



USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

XSSky: Detecting XSS Vulnerabilities through Local Path-Persistent Fuzzing

*Youkun Shi, Fudan University and The Hong Kong Polytechnic University;
Yuan Zhang, Tianhao Bai, Feng Xue, Jiarun Dai, Fengyu Liu, and Lei Zhang,
Fudan University; Xiapu Luo, The Hong Kong Polytechnic University;
Min Yang, Fudan University*

<https://www.usenix.org/conference/usenixsecurity25/presentation/shi-youkun>

**This paper is included in the Proceedings of the
34th USENIX Security Symposium.**

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the
34th USENIX Security Symposium is sponsored by USENIX.

XSSky: Detecting XSS Vulnerabilities through Local Path-Persistent Fuzzing

Younkun Shi^{†‡}, Yuan Zhang[†], Tianhao Bai[†], Feng Xue[†], Jiarun Dai[†], Fengyu Liu[†], Lei Zhang[†],
Xiapu Luo[‡], Min Yang[†]

[†]Fudan University [‡]The Hong Kong Polytechnic University

Abstract

The Cross-Site Scripting (XSS) vulnerability is one of the most prevalent security issues in PHP web applications. To detect XSS vulnerabilities, existing dynamic techniques are commonly hindered by insufficient code exploration capabilities and non-trivial execution environment setup. Comparably, static techniques offer more flexible detection of target code by identifying vulnerable source-sink paths. However, these paths would probably be guarded by custom sanitizers (i.e., implemented to filter malicious inputs). Without establishing reliable sanitizer modeling and analysis techniques, existing work can hardly achieve satisfactory effectiveness.

In light of this, we propose a static sanitizer-tolerant XSS detector, named XSSky. Our key insight is that concrete malicious inputs, which evade sanitizers and trigger XSS vulnerabilities, serve as strong proof of a vulnerability's existence. Based on this idea, XSSky attempts to deterministically curate malicious inputs for potentially vulnerable source-sink paths using a path-persistent fuzzing strategy. Specifically, XSSky first converts each given source-sink path into locally executable Programs Under Test (PUTs). Then it uses XSS-oriented exploit primitives and PHP interpreter feedback to generate malicious inputs to efficiently confirm the existence of vulnerabilities. Evaluation results show that XSSky successfully detected 60 previously unknown XSS vulnerabilities (including 31 caused by sanitizer evasion) across 20 popular PHP web applications. Compared with several existing state-of-the-art techniques, XSSky achieved a precision improvement of 11.48%~642.49% and a recall improvement of 87.51%~172.70%. Furthermore, XSSky identified 18 unique vulnerabilities that none of the baselines could detect.

1 Introduction

To date, PHP web applications have become an integral part of our daily lives. According to statistics [12], more than 75.1% of online websites are developed using PHP, with prominent examples including Wikipedia [13], WordPress [14],

Baidu [3], and Tumblr [11]. With their widespread adoption and enduring popularity, PHP web applications also face increased exposure to security vulnerabilities.

Cross-Site Scripting vulnerabilities (a.k.a., XSS) are among the most prevalent in PHP web applications. Analytics indicate that over the past decade, 52.24% of web vulnerabilities have been classified as XSS vulnerabilities [16], and 86% of PHP web applications contain at least one such vulnerability [2]. These vulnerabilities enable attackers to inject malicious scripts into web pages, leading to severe security breaches such as theft of sensitive information or user session hijacking, posing a significant threat to application security.

To proactively detect XSS vulnerabilities, substantial efforts have been made to develop security testing techniques, which can be broadly divided into two main lines: dynamic analysis [23–27, 43, 48] and static analysis [20, 22, 33, 35, 37, 41]. Dynamic analysis is renowned for its high precision in providing proof of concept (PoC) for vulnerability detection; however, it is often limited by false negatives due to inadequate code coverage. In contrast, static analysis techniques can flexibly scan the entire application's source code, allowing for a thorough examination of each sink within the target application. However, we have observed that these techniques exhibit certain fragility in their effectiveness when analyzing code paths that include sanitizers (i.e., malicious input filters).

To be more specific, existing static approaches [20, 22, 33, 41] heavily rely on expertise-dependent modeling to identify the sanitizers on source-sink paths, and hastily assume that the presence of a sanitizer indicates the non-existence of vulnerabilities. However, this methodology has two significant risks: ❶ The expertise-dependent modeling can hardly recognize diverse custom sanitizers, eventually causing false positives of XSS vulnerability detection. ❷ The presence of a sanitizer does not necessarily guarantee security, considering the existence of flawed sanitizers (e.g., design oversights, implementation flaws, or misuse [35]), consequently resulting in false negatives of XSS vulnerability detection. Faced with this issue, some existing techniques [18, 35, 37] attempt to formally verify the correctness of sanitizers using SMT solvers.

However, due to the inherent limitations of SMT solving (e.g., memory explosion when analyzing complex logic [35]), these techniques have quite limited feasibility and applicability.

Different from existing works, we conceived the idea of integrating dynamic analysis capabilities into static detectors to analyze sanitizer-exist vulnerable paths. To put it straightforwardly, for a potentially vulnerable source-sink path, we aim to generate path-specific PoC to convincingly confirm the vulnerability's existence. Here, an appealing solution appears to migrate path-guided fuzzing techniques [31, 44, 49] (i.e., to guide the code exploration towards given paths) designed for binaries/systems/compilers to PHP web applications. However, these techniques still inevitably explore code unrelated to the target paths (e.g., code exploration before reaching a given path) and require fully built applications for the fuzzing environment, demanding significant manual efforts.

Therefore, we propose a novel path-persistent fuzzing strategy to efficiently confirm whether a given source-sink path is vulnerable. Specifically, for a source-sink path reported by a static tool, we convert it into local executable code snippets, referred to as PUT (*Program Under Testing*). We then employ fuzzing techniques to verify the security of this path. This path-persistent approach not only prevents the fuzzing process from exploring code unrelated to the target but also improves the throughput of the fuzzing process.

