example, the entry point e_2 as shown in the figure. Next, from all the reachable sink locations corresponding to the selected entry, we sample a target sink location (e.g., l_3) based on the scores as well. Once both the entry and sink locations are determined, we sample the next seed from the corresponding corpus (a FIFO circular queue), like s_3 in the figure, and allocate its fuzzing energy, which is the number of mutations. The mutated seed s_3' is then translated into a request to test the web application.

The scoring algorithm is a critical component of our hierarchical scheduling strategy, designed to assess the level of "promising" nature of each entry point and corresponding sink location using multiple parameters, therefore maximizing the efficiency of triggering vulnerabilities. Existing scoring algorithms, e.g., Hawkeye [18], LOLLY [32] and SelectFuzz [34], typically focus on fuzzing a single entry to trigger known vulnerabilities at specific code locations. In contrast, as highlighted in §3.1, the design goal of WDFuzz's scoring algorithm is to explore large code space of web applications efficiently, which have numerous web entries and sink locations. To sum up, our proposed scoring algorithm is expected to offer two unique advantages: ❶ it leverages the hierarchical structure inherent in web applications, recognizing that different web entries, sink locations and parameters have varying potential for triggering vulnerabilities; ❷ it additionally considers a wider range of web-specific factors for achieving efficient and effective fuzzing, e.g., the complexity of web parameters.

Following the above design considerations, our scoring algorithm technically prioritizes those seeds with lower distances towards sink locations, fewer schedules, fewer reached times of sink locations, and lower complexity. Specifically, the score of an entry point e is defined as

$$s_e = -\bar{d}_e - \alpha n_{se} - \beta n_{te} - \gamma n_{ce} - \delta_e, \tag{1}$$

where $\bar{d}_e$ represents the current average distance of the entry e to all reachable sink locations, n_{se} represents how many times the entry e has been scheduled, n_{te} is the count of entry e successfully triggering sink locations, and n_{ce} is the complexity factor of the entry point e which is defined as the average number of mutable value nodes.

The distance $\bar{d}_e$ is initially calculated by averaging the number of basic blocks from the entry point e to each reachable sink. During the fuzzing process, the distance is updated whenever an execution path reaches a basic block which reduces the number of basic blocks required to reach any sink. The coefficients α, β, and γ are empirically set to fine-tune the scoring. δ_e is a penalty factor for those entries that have already triggered bug oracles. This multi-dimensional scoring allows for a balance between exploration and exploitation, ensuring that the fuzzer prioritizes the most promising web entries.

Following the scoring, a weighted sampling process is conducted based on the scores. This ensures that entry points with higher scores are prioritized in the fuzzing process. The weight used in sampling is defined by

$$w_{\mathrm{e}} = f(t)\exp(s_e), \tag{2}$$

where $f(t)$ is a time coefficient that increases from 0 to 1 over the elapsed time t, making the impact of the score grows over time. A sigmoid function, for example, can be used:

$$f(t) = \frac{1}{1+e^{-k(t-t_0/2)}}, \tag{3}$$

where k controls the steepness of the time coefficient's growth, and t_0 represents the time at which $f(t)$ approaches 1, indicating that the sampling process fully utilizes the scoring results.

Once the sampling of entry points is completed, WDFUZZ further calculates the scores for each sink location that can be reached from the selected entry points, based on a similar scoring process. The score s_l of a sink location l can be represented similar to Equation 1, and the weights for sampling sink locations are exactly the same as Equation 2.

WDFUZZ maintains a corpus for each pair of entry points and sink locations. When a seed updates the distance to the corresponding sink location or increases code coverage from the entry point, the seed is added to the corresponding corpus. Once WDFUZZ sampled and determined the current target entry point and sink location to cover, it sequentially retrieves seeds from the corresponding corpus based on a first-in, first-out (FIFO) circular queue. The number of times each seed should be mutated and tested, i.e., its energy, is calculated based on the AFLGo score [16]. In general, the AFLGo score would favor those seeds that are closer to the selected sink, so as to achieve the capability of directed fuzzing. Here, we adopt the original version of AFLGo score, and make necessary implementation customization to make it comply with our fuzzing framework. Specifically, given a seed s, its AFLGo score, denoted as $p(s)$, is calculated by the below formula,

$$p(s) = p_{\mathrm{afl}} \cdot 2^{10\cdot p-5}, \tag{4}$$

where p_{afl} is the standard AFL power. p is an annealing-based power scheduling factor calculated by

$$p = (1-\bar{d})\cdot(1-T_{\mathrm{exp}}) + 0.5T_{\mathrm{exp}}, \tag{5}$$

where $\bar{d}$ is the normalized distance of the seed s to the selected sink. The temperature T_{exp} gradually decreases from 1 to 0 with time, which is defined by

$$T_{\mathrm{exp}} = 1/\left(1+19\cdot\frac{t}{t_x}\right), \tag{6}$$

where t is the elapsed time since the start of the fuzzing campaign, and t_x is the exploration time which is set to 10 minutes as suggested by AFLGo.

4.2.2 Request Tree Mutator

Once a seed is selected, WDFUZZ employs three distinct mutation operators to transform the concrete nodes within the seed's tree structure, listed below.

❶ *Constraint Value Injection.* This operator directly applies extracted constraint values to the node's actual value. If the node has value constraints, there is a probabilistic chance that the mutation will use one of these constraint values.