While this novel approach offers significant advantages, it also presents two key challenges that must be addressed to ensure its successful implementation. One of the primary challenges involves the sound conversion of source-sink paths reported by static tools into locally executable PUTs, as these paths often contain numerous undefined symbols. If not properly addressed, this issue can affect the executability of fuzzing. Another major challenge is determining how to effectively and efficiently detect XSS vulnerabilities by fuzzing these PUTs. We have found that existing dynamic testing techniques are limited by two factors. First, they lack fine-grained consideration for XSS exploitation construction (e.g., sink context, as clarified in §2.2), which leads to attempts with invalid test cases. Second, they lack an understanding of sanitizers, making it difficult to mutate characters intercepted by sanitizers purposefully, and struggle to assess the effectiveness of current mutation strategies.

In this work, we focus on reflected server-side XSS vulnerability detection and propose XSSky to address the above challenges. Regarding PUT conversion, for a given source-sink path, XSSky first performs a bottom-up data flow analysis on undefined variables to recover their definitions. During this process, for variables lacking a complete def-use chain, XSSky follows existing techniques [19, 40] to convert them into fuzzer-controllable temporary variables. Then, XSSky conducts undeclared function localization to iteratively locate the definitions of the function calls within the code path. In terms of fuzzing, XSSky addresses the issue through the following two aspects. On one hand, we have compiled a com-

prehensive set of exploit primitives (i.e., eight sink-context-aware exploit grammars) to guide the fuzzer in efficiently generating test cases. On the other hand, we propose a novel fuzzing scheme based on the feedback of the PHP interpreter. By hooking string comparison and modification functions and using the hooked values as feedback, the fuzzer can determine which characters in the test case are blocked by the sanitizer and whether the mutated test case bypasses the sanitizer.

We evaluated XSSky on 20 popular, real-world PHP web applications. The evaluation results show that XSSky successfully converted 6,997 / 7,005 source-sink paths reported by TChecker [41] into locally executable PUTs. Through comprehensive fuzzing tests, XSSky successfully discovered 60 critical XSS vulnerabilities, notably including 31 instances where sanitizers were already deployed. We then compared the effectiveness of XSSky against several existing techniques. The results confirmed that XSSky achieved a precision improvement of 11.48%~642.49% and a recall improvement of 87.51%~172.70% in XSS vulnerability detection. Moreover, XSSky not only detected all vulnerabilities found by baselines but also discovered 18 additional vulnerabilities that none of the baselines could detect, highlighting its advantage in advancing static techniques with a novel fuzzing scheme.

Contributions. The contributions of this paper are as follows:

- We present a novel path-persistent and interpreter-feedback fuzzing strategy to efficiently confirm whether a given source-sink path is vulnerable.
- Our experiments against real-world popular PHP web applications show XSSky is accurate and effective against XSS vulnerabilities, contributing to the discovery of 60 previously unknown XSS vulnerabilities, with 18 unique ones that none of the compared baselines could detect.
- We compare XSSky with several SOTA techniques, and the results demonstrate that XSSky outperforms the baselines by improving precision by 11.48%~642.49% and recall by 87.51%~172.70%. Moreover, XSSky identified 18 unique vulnerabilities that none of the baselines could detect.

2 Background

In this section, for ease of understanding our ideas and the techniques proposed later, we first introduce two key aspects of XSS vulnerability exploitation (§2.2). Then, we provide an overview of sanitizers (§2.3), followed by a discussion on the pros and cons of existing techniques (§2.4).

2.1 XSS Vulnerability Types

In this part, we introduce the two main types of XSS vulnerabilities: client-side and server-side XSS.

Client-side XSS occurs when untrusted data is processed by client-side JavaScript without proper sanitization. This can lead to malicious behaviors in the user's browser, such as manipulating the Document Object Model (DOM) or abusing

JavaScript APIs like `eval()`. For instance, if a webpage insecurely processes data retrieved from `location.hash`, an attacker could craft a URL like `abc.com#` to trigger execution. **Server-side XSS** arises when untrusted user input is improperly sanitized by the server-side before being embedded in web content. The malicious payload is delivered through server-handled sources such as HTTP request parameters, cookies, or database entries. This kind of vulnerability includes stored XSS (payload persisted in server databases) and reflected XSS (payload immediately echoed in server responses).

This paper focuses on reflected server-side XSS vulnerabilities originating from insecure PHP code implementations (stored server-side XSS detection can be extended and discussed in §6). In contrast, client-side XSS vulnerabilities, which arise from vulnerable JavaScript code, are beyond the scope of this paper.

2.2 XSS Vulnerability Exploitation

In this part, we introduce two key aspects of XSS vulnerability exploitation: **commonly used exploit components (§2.2.1)** and **sink contexts (§2.2.2)**.

2.2.1 Common components of XSS exploits

Based on prior studies [25, 35, 39], the exploits of XSS vulnerability consist of the following three key components.

Terminator Component is used to escape user input from the current syntactical structure, paving the way for subsequently injecting JavaScript-embedded code or directly constructing malicious JavaScript code.

JavaScript Embedded Component is utilized to create an executable JavaScript context within HTML code, enabling the execution of malicious JavaScript code injected by attackers. It can be divided into the following three types.

- *Protocols*: involves using URI schemes like "javascript" protocol within HTML hyperlink attribute values to execute JavaScript code.
- *HTML event handler attributes*: includes numerous attributes (e.g., "onclick") that execute JavaScript in response to user interactions or page events.
- *Inline JavaScript tags*: entails directly placing JavaScript code within "<script>" tags or embedding it within HTML elements.

JavaScript Code Component refers to the malicious code intended to be executed by the browser (e.g., steal user cookies). A common method to demonstrate the presence of an XSS vulnerability is to trigger a popup window using `alert("XSS")`.

2.2.2 Sink contexts

As a form of code injection, constructing an XSS exploit using the aforementioned components also depends on the

syntactical context in which the sink is used (a.k.a., sink contexts). According to OWASP [7], we can categorize the sink contexts into four main types, as shown in Figure 1.

```
1 // HTML Context
2 echo $input;
3 // URL Context
4 echo '<a href="'. $input. '></a>';
5 // HTML Attributes Context
6 echo "<input value='". $input. "'>";
7 // JavaScript Context
8 echo "<input onclick='". $input. "'>"; // type 1
9 echo "<script>". $input. "</script>"; // type 2
```

Figure 1: Examples for each type of sink context.

HTML Context. In this context, user input is directly embedded within the HTML content of a web page. Attackers may exploit this by injecting malicious HTML or script tags, such as `<script>alert('XSS')</script>`.

URL Context. In the URL context, user input is embedded within HTML hyperlink attribute values, often seen in `<a href="` or `...";
3 // Exploit: javascript:alert(1)
```

---

Figure 2: An example of XSS vulnerabilities caused by the misuse of the PHP built-in sanitizer `htmlspecialchars()`.

## 2.4 Existing Work for XSS Detection

To date, various techniques have been proposed to detect XSS vulnerabilities, which can be broadly categorized into two main lines: dynamic analysis [23–27, 43, 48] and static analysis [20, 22, 33, 35, 37, 41]. Generally, dynamic approaches usually employ crawlers to identify requests that might trigger XSS vulnerabilities and use bug oracles to confirm these vulnerabilities. While these approaches provide high precision by providing Proof of Concept (PoC) for the vulnerabilities detected, they suffer from limited code coverage, which hinders a thorough analysis of all potential sinks in the target application. For instance, the state-of-the-art technique, ReScan [24], achieves coverage of less than 50% of the code lines in the widely used PHP web application WordPress [14].

In contrast, static approaches offer an advantage in this regard. By scanning the entire application’s source code, these techniques can comprehensively analyze each sink within the target application. However, these approaches struggle with accurately analyzing source-sink paths that include security sanitizers. Specifically, the mainstream design [20, 22, 33, 41] often assumes that if a predefined sanitizer function (e.g., `htmlspecialchars()`) appears on the source-sink path, the path is deemed secure. However, this design faces two major challenges. First, due to the diverse implementations of custom sanitizers, relying solely on expert modeling may fail to accurately detect them, resulting in false positives from vulnerability detection. Second, the mere presence of a sanitizer on a path does not ensure path security, as sanitizers themselves may be vulnerable, potentially leading to false negatives in detecting vulnerabilities, as discussed in §2.3. Although some recent methods [18, 35, 37] attempt to verify the correctness of sanitizers by using SMT solvers, they require extensive modeling by security experts, particularly when it comes to modeling diverse features of regular expressions.

Furthermore, these methods often suffer from issues such as memory explosion and solver failures due to the complex business logic intertwined with sanitizers, which limits their feasibility and applicability.

## 3 Design Overview

In this section, we will first present our high-level idea (§3.1), then discuss the challenges encountered in achieving this idea (§3.2) along with our key insights for addressing them (§3.3). Finally, we will introduce the workflow of our proposed approach using a real-world running example (§3.4).

### 3.1 Our High-Level Idea

Compared to purely dynamic analysis, we favor static techniques for their satisfactory comprehensiveness in code analysis. However, as detailed in §2.4, static analysis often faces challenges with high false positive and false negative rates, primarily due to under-identification or evasion of sanitizers. To address these issues while preserving the benefits of static analysis, our key idea is to employ fuzzing to directly verify source-sink paths reported by static tools, thereby providing path-specific Proof of Concepts to confirm vulnerabilities.

**Advantages.** Overall, our integrated approach offers several benefits. Unlike purely dynamic analysis, it utilizes the comprehensive coverage provided by static analysis, thus avoiding the limitations associated with unexplored code. Moreover, it also enhances purely static analysis by removing the necessity for intensive modeling and hastily made assumptions required for analyzing sanitizers. By directly verifying code paths with fuzzing techniques, our idea can mitigate false positives from the under-identification sanitizers and false negatives from the sanitizer evasion. This ultimately improves the overall effectiveness of vulnerability detection.

**Design Choice.** While integrating fuzzing techniques into static tools presents an appealing solution, achieving satisfactory performance is still challenging. Our primary goal is to verify the potentially vulnerable paths reported by static tools without unnecessary code exploration. However, traditional fuzzing methods, or even directed fuzzing, often explore irrelevant paths, leading to unacceptable performance overhead. Specifically, directed fuzzing could be limited by the following factors: (1) extra human efforts are required to make the entire application code run, while security testing only focuses on a small part of it; (2) it inevitably delves into irrelevant paths while attempting to reach targets; (3) it may encounter difficulties when dealing with complex web behaviors, such as exploring web interactions that need to be triggered in sequence on the client side [29, 46].

To address these issues, we propose a novel path-persistent fuzzing strategy to efficiently generate PoC for a given source-sink path. Specifically, we can convert the source-sink path reported by a static tool into local executable code snippets,

referred to as PUT (*Program Under Testing*). Then, we can employ fuzzing techniques to verify the security of this path. This path-persistent approach not only restricts the fuzzing process to the relevant code, but also significantly boosts the fuzzing throughput by eliminating the need to uphold the preliminary requirements for reaching the path.

## 3.2 Challenges

Following the aforementioned high-level idea, we need to address the following two key challenges:

**Challenge I:** *How to soundly convert the paths reported by static tools into executable PUTs?* Existing static tools usually focus on the data-flow dependency of sink parameters on sources. As a result, the source-sink paths they report often contain some undefined symbols (i.e., variables and function calls), which hinder executability. Therefore, initializing these undefined symbols becomes key to the conversion of the PUTs, but this is not easy. Specifically, two main issues need to be addressed: (1) the undefined variables may introduced through file inclusion or be assigned through runtime operation (e.g., database queries), making them difficult to initialize during local execution of the PUTs; (2) the undeclared function calls might be object function invocations, making it difficult to accurately find their call targets when the correct runtime object type cannot be determined.

**Challenge II:** *How to effectively and efficiently detect the XSS vulnerabilities by fuzzing PUTs?* Despite numerous efforts to identify XSS vulnerabilities using dynamic testing techniques [23–26, 43, 48], we observed that these approaches are less effective in this regard and remain limited in the following two aspects: (1) *sink context-insensitivity*. XSS vulnerabilities are essentially a form of HTML and JavaScript code injection. Therefore, during fuzzing, test case generation must be sensitive to the context of the sink, ensuring that the injected code can seamlessly integrate into the existing syntax structure. Overlooking the sink context can lead to a large number of ineffective attempts, and even lead to missing vulnerabilities. (2) *insufficient feedback mechanism*. The mutation strategy is vital in fuzzing as it determines which components to mutate and how. Current techniques either traverse predefined exploits [23, 24, 26, 43], using random mutation methods [48], or employing genetic algorithms [25]. However, they often lack a robust feedback mechanism. This absence hinders the understanding of why current test cases are blocked by sanitizers and what mutations should be applied, thereby inhibiting the ability to perform targeted and effective mutations.

## 3.3 Key Insights

To tackle the aforementioned challenges, our approach is built upon two key insights.

**Key Insight I:** *Adaptable undefined symbols initialization strategy*. During the conversion of code paths into PUTs, our

approach involves relocating undefined symbols by traversing the call graph (CG) or program dependency graph (PDG) and integrating their definition-related code into paths identified by static tools. However, externally-declared and dynamically-assigned variables, as well as object function invocations with indeterminate object types, are challenging to address solely through static-analysis-based graph traversal. To largely ensure the executability and eliminate false negatives of vulnerability detection, we propose a trade-off yet adaptable strategy for these variables and functions. For undefined variables, we replace them with fuzzer-controllable temporary variables, and assign the correct variable type and random variable values during runtime. For object function invocations with indeterminate object types, we adopt an over-approximation approach which collects all potential call targets and creates a PUT for each of them. If an XSS vulnerability is triggered in any PUT, the path is considered vulnerable.

**Key Insight II:** *Sink context-aware and interpreter feedback-guide fuzzing scheme*. As discussed in §2.2, our key observation hints that, XSS vulnerabilities with different sink contexts have distinct yet stable exploit primitives. This observation allows us to design context-sensitive exploit grammars that limit unnecessary testing space. Specifically, for sinks with different contexts, we tailor combinations of XSS exploit components to guide the fuzzer in generating test cases that match the syntax structure of the sink context. Regarding the feedback mechanism, we aim to collect essential feedback from the PHP interpreter to guide the fuzzer in iterating test case mutations and confirming vulnerabilities. This includes collecting sanitizer-related information, such as line numbers of each hooked sanitizer, input values, matched values, and return values. Specifically, the feedback serves two main purposes: (1) it enables the fuzzer to identify which characters in the generated test case were flagged by the sanitizer, facilitating more targeted mutations; (2) it helps the fuzzer in determining the effectiveness of the current mutation strategy by analyzing whether more sanitizers have been bypassed, thereby aiding in the choice of mutation strategies for future iterations. Notably, we are not the first to leverage interpreter feedback to guide PHP fuzzing. However, existing works [28, 42, 47] primarily use it to enhance code exploration capabilities, whereas our approach uses it to increase code exploitation capabilities.

## 3.4 Running Example

Let's use a real-world example to further illustrate our ideas. This example is selected for its multi-patch lifecycle. The vulnerability was first disclosed as CVE-2018-19993. After being patched, an evasion occurred again, and it was then disclosed as CVE-2020-14475. Subsequently, it was evaded once more and was finally fully fixed on June 13, 2022. This showcases the error-prone nature of XSS sanitizers and motivates our work. Figure 3 presents the workflow of our approach

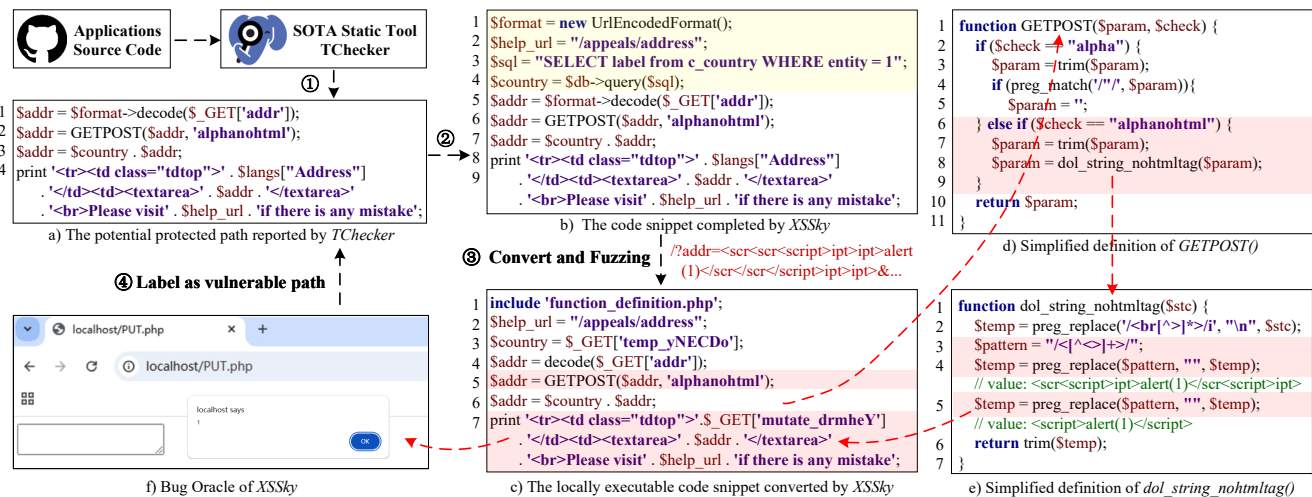


Figure 3: A running example of XSS vulnerability found in Dolibarr (5.5k stars); Fully fixed on June 13, 2022.

(named XSSky) for detecting an XSS vulnerability in Dolibarr. The process involves two key stages.

**Pre-processing: Source-Sink Path Location.** First, XSSky employs the state-of-the-art static detector TChecker [41]) to analyze the target application. Figure 3 (a) shows a source-sink path reported by TChecker. In this path, there is a data flow dependency between the user-controlled source (i.e., `$GET['addr']` in line 1) and the parameter of XSS sink (i.e., `$addr` of `print` in line 4).

**Stage I: PUT Conversion.** Second, XSSky converts this code path into a locally executable PUT. Following our key insights discussed in §3.3, XSSky begins by initializing the undefined variables. It performs a bottom-up data flow analysis on these variables, tracing back until it reaches constants, super-global variables, or points where no further data flow edges exist. Figure 3 (b) presents the original path refined by XSSky after performing data-flow backward analysis. The sections highlighted with a light yellow background (i.e., lines 1-4) indicate the definitions relocated by XSSky. Then, XSSky converts variables lacking def-use relationships in the PDG into temporary variables (e.g., line 7 in Figure 3 (c)). Finally, XSSky locates and saves the definition of callee functions in a file named "function\_definition.php", and then includes it in the source-sink path via file inclusion (line 1 in Figure 3 (c)).

**Stage II: PUT Fuzzing.** Third, XSSky leverages fuzzing techniques to confirm vulnerabilities in the PUT. The vulnerability arises from developer oversight in the implementation of the sanitizer. Specifically, the developers implemented the sanitizing logic within the function `dol_string_nohtmltag()`, as shown in Figure 3 (e). The developers intended to filter out the dangerous characters "<>", but due to oversight, they only accounted for a limited number of encoding bypass scenarios, thereby resulting in a vulnerability caused by triple-encoding bypasses. When a malicious input, e.g., "<sc<scr<script>ipt>ript>alert(1)</sc</scr</script>ipt>ript>"

is provided, it becomes "<script>alert(1)</script>" after passing through the sanitizer function `preg_replace()` twice, thus leading to an XSS vulnerability.

The Fuzzer engine of XSSky detects the vulnerability through the following steps: ① XSSky analyzed the sink context and identified it as HTML context. ② Based on the priority scheduling algorithm, XSSky first selected `<Inline,JS>` as the exploit grammar and generated the corresponding initial test case "<script>alert(1)</script>". ③ Through feedback, XSSky determined that the inline component "<script>" in the test case was detected by the sanitizer, allowing it to focus on mutating this component. ④ By attempting a series of mutation strategies for the inline component, XSSky discovered that using a double-encoding strategy allows for passing through more sanitizers (triggering feedback from both line 4 and line 5). Consequently, it chose to continue with this strategy, generating a triple-encoding test case. ⑤ Through validation with a bug oracle (as shown in Figure 3 (f)), XSSky confirmed the vulnerability.