❷ *Type-Based Mutation.* This involves creating random variations of the node's value based on its type. Our mutation strategies are carefully designed for each type of value. For string values, WDFUZZ inserts escape characters and common payloads, such as single quotes, to test for SQL injection, path traversal, and other common vulnerabilities. Additionally, we perform common case transformations, and random insertions and deletions to further diversify the input variations. For numerical values, WDFUZZ leverages Benford's Law [15] to generate values that mimic the natural distribution of numbers. We also tailor the value mutation process for

Table 1: Vulnerability detection results for WDFUZZ. The names and versions of the commercial closed-source web applications are anonymized due to legal reasons.

#	Application	Version	Known Vuln.	Detected Entries	Vulnerable Entries	WDFUZZ				Witcher			
						Detected Vuln.	Recall	Avg. TTE/s	Unknown Vuln.	Detected Vuln.	Recall	Avg. TTE/s	Unknown Vuln.
1	jeecg-boot	3.2.0	5	693	53	5	100.0%	30.0	16	2	40.0%	1560.5	0
2	jeesite	1.2.6	3	202	10	1	33.3%	3.0	5	0	0.0%	-	0
3	jshERP	2.3	8	70	52	7	87.5%	78.3	23	0	0.0%	-	0
4	MCMS	5.2.4	12	161	60	12	100.0%	34.3	4	1	8.3%	3481.0	0
5	RuoYi	4.5.1	11	160	18	11	100.0%	120.0	0	0	0.0%	-	0
6	SpringBlade	3.6.0	1	128	1	1	100.0%	1.0	0	1	100.0%	67.0	0
7	Halo	1.4.9	1	217	5	1	100.0%	30.0	0	0	0.0%	-	0
8	DreamerCMS	4.0.1	2	108	12	2	100.0%	13.5	0	0	0.0%	-	0
9	PublicCMS	4.0	6	170	12	4	66.7%	44.0	0	0	0.0%	-	0
10	Yudao	2.0.0	0	938	5	0	-	-	1	0	-	-	0
11	lamp-boot	3.10.0	0	102	3	0	-	-	1	0	-	-	0
12	Commercial-1	***	3	27816	1265	3	100.0%	98.0	30	0	0.0%	-	0
13	Commercial-2	***	1	2482	569	1	100.0%	120.0	8	0	0.0%	-	0
14	Commercial-3	***	3	10634	36	3	100.0%	233.3	4	0	0.0%	-	0
15	WebGoat	2023.9	12	40	15	12	100.0%	3.2	0	11	91.7%	64.8	0
	Total		**68**	**43921**	**2116**	**63**	**92.6%**	**60.6**	**92**	**15**	**22.1%**	**492.1**	**0**

specific types, such as dates and booleans, to produce random values.

❸ *Type-Changing Mutation.* This mutation operator randomly changes the type of a value node and generates a new value based on the new type. This can lead to significant changes in the input structure, potentially revealing type-related vulnerabilities during the processing of input parameters.

Since the sink location to be tested is determined before the mutation, WDFUZZ will probabilistically mute the parameters that are not related to the sink location under test. This approach helps further reduce the exploration space. The stage (c) in Figure 3 illustrates how WDFUZZ mutates a request tree with the above-mentioned operators and translates it into a concrete request.

4.3 Implementation

The implementation of WDFUZZ is divided into distinct static, dynamic, and instrumentation components, each contributing to the overall efficacy of the fuzzing process. For the static preparation phase, we implement the various modules described in §4.1 as plugins of Tai-e [40], a program analysis framework for Java. The static analysis component comprises 20.2k lines of Java code.

The dynamic fuzzing component is built on top of libAFL [26], a fuzzing framework. We implement the scheduling and mutation strategies in §4.2 with 4.7k lines of Rust code and 1.5k lines of Python code. We conducted a sensitivity testing over the hyper-parameters of our hierarchical scheduling technique using a dataset of 10 known vulnerabilities to fine-tune the parameters. The final parameter settings used in our experiments are: $\alpha = 0.3$, $\beta = 0.3$, $\gamma = 0.3$, $\delta_e = 3$.

The instrumentation serves two primary purposes: feedback collection and bug oracle. For feedback, we insert instrumentation at basic blocks along the vulnerable paths to provide distance information using a Java agent. The bug oracle of WDFUZZ employs two detection strategies: error-based and invocation-based. The error-based approach, similar to Witcher's fault escalation [42], captures error messages by intercepting system calls such as `write`, `execve`, and `send` with a preload shared library. However, there do exist some exceptions or errors that would be captured at the application layer (i.e., the Java layer) and cannot be observed at the system call layer. Hence, it is also essential to instrument Java methods to monitor whether their runtime arguments are controlled by malicious inputs. Considering the above, WDFUZZ incorporates an invocation-based oracle that instruments sensitive Java methods and exception constructors to detect attacker-controlled argument values. For example, to identify arbitrary file read/write vulnerabilities, WDFUZZ examines whether the web application opens arbitrary files specified in the input web parameters by checking the arguments of Java file APIs (e.g., `File.<init>`). Our bug oracle mechanism supports the detection of a wide range of vulnerabilities, including SQL injection, command execution, arbitrary file read/write, SSRF, and SSTI. The instrumentation component includes 4.7k lines of Java and 430 lines of C code.

5 Evaluation

In this section, we aim to answer the following research questions to evaluate the effectiveness and efficiency of WDFUZZ.

RQ1: How does WDFUZZ compare to state-of-the-art web fuzzer, i.e., Witcher [42], in terms of vulnerability detection capabilities? In order to evaluate the vulnerability detection capability of WDFUZZ, we conduct a comprehensive comparison between WDFUZZ and Witcher on a benchmark dataset of real-world vulnerabilities collected from diverse Java web applications.

RQ2: Can WDFUZZ identify previously unknown real-world vulnerabilities? This evaluation involves deploying WDFUZZ on various applications to detect previously unknown zero-day exploits, thereby demonstrating its effectiveness in practical scenarios.