## 4 Design of XSSky

In this section, we present the design details of XSSky. As previously clarified in §3.4, XSSky relies on existing static analyzers to identify source-sink paths of potential XSS vulnerabilities, and we do not claim contributions at this point. Currently, we build XSSky on top of TChecker [41], which is the state-of-the-art static detector for XSS vulnerabilities. With the provided source-sink paths in hand, XSSky leverages two key modules to confirm the XSS vulnerability implied in this path: ① *PUT Conversion Module* (§4.1) converts the given source-sink paths into locally executable PUTs. ② *PUT Fuzzing Module* (§4.2) leverages fuzzing techniques to confirm vulnerabilities on these PUTs.

## 4.1 PUT Conversion Module

The source-sink paths reported by existing static tools inevitably contain undefined symbols. Hence, to ensure that the path-related code snippets are locally executable, the *PUT Conversion Module* introduces two types of path-related code enhancement, respectively for addressing undefined variables (§4.1.1) and undeclared functions (§4.1.2).

### 4.1.1 Undefined Variable Initialization

**General Procedure.** In general, for a given source-sink path, XSSky traverses the code property graph (CPG) of the target application to identify the def-use chain for each undefined variable, and accordingly merges these definition-related code pieces into the original source-sink path. Specifically, XSSky performs a bottom-up data flow analysis on each undefined variable to trace backward along its def-use chain. The tracing is terminated until it encounters constants, and super-global variables (e.g., `$_GET`). After that, XSSky sequentially merges the assignment statements on the def-use chain into the beginning of the original source-sink path for variable initialization. **Handling Variable with Incomplete Def-Use Chain.** Notably, as thoroughly discussed in §3.2, complex PHP language features or dynamic assignments make it challenging [21, 30, 41] for static analysis frameworks to construct a complete def-use chain for all variables, thereby hindering the variable initialization. To sum up, there mainly exist two situations of incomplete def-use edges, respectively arising when handling the following two types of variables: ❶ *externally-declared variables*, i.e., variables introduced through file inclusion; ❷ *dynamically-assigned variables*, e.g., variables assigned through runtime database queries, file manipulation.

Here, faced with the infeasibility of static analysis, we propose a trade-off yet adaptable initialization strategy, to largely ensure the executability of generated PUTs. That is, directly converting these undefined variables (i.e., variables with incomplete def-use chains) into temporary variables that can be controlled by the fuzzer, and randomly assign values to them during runtime. Note that, for achieving local fuzzing [19, 40] of specific code pieces, it has become a common practice to replace undefined variables with fuzzer-controllable variables. The key difference lies in that, existing works commonly focus on C/C++ programs rather than PHP programs. Comparably, PHP is a weakly typed language, which means that for each undefined variable, not only the value assignment but also the variable type is undetermined. To tackle this issue, we would timely correct the variable type of each fuzzer-controllable variable during runtime based on interpreter feedback (i.e., the PHP interpreter would report error messages once encountering incorrect variable types). Technically, in our work, these temporary variables are constructed using the super-global variable `"$_GET"` with a unique key, which consists of the prefix `"temp_"` followed by six random case-insensitive characters.

For ease of understanding, taking Figure 3 (c) as an example, the undefined variable `$langs["Address"]` is converted into `$_GET['temp_drmheY']` (line 7).

### 4.1.2 Undeclared Function Localization

**General Procedure.** Similar to the procedure of undefined variable initialization (see §4.1.1), for each invoked function call on the source-sink path, XSSky identifies the call site on CPG (i.e., a call site node) and then locate the invoked function by querying the CPG (i.e., a function node that is connected to the call site node through a call edge). Then, based on the file and starting line information recorded at the function node, XSSky can accordingly extract the function implementation from the codespace of the target application. Eventually, XSSky incorporates these function-definition-related code pieces into the original source-sink path. It is important to note that the newly integrated function may also introduce additional undeclared function calls, XSSky iteratively locates and integrates these function definitions until no more undeclared function calls exist. When implementing the prototype of XSSky, for ease of maintenance and management of PUT code, we save function-definition-related code pieces in a file named `function_definition.php` within the same directory as the code path file, and incorporate the function definitions into the original source-sink path via file inclusion (e.g., line 1 in Figure 3 (c)).

**Handling Object Function Invocation with Indeterminate Object Type.** Nonetheless, as discussed in §3.2, considering the existence of object function invocations (i.e., to invoke a function that is defined within a class), it remains an open challenge [20, 21, 41] to construct complete and accurate call graphs for PHP applications. Consequently, without the capability to determine the correct runtime object type of the target object function invocation, it is quite difficult to achieve precise function definition localization and PUT conversion.

Here, we propose an over-approximate approach to tackle this issue, and the key rationale behind is that vulnerability detectors are more tolerant of false positives than false negatives. To be specific, given an object function invocation with an indeterminate runtime type, XSSky constructs the candidate set of invoked functions by collecting all object functions that share the same function signature (i.e., function name and number of parameters). With this candidate set in hand, XSSky constructs multiple PUTs by respectively and independently integrating each candidate function definition into the original source-sink path. In such a manner, once any of the constructed PUTs is found to be vulnerable, XSSky confirms the vulnerability existence.

## 4.2 PUT Fuzzing Module

For the PUTs converted by the preceding module, the *PUT Fuzzing Module* confirms vulnerabilities through four steps.

First, it performs a runtime DOM analysis to determine the sink context of the XSS vulnerability implied in the PUT (§4.2.1). Second, based on the identified sink context, it selects appropriate exploit grammars and initializes the test cases for fuzzing (§4.2.2). Third, it tries to generate vulnerability-triggerable inputs by employing various mutation strategies and feedback mechanisms (§4.2.3). Finally, it confirms and reports vulnerabilities by monitoring the bug oracle (§4.2.4).

#### 4.2.1 Sink Context Analysis

As clarified in §2.2.2, the construction of XSS exploits is highly dependent on the syntactical context of the sink (i.e., sink context). Hence, to guide the fuzzer in generating effective test cases, XSSky first identifies the sink context. To be specific, for a given PUT, considering the diverse dynamic characteristics of HTML, XSSky opts to perform runtime DOM analysis to precisely analyze the HTML structure surrounding the sink and determine the sink context. This process is mainly divided into two steps: first, XSSky constructs a runtime environment for the PUT and retrieves the PUT's runtime response by crafting requests; then, by parsing the DOM tree within the response, XSSky accurately analyzes and determines the sink context.

**Step I: Retrieving the Runtime Response.** Firstly, XSSky places the converted PUT into the local Apache web directory, and then simulates a browser accessing the PUT by crafting a regular request. Notably, when constructing the request, XSSky assigns a unique string (by default, the MD5 hash of the current timestamp) to the request parameter of the source. This facilitates the subsequent rapid identification of the source's location within the DOM tree node.

**Step II: Determining the Sink Context.** Upon receiving the corresponding response, XSSky proceeds to the second step of determining the sink context. First, XSSky parses the DOM tree of the response using the Python library `lxml` [5]. Then, it locates the node where the sink parameter resides in the DOM tree by utilizing the previously assigned unique string. Finally, XSSky determines the sink context by analyzing this node and determining which of the following criteria are met:

- **JavaScript Context:** This context is identified if the node's tag attribute value is `script`, or if the node contains an `attri` attribute where the key is an HTML event attribute (e.g., `onclick`) and the value is the previously assigned unique string (i.e., source).
- **URL Context:** This context is determined if the node contains an `attri` attribute where the key is an HTML hyperlink attribute (e.g., `href`) and the value is the source.
- **HTML Attributes Context:** This context is identified if the node has an `attri` attribute where the key is not an HTML hyperlink or event attribute and the value is the source.
- **HTML Context:** This context is assumed when none of the aforementioned conditions are satisfied.

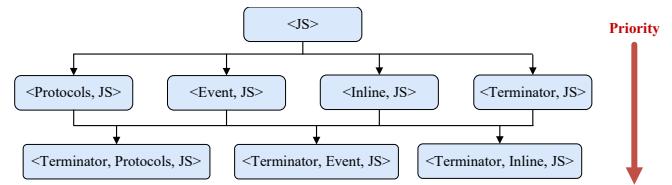


Figure 4: The priorities of different exploit grammars.

#### 4.2.2 Test Case Initialization

As discussed in §2.2.2, due to the syntax-sensitive nature of code injection, the sink context will influence the construction of the XSS exploits. Building on this, we defined appropriate exploit grammars to initialize test cases, which then guided the fuzzer in generating and mutating suitable inputs. Table 1 presents the exploit grammars we defined for different sink contexts along with their corresponding initial test cases. The underlying rationale behind this design is as follows.

**String Concatenation within Sink Contexts.** We have identified that in specific sink contexts, such as URL and JavaScript contexts, the syntax required for constructing XSS exploits is affected by whether the sink parameter is part of a string concatenation. In a JavaScript context, for example, if the sink parameter is used independently, as in "`<input onclick=\".$input.\">`", an attacker can exploit this solely using the JavaScript Code component. Conversely, when the sink parameter is concatenated with other strings, such as "`<input onclick=foo\".$input.\"foo\">`", the attacker must employ a Terminator component to break out of the string before applying the JavaScript Code component to insert malicious JavaScript. Hence, we further classify the sink contexts into six types depending on whether their parameters are concatenated with strings, as detailed in Table 1.

**Priority of Exploit Grammars.** As we introduced in §2.2.1, for XSS exploit components `<Terminator, Protocol-/Events/Inline, JS>` can form a total of eight different exploit grammars, as shown in Figure 4. However, it is clear that these grammars cannot be indiscriminately applied across all sink contexts. Such an approach would produce many invalid test cases that do not correspond to the sink context, severely reducing the efficiency of fuzzing. More importantly, it would also neglect the prioritization of exploit grammars, leading to many unnecessary attempts. Taking the HTML context as an example, certain grammars are fundamentally incompatible (e.g., `<JS>` or `<terminator JS>`, which are unusable due to the absence of a JavaScript execution environment), rendering attempts with these grammars unnecessary. Furthermore, considering the hierarchical relationship between grammars, once the simplest grammar has been tested, its supersets do not need to be tested. For instance, if all fuzzing test cases guided by the exploit grammar `<Inline, JS>` are completed without discovering vulnerabilities, there is no need to attempt its superset `<Terminator, Inline, JS>`, which also suits

Table 1: The exploit grammars and corresponding initial test cases for different sink context.

| Sink Context            | String Concat | Code Demo                                                                                        | Exploit Grammar                                                                     | Initial Test Case                                                                          |
|-------------------------|---------------|--------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| HTML Context            | -             | echo \$input;<br>echo <p>.\$input.</p>;                                                          | <Inline, JS><br><Protocols, JS><br><Events, JS>                                     | <script>alert(1)</script><br><a href=javascript:alert(1)>1</a><br><input onerror=alert(1)> |
| URL Context             | w/o           | echo "<a href="".\$input."></a>";                                                                | <Protocols, JS><br><Terminator, Events, JS><br><Terminator, Inline, JS>             | javascript:alert(1)<br>" onerror=javascript:alert(1) "<br>><script>alert(1)</script>       |
|                         | w/            | echo "<a href=foo"".\$input.">foo</a>";                                                          | <Terminator, Protocols, JS><br><Terminator, Events, JS><br><Terminator, Inline, JS> | ><a href=javascript:alert(1)><br>" onerror=alert(1) "<br>><script>alert(1)</script>        |
| HTML Attributes Context | -             | echo "<input value="".\$input.">";<br>echo "<input value=foo"".\$input.">foo";                   | <Terminator, Events, JS><br><Terminator, Protocols, JS><br><Terminator, Inline, JS> | " onerror=alert(1) "<br>><a href=javascript:alert(1)><br>><script>alert(1)</script>        |
| JavaScript Context      | w/o           | echo "<input onclick="".\$input.">";<br>echo "<script>"".\$input."></script>";                   | <JS>                                                                                | alert(1)                                                                                   |
|                         | w/            | echo "<input onclick=foo"".\$input.">foo";<br>echo "<script>var pam="".\$input.">foo;</script>"; | <Terminator, JS>                                                                    | ";alert(1);//                                                                              |