RQ3: What is the contribution of each component of WDFuzz to its overall performance? This research question aims to analyze the individual components of WDFuzz, i.e., the vulnerable path extraction module, the semantic constraint extraction module, and the hierarchical scheduling algorithm, to assess their contributions to the overall effectiveness.

5.1 Dataset Construction

To convincingly evaluate the effectiveness of WDFuzz, we selected representative and popular Java web applications as our evaluation targets, mainly including both open-source ones and closed-source commercial ones. For the open-source applications, we considered popular Java web applications that were actively maintained by the open-source community. To be specific, we finally selected 11 applications with over 2,000 stars on GitHub, indicating a significant level of community interest and usage. Notably, these applications are all implemented using common web frameworks that are technically supported by WDFuzz. In terms of closed-source applications, we considered 3 target applications that are developed using common web frameworks and have publicly disclosed vulnerabilities. Besides, to further improve the diversity of target applications, we also incorporated the WebGoat project [4], an intentionally curated vulnerable application for security training purposes.

Given the above 15 target applications, we further curated a robust dataset of known vulnerabilities that affect these applications. Specifically, we conducted an extensive search on the CVE databases, and collected all the publicly disclosed vulnerabilities supported by our bug oracles. We attempted to manually reproduce these vulnerabilities, and those that could be successfully replicated were all included in our dataset. In total, our dataset comprises 68 known vulnerabilities, including diverse vulnerability types such as SQL injection, command injection, arbitrary file read/write, SSRF, and SSTI.

5.2 Result Overview

We evaluated WDFuzz on the dataset to answer the previously mentioned research questions. In terms of vulnerability discovery capability (§5.3), WDFuzz achieved a recall rate of 92.6% for known vulnerabilities, detecting 3.2 times more vulnerabilities compared to Witcher. Regarding the time-to-exposure for vulnerability reproduction, WDFuzz also demonstrated a 87.69% reduction in time compared to Witcher. Furthermore, the efficiency of vulnerability discovery is 7.1 times greater than that of Witcher.

In terms of discovering unknown vulnerabilities in real-world web applications, WDFuzz has identified 92 previously unknown vulnerabilities (§5.4). We have responsibly reported all the vulnerabilities, and 19 have been confirmed and fixed by vendors so far. Besides, ablation studies indicate that each module of WDFuzz contributes positively to the effectiveness of web vulnerability detection (§5.5).

5.3 RQ1: Reproducing Known Vulnerabilities

To evaluate the vulnerability detection capability of WDFuzz, we conducted a series of experiments on the known vulnerability dataset described in §5.1. Each entry point under test is allocated with a fuzzing time budget of 2 minutes for both WDFuzz and Witcher, as suggested in [42]. Meanwhile, we provided Witcher with the results of our static analysis for attack surface identification, as we found that the entry points identified by its crawler were too limited to obtain meaningful comparative results. Therefore, we actually enhanced Witcher's original approach for our comparative experiments.

At first glance, as shown in Table 1, the vulnerable path extraction technique has reduced the initial pool of 43,921 entry points under test to only 4.8% (2,116) of the entries, significantly narrowing the search space of web applications.

As for the recall of known vulnerabilities, WDFuzz successfully identified 92.6% (63/68) of the known vulnerabilities. In contrast, Witcher was only able to find 15 vulnerabilities, which means WDFuzz demonstrated 3.2 times greater effectiveness in vulnerability discovery. We analyzed the 5 vulnerabilities missed by WDFuzz and found that these vulnerabilities were all high-order vulnerabilities, which required sequentially triggering multiple web entries to exploit. WDFuzz is not designed to detect high-order vulnerabilities, but we believe that it is a promising future research area.

Furthermore, we compared the average time-to-exposure (TTE) of vulnerabilities between WDFuzz and Witcher. As illustrated in Table 1, WDFuzz not only discovered more vulnerabilities but also reduced 87.69% of the time required to expose the known vulnerabilities.

Finally, we evaluated the overall vulnerability discovery efficiency by measuring the number of vulnerabilities discovered per hour. WDFuzz could identify 59.39 vulnerabilities per hour, which is 7.1 times faster than Witcher.

In summary, these findings prove that WDFuzz outperforms existing state-of-the-art web fuzzers in terms of both effectiveness (finding more vulnerabilities) and efficiency (finding vulnerabilities faster).

5.4 RQ2: Identifying Unknown Vulnerabilities

As illustrated in Table 1, WDFuzz successfully identified 92 previously unknown vulnerabilities. These vulnerabilities lie in high-risk categories, including 83 SQL injections, 4 SSRFs, 4 arbitrary file reads/writes, and 1 SSTI, which pose significant threats including sensitive data breaches, disruption of web services, and potential takeover of operating systems. In contrast, the benchmark fuzzer, Witcher, failed to report any of these vulnerabilities.

```
 1 @PostMapping("/user/pageAll")
 2 public R page(@RequestBody PageParams params) {
 3   // model must be a BaseEmployeePageQuery object
 4   BaseEmployeePageQuery model = params.getModel();
 5   // scope must equal BIND or UNBIND
 6   if (StrUtil.equalsAny(model.getScope(), SCOPE_BIND,
        ↪ SCOPE_UN_BIND) && model.getRoleId() != null) {
 7     // insecure concatenation of SQL statement
 8     String sql = "select employee_id from err where employee_id =
        ↪ e.id and role_id = " + model.getRoleId();
 9     // scope must equal BIND (1)
10     if (SCOPE_BIND.equals(model.getScope())) {
11       // SQL injection sink location
12       wrap.inSql(BaseEmployee::getId, SQL);
13   }}
14   // ...
```

(a) The vulnerable code.

```
{"size": 1, "current": 1, "sort": "id",
 "model": {
  "scope": "1",
  "roleId": "'payload",
  "isDefault": false,
  "state": false,
  "userId": 1,
  "positionId": 1,
  "orgIdList": [],
  "email": "fuzz@example.com",
  "realName": "A1bc"
}}
```

(b) The payload of the vulnerability.

Figure 5: A real-world vulnerability discovered by WDFuzz in lamp-boot.

We have taken a proactive and responsible approach to report all discovered vulnerabilities to the corresponding developers and stakeholders. As of now, 19 of the vulnerabilities have been confirmed by the application developers. Here, we also analyzed the root causes of the 19 CVE-/CNVD-indexed vulnerabilities, as listed below.

- **Root cause 1: Direct concatenation of parameters (16/19).** Among these cases, user inputs are directly concatenated into SQL queries, URLs, or other sensitive parameters, consequently causing injection vulnerabilities. The vulnerability illustrated in Figure 5 is a typical example of this root cause.
- **Root cause 2: Misuses of framework APIs (3/19).** This root cause arises when framework APIs are improperly used. For instance, when configuring MyBatis framework, if developers pass an input parameter into a SQL query using `#{input}`, the query will be guarded by prepared statement technique [41] provided by MyBatis, which safely parameterize the input. However, some developers may mistakenly use `${input}` in the query which allows MyBatis to insert the user input directly, resulting in SQL injections.

A Real-world Case Study from lamp-boot. As shown in Figure 5a, the vulnerability arises from an insecure SQL statement concatenation in the `page` method in line 8. The vulnerability requires the fuzzer to construct a complex JSON payload whose fields must adhere to specific semantic constraints, as illustrated in Figure 5b. The input must be structured in such a way that it binds to the `PageParams` class, with the `model` field being an instance of `BaseEmployeePageQuery`. Crucially, to trigger the SQL injection vulnerability in line 12, the `scope` field within the `model` object must be exactly set to the string “1”, indicating the constant `BIND`.

The full version of WDFuzz successfully discovered this vulnerability, while Witcher failed to detect it. The primary reason lies in the fuzzer’s ability to accurately generate the required nested JSON structure while also satisfying the semantic constraints on the fields such as `email`, `state`, `userId`, and `orgIdList`, which must be an email address, a boolean, an integer, and a JSON list, respectively. Furthermore, WDFuzz also managed to extract the constraints for the `scope` field and pass the validation of the application, thereby exposing the SQL injection effectively.

5.5 RQ3: Ablation Studies

In our ablation study, we systematically evaluate the contribution of each module within WDFuzz by incrementally reintroducing them to assess their impact on performance. Our approach involves structuring the experiments around three core modules: the vulnerable path extraction, the semantic constraint extraction, and the hierarchical scheduling strategy. By isolating these modules, we aim to quantify their individual effects on the overall effectiveness of WDFuzz. The detailed configurations are listed below.

- WDFuzz$_b$ (*b̲aseline*): This setup applies CrawlerGo [3], an industrial web crawler, to find entries and parameters, with a fixed scheduling strategy that fuzzes every entry for 2 minutes, serving as the baseline of the ablation study.
- WDFuzz$_e$ (*e̲ntry extraction*): This configuration uses the statically extracted vulnerable web entries and parameters without structure and constraint information, and applies the fixed scheduling strategy in baseline, to assess the vulnerable path extraction module.
- WDFuzz$_c$ (*c̲onstraint extraction*): This variant incorporates the parameter structure and semantic constraint extraction module, as well as the fixed scheduling strategy to evaluate how the semantic constraint extraction module affects the vulnerability detection performance.
- WDFuzz (The full version): This full version of WDFuzz combines all the parameter structure, the constraint extraction, and the hierarchical scheduling strategy modules.

The results of our ablation study are summarized in Table 2. Each module contributes to a notable increase in recall rates for known vulnerabilities, with improvements ranging from 20% to 30%. A detailed analysis below reveals the underlying reasons for these efficiency gains.

Firstly, the implementation of static analysis for vulnerability entry point extraction (WDFuzz$_e$) significantly expands the attack surfaces compared to the crawler-based approach

Table 2: Ablation study results for WDFuzz. The names and versions of the commercial closed-source web applications are anonymized due to legal reasons. The two web applications with no known vulnerabilities are hided in this table.

#	Application	Version	Known Vuln.	WDFuzz$_b$			WDFuzz$_e$			WDFuzz$_c$			WDFuzz		
				Detected Vuln.	Recall	Avg. TTE/s	Detected Vuln.	Recall	Avg. TTE/s	Detected Vuln.	Recall	Avg. TTE/s	Detected Vuln.	Recall	Avg. TTE/s
1	jeecg-boot	3.2.0	5	0	0.0%	-	0	0.0%	-	2	40.0%	1380.0	5	100.0%	30.0
2	jeesite	1.2.6	3	0	0.0%	-	1	33.3%	729.0	1	33.3%	481.0	1	33.3%	3.0
3	jshERP	2.3	8	3	37.5%	928.7	3	37.5%	1040.3	7	87.5%	256.0	7	87.5%	78.3
4	MCMS	5.2.4	12	0	0.0%	-	2	16.7%	1140.5	3	25.0%	488.3	12	100.0%	34.3
5	RuoYi	4.5.1	11	7	63.6%	107.7	0	0.0%	-	11	100.0%	98.3	11	100.0%	120.0
6	SpringBlade	3.6.0	1	0	0.0%	-	1	100.0%	1.0	1	100.0%	1.0	1	100.0%	1.0
7	Halo	1.4.9	1	0	0.0%	-	1	100.0%	246.0	1	100.0%	259.0	1	100.0%	30.0
8	DreamerCMS	4.0.1	2	0	0.0%	-	0	0.0%	-	2	100.0%	48.5	2	100.0%	13.5
9	PublicCMS	4.0	6	0	0.0%	-	4	66.7%	180.3	4	66.7%	151.3	4	66.7%	44.0
12	Commercial-1	***	3	0	0.0%	-	0	0.0%	-	0	0.0%	-	3	100.0%	98.0
13	Commercial-2	***	1	0	0.0%	-	0	0.0%	-	0	0.0%	-	1	100.0%	120.0
14	Commercial-3	***	3	0	0.0%	-	3	100.0%	1120.7	3	100.0%	884.3	3	100.0%	233.3
15	WebGoat	2023.9	12	0	0.0%	-	12	100.0%	110.1	12	100.0%	9.8	12	100.0%	3.2
	Total		**68**	**10**	**14.7%**	**354.0**	**27**	**39.7%**	**436.4**	**47**	**69.1%**	**240.7**	**63**	**92.6%**	**60.6**