the HTML context. This is because the failure of the simplest grammar indicates that the current sanitizer effectively handles the dangerous characters involved. Consequently, its superset, which inherently employs these dangerous characters, is also likely to fail. Based on these insights, when designing exploit grammars for a sink context, we consider not only their compatibility with the sink context but also their priority. Specifically, we prioritize testing the simplest exploit grammar first, as presented in Figure 4.

### 4.2.3 Mutation and Feedback

This phase guides which characters of the test case should be mutated and how they should be mutated during fuzzing.

**Mutation Strategies.** First, we introduce the mutation strategy of XSSky. The main goal of mutation is to transform the initial test case to trigger XSS vulnerabilities and evade potentially vulnerable sanitizers. To achieve this goal, we investigated known CVEs, and existing evasion techniques from the Internet [7, 25, 35, 39]. As a result, we have collected eight different evasion methods, which are used to construct 18 different XSS exploit component mutation strategies, as illustrated in Table 2.

- *M1: Character Casing Variation.* This strategy alters character cases within inputs to bypass case-sensitive sanitizers (e.g., changing <script> to <ScRipT>).
- *M2: Multi-Form Keyword.* By duplicating keywords, this strategy exploits sanitizer handling of repeated patterns (e.g., transforming <script> into <scr<script>ipt>).
- *M3: Invisible Character Embedding.* This strategy involves inserting invisible characters like spaces or tabs into inputs (e.g., changing <script> to <script%09>) to disrupt pattern-matching mechanisms.
- *M4: Special Characters Embedding.* By introducing special characters into inputs to disrupt the parsing logic of sanitizers, e.g., changing onload= to onload!#\$%&=.

- *M5: Alternative Keywords.* By substituting standard keywords with alternatives that have similar functions (e.g., replacing <img onerror= with <svg onload=), this strategy tests the completeness of keyword-based sanitizers.
- *M6: Unicode Encoding.* This strategy involves using Unicode escape sequences to represent characters, such as converting alert(1) to al\u0065rt(1). This can bypass sanitizers that do not properly handle character encodings.
- *M7: Equivalent Semantic Variation.* By employing different syntactic forms that have the same semantic meaning, e.g., changing alert(1) to top["al"+"ert"](1). It evaluates a sanitizer's ability to handle equivalent expressions.
- *M8: Tricks.* This strategy includes a variety of clever techniques designed to alter the structure of code or data slightly, such as using unconventional syntax or exploiting quirks in JavaScript language (e.g., alert`1`). These tricks can test the edge cases of a sanitizer's parsing and execution logic.

**Feedback Mechanism.** Then, we introduce the feedback mechanism of XSSky. As discussed in §3.3, we can hook all string comparison functions (e.g., php\_pcre\_match\_impl) and string modification functions (e.g., php\_pcre\_replace\_impl) in the PHP interpreter. The hooked information can then be used as feedback to guide the mutation and generation of test cases in the fuzzing process.

Figure 5 illustrates an example of how we hook the PHP built-in function preg\_replace(). The inserted code captures and logs key information every time preg\_replace() is invoked, which is then fed back to the fuzzer to guide the mutation of test cases. The logged information includes the line number where the function is invoked, the input string, the matched substring, and the return string after modification. This detailed logging provides valuable context for understanding how preg\_replace() is utilized at runtime and suggests whether the input values remain consistent with expectations after being processed by the function.

**Fuzzing Scheme.** Finally, we describe how XSSky utilizes mutation strategies and feedback mechanisms to iteratively

Table 2: Mutation strategies for different components.

| Scope       | Strategies                        | Examples                            |
|-------------|-----------------------------------|-------------------------------------|
| <Inline>    | M1: Character Casing Variation    | <script>→<ScRiPt>                   |
|             | M2: Multi-Form Keyword Encoding   | <script>→<scr<script>ipt>           |
|             | M3: Invisible Character Embedding | <script>→<script%09>                |
| <Protocols> | M1: Character Casing Variation    | javascript: → jAVasCriPt:           |
|             | M2: Multi-Form Keyword Encoding   | javascript: → javasjavascriptript:  |
|             | M3: Invisible Character Embedding | javas#x09;cript: → java#x09;script: |
| <Events>    | M1: Character Casing Variation    | onerror= → oNErRor=                 |
|             | M2: Multi-Form Keyword Encoding   | onerror= → oneronerror=             |
|             | M3: Invisible Character Embedding | onerror= → onerror%09=              |
|             | M4: Special Characters Embedding  | onload= → onload!#\$%&=             |
|             | M5: Alternative Keywords          | <img onerror= → <svg onload=        |
| <JS>        | M1: Character Casing Variation    | alert(1) → AlErT(1)                 |
|             | M2: Multi-Form Keyword Encoding   | alert(1) → alealert(1)              |
|             | M3: Invisible Character Embedding | alert(1) → alert%09(1)              |
|             | M5: Alternative Keywords          | alert(1) → confirm(1)               |
|             | M6: Unicode Encoding              | alert(1) → alu0065rt(1)             |
|             | M7: Equivalent Semantic Variation | alert(1) → top["al"+"ert"](1)       |
|             | M8: Tricks                        | alert(1) → alert`1`                 |

mutate initial test cases during the fuzzing process. Initially, XSSky begins with the initial test case derived from the first predefined exploit grammar. By analyzing the sanitizer-flagged string in the feedback, it can identify which characters are being blocked by the sanitizer. Subsequently, XSSky applies a corresponding mutation strategy to make targeted mutations, which are then re-input. By examining the line number information in the feedback, XSSky determines whether more restrictions have been bypassed, thereby evaluating the effectiveness of the current mutation strategy. If further restrictions are bypassed, XSSky continues using the same mutation strategy in subsequent iterations. Conversely, if no progress is made, it will switch to a different mutation strategy until a vulnerability is confirmed by the bug oracle. If all mutation strategies are exhausted without generating a test case that triggers a vulnerability, XSSky proceeds to the next exploit grammar and repeats the process. The tested PUT is only deemed secure after all exploit grammars have been tested without success.

#### 4.2.4 Bug Oracle

This phase is used to determine whether a particular test case has revealed an XSS vulnerability during the fuzzing process. Generally, a proof of concept (PoC) confirming the presence of an XSS vulnerability aims to trigger a browser popup. Therefore, XSSky employs the Python library Selenium [6] to simulate browser requests to access PUT and listen for JavaScript popups in the browser. Additionally, since some test cases require interaction to be triggered, such as clicking links (e.g., <a href="javascript:alert(1)">clickme</a>)