(WDFuzz$_b$), resulting in a 2.7 times increase in the number of discovered vulnerabilities. This increase in entry points and vulnerabilities inevitably leads to longer testing times, with a slight decrease in the number of vulnerabilities discovered per hour, which is an expected trade-off.

Building upon the extraction of entry points, we further enhanced our fuzzer's capabilities by incorporating parameter structure and semantic constraint extraction (WDFuzz$_c$). This modification led to an additional 74.07% increase in vulnerability discovery, while the overall TTE even experienced a slight reduction. Specifically, the rate of vulnerabilities detected per hour improved by 81.33%. The underlying reason for this efficiency is that with parameter structure and semantic constraint information, the fuzzer can generate more complex structures, such as JSON formats, as well as values that align with the conditions necessary for vulnerability exploitation. Consequently, the additional constraints facilitate the discovery of deeper, more complex vulnerabilities, while also accelerating the identification of shallow vulnerabilities.

Additionally, by integrating hierarchical scheduling, the full version of WDFuzz achieved a further 34.04% increase in vulnerabilities discovered, with a corresponding 66.24% decrease in TTE. The hourly discovery rate of vulnerabilities improved by 2.97 times. This enhancement is contributed by the hierarchical scheduling strategy, which prioritizes the allocation of energy to the easiest-to-trigger vulnerabilities in the first place. Once these vulnerabilities are successfully triggered, the scheduling algorithm reallocates resources to target more difficult vulnerabilities, thereby ensuring that these challenging cases receive more focused testing energy compared to the fixed scheduling approaches. As a result, the hierarchical scheduling strategy enables quicker exposure of simpler vulnerabilities while testing complex vulnerabilities with more resources, ultimately maximizing both the effectiveness and efficiency of vulnerability detection.

Interestingly, we observed some anomalies in the RuoYi application (Application 5). The baseline method WDFuzz$_b$ yielded a higher number of vulnerabilities compared to the entry point extraction method WDFuzz$_e$. This disparity arises because the crawler can capture a few nested structures prefilled in the front end, such as `params[dataScope]`, which the WDFuzz$_e$ method cannot generate due to the lack of structural information. However, it is important to note that despite the crawler's ability to occasionally uncover certain structures, its overall capability to discover the attack surface remains significantly inferior. Consequently, the number of vulnerabilities detected by the crawler-based approach is far fewer than those revealed by static analysis-based approaches. Besides, once we incorporated the ability to extract and generate complex structures and constraints, WDFuzz demonstrated significantly superior overall performance compared to the crawler-based approach, further emphasizing the effectiveness of our enhancements in vulnerability discovery.

6 Discussion

Generalization of WDFuzz. Although we implemented a prototype of WDFuzz in Java, our methodology can be easily adapted to other programming languages. For instance, when testing PHP web applications, one can simply utilize static analysis tools like PHPJoern [13] and instrumentation tools like Xdebug [8], thereby replacing the toolchain in our prototype to facilitate fuzzing of PHP applications.

Detection of High-order Vulnerabilities. WDFuzz currently do not have the capability to identify the high-order vulnerabilities which require sequentially triggering various web entries to exploit. Detecting source-to-sink flows that span across several entry points is generally a challenging problem. Therefore, we focus on vulnerabilities that can be triggered by a single request in this paper. This type of single-step vulnerability represents the vast majority of all the known vulnerabilities, making up 92.6% of our dataset. We believe that detecting high-order vulnerabilities is a great research topic and consider it an important direction for future research.

Limitations of Static Preparation Phase. Due to the inherent limitations of static analysis, the static preparation phase of WDFUZZ (see §4.1) would inevitably report some inaccurate results. For example, to achieve a satisfactory tradeoff between scalability and precision of semantic constraint extraction (see §4.1.3), WDFUZZ would simplify loops by considering only their single iterations. Hence, it may produce inaccurate results when analyzing validation functions involving complex loops. Besides, for the vulnerable path extraction (see §4.1.2), WDFUZZ would miss some vulnerable paths with complex dynamic language features (e.g., reflections and native interfaces in Java). However, the evaluation results show that the current version of WDFUZZ can successfully identify 63 of 68 benchmark vulnerabilities, indicating that the impact of erroneous results from static analysis is limited. In the future, WDFUZZ can be enhanced by incorporating more advanced static analysis techniques [29, 35, 48].

7 Related Work

Static Analysis. Static analysis techniques aim to identify potentially vulnerable paths from user-controlled inputs (sources) to security-sensitive operations (sinks) in the source code without executing the program [9–11, 14, 25, 33, 39, 43, 46]. To address the challenges posed by large and heterogeneous codebases in enterprise Java web applications, Wang et. al. proposed ANTaint [46], which employed lazy loading of library APIs to construct large-scale call graphs while dynamically transforming code to support various web frameworks. TChecker [33] further improved the static analysis approach with a context-sensitive inter-procedural taint analysis, constructing accurate call graphs based on type inference and incorporating a context selection algorithm to reduce overhead. However, static analysis is often plagued by high false positive rates, which lead to significant manual efforts to double-check the reports. Additionally, these techniques typically cannot generate actual PoCs, limiting their usability in the scenarios of vulnerability detection and mitigation.

Black-Box Scanning. Black-box scanning approaches analyze web applications from an external perspective without access to the source code [2, 5, 7, 12, 21–24]. These methods include several general-purpose vulnerability scanners, such as Burp Suite [2], Wapiti [7], and OWASP ZAP [5], which provide comprehensive vulnerability detection capabilities. In addition, some researches have focused on testing RESTful APIs and leveraged OpenAPI specifications to guide the testing [12, 22]. Another notable approach, Black Widow [24], established a web application navigation model and injected unique identifiers into inputs to detect cross-site scripting (XSS) vulnerabilities through inter-page dependencies. Additionally, ReScan [23] functioned as a middleware proxy between scanners and web applications, enhancing the scanning capabilities of existing black-box scanners by enabling functionalities like re-login and page relationship discovery. However, black-box techniques face several limitations, including low coverage rates that hinder the exploration of deeper vulnerabilities, and the reliance on complex front-end interactions to obtain feedback.

Grey-Box Fuzzing. Grey-box fuzzing represents a hybrid approach that combines aspects of both static and dynamic analysis [27, 28, 38, 42, 45, 49]. For instance, webFuzz [45] modified PHP files directly to obtain coverage feedback, guiding the detection of XSS vulnerabilities. CeFuzz [49] used paths from entry PHP files to known vulnerability locations as inputs and prioritized seeds with the most bypassed condition checks along those paths. Witcher [42] employed interpreter instrumentation to provide coverage feedback and enhance generalization across different web application languages, while utilizing error messages as bug oracles. Atropos [28] instrumented PHP comparison functions to obtain expected key-value pairs as feedback, while providing eight comprehensive bug oracles. Nevertheless, existing grey-box web fuzzers exhibit shortcomings, such as the very low coverage rates inherited from the use of black-box crawlers and incompatibility with web framework development patterns. Moreover, coverage-based feedback often leads to the exploration of numerous irrelevant paths, and the scheduling strategies are naive, lacking considerations for selecting optimal entry points to maximize vulnerability detection efficiency.

8 Conclusion

Fuzzing for web applications is a vital research area, while existing web fuzzers were limited in effectiveness and efficiency. In this paper, we introduce a novel web application fuzzer, WDFUZZ, that can effectively extract parameter structures and semantic constraints and employs a novel hierarchical scheduling strategy to prioritize the seeds. By applying WDFUZZ to real-world web applications, we find WDFUZZ outperforms state-of-the-art web fuzzers, discovering 3.2 times more vulnerabilities while reducing the time for vulnerability identification by 87.69%. To date, WDFUZZ has successfully identified 92 previously unknown vulnerabilities in real-world applications, with 19 CVE or CNVD ID assignments.

Acknowledgement

We thank the anonymous reviewers for the helpful comments and feedback. This work was supported in part by the National Natural Science Foundation of China (U2436207, 62172105, 62402116, 62202106, 62102093). Yuan Zhang and Min Yang are the corresponding authors. Yuan Zhang was supported in part by the Shanghai Pilot Program for Basic Research - Fudan University 21TQ1400100 (21TQ012). Min Yang is a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

Ethics Considerations

This work poses no ethical concerns. All testing activities were conducted within our locally set up offline environment, ensuring that there was no interaction with or impact on any real-world systems or user data. We have proactively reported all vulnerabilities we discovered and assisted developers in fixing these vulnerabilities. As a result, 19 vulnerability identifiers have been assigned as a confirmation for our efforts.

Open Science

In alignment with the open science policy, we are committed to fully following the conference's artifact evaluation guidelines. We release the artifact[9] including the source code of WDFuzz, the datasets and baselines used for evaluation in our research. This initiative aims to enhance the reproducibility and replicability of scientific findings, ensuring that our work can be verified and built upon by other researchers in the field.

References

[1] AFL documents - Understanding the status screen. https://github.com/google/AFL/blob/master/docs/status_screen.txt#L219.

[2] Burp Suite - Application Security Testing Software. https://portswigger.net/burp.

[3] CrawlerGo. https://github.com/Qianlitp/crawlergo.

[4] OWASP webgoat. https://owasp.org/www-project-webgoat/.

[5] Owasp zed attack proxy (zap). https://www.zaproxy.org/.

[6] Spring dominates the Java ecosystem with 60% using it for their main applications. https://snyk.io/blog/spring-dominates-the-java-ecosystem-with%2D60-using-it-for-their-main-applications/.

[7] Wapiti: a free and open-source web-application vulnerability scanner. https://wapiti-scanner.github.io/.

[8] Xdebug - Debugger and Profiler Tool for PHP. https://xdebug.org/.

[9] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. Static analysis of java enterprise applications: frameworks and caches, the elephants in the room. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 794–807, London, UK, June 2020.

[10] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM sigplan notices*, 49(6):259–269, June 2014.

[11] Steven Arzt, Siegfried Rasthofer, Robert Hahn, and Eric Bodden. Using targeted symbolic execution for reducing false-positives in dataflow analysis. In *Proc. ACM SIGPLAN International Workshop on State of the Art in Program Analysis (SOAP)*, pages 1–6, Portland, OR, USA, June 2015.

[12] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful REST API fuzzing. In *Proc. ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 748–758, Montreal, Canada, May 2019.

[13] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *Proc. IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 334–349, Paris, France, April 2017.

[14] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Active learning of points-to specifications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 678–692, Philadelphia, PA, USA, June 2018.

[15] Frank Benford. The law of anomalous numbers. *Proceedings of the American philosophical society*, 78(4):551–572, March 1938.

[16] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2329–2344, Dallas Texas USA, November 2017.

[17] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. A systematic review of fuzzing techniques. *Computers & Security*, 75:118–137, June 2018.

[18] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2095–2108, New York, NY, October 2018.

[9] https://zenodo.org/records/14718601

[19] Miao Chen, Tengfei Tu, Hua Zhang, Qiaoyan Wen, and Weihang Wang. Jasmine: A static analysis framework for spring core technologies. In *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1–13, Rochester, MI, USA, January 2022.

[20] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 332–343, Singapore, Singapore, September 2016.

[21] Davide Corradini, Michele Pasqua, and Mariano Ceccato. Automated black-box testing of mass assignment vulnerabilities in restful apis. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2553–2564, Melbourne, Australia, May 2023.

[22] Gelei Deng, Zhiyi Zhang, Yuekang Li, Yi Liu, Tianwei Zhang, Yang Liu, Guo Yu, and Dongjin Wang. NAUTILUS: Automated RESTful API Vulnerability Detection. In *Proc. USENIX Security Symposium*, pages 5593–5609, Anaheim, CA, August 2023.

[23] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. ReScan: A Middleware Framework for Realistic and Robust Black-box Web Application Scanning. In *Proc. Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, February 2023.

[24] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. Black Widow: Blackbox Data-driven Web Scanning. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, pages 1125–1142, Los Alamitos, California, May 2021.

[25] Pratik Fegade and Christian Wimmer. Scalable pointer analysis of data structures using semantic models. In *Proc. International Conference on Compiler Construction*, pages 39–50, New York, NY, United States, February 2020.

[26] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. LibAFL: A framework to build modular and reusable fuzzers. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1051–1065, Los Angeles, CA, USA, November 2022.

[27] François Gauthier, Behnaz Hassanshahi, Benjamin Selwyn-Smith, Trong Nhan Mai, Max Schlüter, and Micah Williams. Experience: model-based, feedback-driven, greybox web fuzzing with BackREST. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, Berlin, Germany, June 2022.

[28] Emre Güler, Sergej Schumilo, Moritz Schloegel, Nils Bars, Philipp Görz, Xinyi Xu, Cemal Kaygusuz, and Thorsten Holz. Atropos: Effective fuzzing of web applications for server-side vulnerabilities. In *Proc. USENIX Security Symposium*, Philadelphia, PA, August 2024.

[29] Minseok Jeon and Hakjoo Oh. Return of cfa: call-site sensitivity can be superior to object sensitivity even for object-oriented programs. In *Proc. ACM on Programming Languages (POPL)*, pages 1–29, New York, NY, January 2022.

[30] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proc. International Conference on Software Engineering (ICSE)*, pages 672–681, San Francisco, California, USA, May 2013.

[31] Rana Fouad Khalil. *Why Johnny Still Can't Pentest: A Comparative Analysis of Open-Source Black-box Web Vulnerability Scanners*. PhD thesis, Université d'Ottawa/University of Ottawa, 2018.

[32] Hongliang Liang, Yini Zhang, Yue Yu, Zhuosi Xie, and Lin Jiang. Sequence coverage directed greybox fuzzing. In *Proc. International Conference on Program Comprehension (ICPC)*, pages 249–259, Montreal, Quebec, Canada, May 2019.

[33] Changhua Luo, Penghui Li, and Wei Meng. Tchecker: Precise static inter-procedural analysis for detecting taint-style vulnerabilities in php applications. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2175–2188, Los Angeles, U.S.A., November 2022.

[34] Changhua Luo, Wei Meng, and Penghui Li. Select-Fuzz: Efficient directed fuzzing with selective path exploration. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, pages 2693–2707, San Francisco, CA, May 2023.

[35] Wenjie Ma, Shengyuan Yang, Tian Tan, Xiaoxing Ma, Chang Xu, and Yue Li. Context sensitivity without contexts: A cut-shortcut approach to fast and precise pointer analysis. In *Proc. ACM on Programming Languages (PLDI)*, pages 539–564, New York, NY, June 2023.

[36] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, October 2019.

[37] M Muralidharan, Keshav Balaji Babu, and G Sujatha. W3bnnr: An automated tool for information gathering,

vulnerability scanning, attacking and reporting for injection attacks on web application. In *Proc. Advanced Computing and Communication Technologies for High Performance Applications (ACCTHPA)*, pages 1–4, Cochin, Kerala, India, January 2023.

[38] Sebastian Neef, Lorenz Kleissner, and Jean-Pierre Seifert. What all the phuzz is about: A coverage-guided fuzzer for finding vulnerabilities in php web applications. In *Proc. ACM Asia Conference on Computer and Communications Security*, pages 1523–1538, Singapore, Singapore, July 2024.

[39] Mathias Romme Schwarz. *Design and analysis of web application frameworks*. PhD thesis, Datalogisk Institut, Aarhus Universitet, 2013.

[40] Tian Tan and Yue Li. Tai-e: A developer-friendly static analysis framework for Java by harnessing the good designs of classics. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 1093—1105, Seattle, WA, July 2023.

[41] Stephen Thomas, Laurie Williams, and Tao Xie. On automated prepared statement generation to remove sql injection vulnerabilities. *Information and Software technology*, 51(3):589–598, March 2009.

[42] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect SQL and command injection vulnerabilities. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, pages 2658–2675, San Francisco, CA, May 2023.

[43] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. *ACM Sigplan Notices*, 44(6):87–97, June 2009.

[44] Akshay Utture, Shuyang Liu, Christian Gram Kalhauge, and Jens Palsberg. Striking a balance: pruning false-positives from static call graphs. In *Proc. IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 2043–2055, Pittsburgh, Pennsylvania, May 2022.

[45] Orpheas van Rooij, Marcos Antonios Charalambous, Demetris Kaizer, Michalis Papaevripides, and Elias Athanasopoulos. webFuzz: Grey-Box fuzzing for web applications. In *Computer Security - ESORICS 2021*, pages 152–172, Darmstadt, Germany, October 2021.

[46] Jie Wang, Yunguang Wu, Gang Zhou, Yiming Yu, Zhenyu Guo, and Yingfei Xiong. Scaling static taint analysis to industrial SOA applications: A case study at alibaba. In *Proc. ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1477–1486, Virtual Event USA, November 2020.

[47] Gebrehiwet B Welearegai, Max Schlueter, and Christian Hammer. Static security evaluation of an industrial web application. In *Proc. ACM/SIGAPP Symposium on Applied Computing (SAC)*, pages 1952–1961, Limassol, Cyprus, April 2019.

[48] Yifan Zhang, Yuanfeng Shi, and Xin Zhang. Learning abstraction selection for bayesian program analysis. In *Proc. ACM on Programming Languages (OOPSLA)*, pages 954–982, New York, NY, April 2024.

[49] Jiazhen Zhao, Yuliang Lu, Kailong Zhu, Zehan Chen, and Hui Huang. Cefuzz: An directed fuzzing framework for php rce vulnerability. *Electronics*, 11(5):758, March 2022.

A Sink List

The sinks WDFUZZ used to find common security-sensitive operations are listed in Table 3, with all overloaded methods included.

Table 3: Sink Types and Corresponding Sinks

Command Injection
`java.lang.ProcessBuilder.<init>`
`java.lang.ProcessBuilder.command`
`java.lang.ProcessBuilder.start`
`java.lang.ProcessBuilder.startPipeline`
`java.lang.Runtime.exec`
SQL Injection
`java.sql.DriverManager.getConnection`
`java.sql.PreparedStatement.executeQuery`
`java.sql.Statement.execute`
`java.sql.Statement.executeUpdate`
`java.sql.Statement.executeQuery`
`org.hibernate.impl.SessionImpl.createQuery`
`org.hibernate.Query.list`
`org.hibernate.Session.createQuery`
`org.hibernate.Session.createSQLQuery`
SSRF
`java.awt.Desktop.browse`
`java.awt.Desktop.mail`
`java.awt.Desktop.open`
`java.net.URI.create`
`java.net.URL.getContent`
`java.net.URL.openConnection`
`java.net.URL.openStream`
`java.net.URLClassLoader.findClass`
`java.net.URLClassLoader.findResource`
`java.net.URLClassLoader.getResourceAsStream`
`java.net.URLClassLoader.newInstance`
Arbitrary File Reading
`java.io.BufferedReader.read`
`java.io.FileInputStream.read`
`java.io.FileInputStream.<init>`
Arbitrary File Writing
`java.io.BufferedWriter.write`
`java.io.File.createNewFile`
`java.io.File.createTempFile`
`java.io.FileOutputStream.write`
`java.io.FileOutputStream.<init>`
`java.nio.channels.AsynchronousByteChannel.write`
`java.nio.channels.AsynchronousFileChannel.write`
`java.nio.channels.DatagramChannel.write`
`java.nio.channels.FileChannel.write`
`java.nio.channels.GatheringByteChannel.write`
`java.nio.channels.SeekableByteChannel.write`
`java.nio.channels.SocketChannel.write`
`java.nio.channels.WritableByteChannel.write`
`java.nio.file.Files.write`
`java.nio.file.FileSystems.newFileSystem`

B Fuzzing Loop Algorithm

Algorithm 1 Fuzzing Loop

```
Input: Web Application $A$
Output: Detected Vulnerabilities $V$
Initialize $V \leftarrow \emptyset$
Initialize distances $D \leftarrow \{$Initial distances$\}$
Initialize $Corpus \leftarrow \{$Initial seeds$\}$
Initialize time $t \leftarrow 0$
Initialize $start\_time \leftarrow$ CurrentTime()
while $t <$ TIME_BUDGET do
    $S_e \leftarrow$ EntryScoring$(A, D)$
    $e \leftarrow$ SampleEntryPoint$(S_e)$
    $S_l \leftarrow$ SinkLocationScoring$(e, A, D)$
    $l \leftarrow$ SampleSinkLocation$(S_v)$
    $s \leftarrow$ NextSeed$(Corpus_{e,v})$
    $mutation\_count \leftarrow$ AFLGoScore$(s)$
    for $i \leftarrow 1$ to $mutation\_count$ do
        $s' \leftarrow$ MutateSeed$(s)$
        Execute $s'$ against $A$
        if vulnerability detected then
            $V \leftarrow V \cup \{$detected vulnerability$\}$
        end if
        if distance of $e$ and $v$ updated to $d$ then
            $D_{e,v} \leftarrow d$
            $Corpus_{e,v} \leftarrow Corpus_{e,v} \cup \{s'\}$
        end if
    end for
    $t \leftarrow$ CurrentTime() $-$ $start\_time$
end while
return $V$
```