```

1 PHPAPI zend_string *php_pcre_replace_impl
 (pcre_cache_entry *pce, char *subject, zend_string *replace_str){
2 ...
3 match_data = pcre2_match_data_create_from_pattern(pce->re);
4 count = pcre2_match(pce->re, (PCRE2_SPTR)subject, match_data);
5 if (count >= 0) { /* Matched */
6 ...
7 offsets = pcre2_get_ovector_pointer(match_data);
8 size_t match_len = offsets[1] - offsets[0];
9 lineno = zend_get_executed_lineno();
10 matched_string = zend_string_init(subject + offsets[0], match_len);
11 FILE *log_file = fopen(LOG_FILE, "a");
12 if (log_file) {
13 fprintf(log_file, "%d", lineno); /* Line number */
14 fprintf(log_file, "%s", subject); /* Input string */
15 fprintf(log_file, "%s", matched_string); /* Matched string */
16 fprintf(log_file, "%s", replace_str); /* Return string */
17 fclose(log_file);
18 }
19 ...
20 }

```

Figure 5: An example of hooking the string modification function `preg_replace()` in PHP interpreter. The inserted hook code is highlighted with a light yellow background.

or hovering the cursor over elements (e.g., <input onmouseover=alert(1)>), XSSky uses a crawler to simulate these interactions. This helps to thoroughly evaluate whether a popup is effectively triggered after the test case is injected, thereby confirming the XSS vulnerability.

## 5 Evaluation

**Implementation.** We implemented XSSky based on the state-of-the-art static analyzer TChecker [41], with 7,004 lines of code in Python. Specifically, in the path location stage, we directly employed TChecker. Due to its implementation of detecting potentially vulnerable paths, which stops analysis upon encountering pre-modeled sanitizers, we modified this logic to ensure it can also report paths containing sanitizers completely. In the PUT conversion stage, we conducted our analysis based on the code property graph (CPG) constructed by the existing framework. However, during our evaluation phase, we observed minor bugs in call graph construction; specifically, a few call sites did not establish call edges with their apparent call targets. We addressed these issues with appropriate fixes. In summary, this module was implemented in a total of 4,512 lines of code. During the fuzzing stage, we implemented a total of 2,492 lines of code, including 347 lines for sink context analysis, 1,737 lines for implementing mutation strategies and the fuzzing scheme, and 408 lines for the implementation of the bug oracle.

**Experiments.** Our evaluation is organized by answering the following research questions:

- RQ1: How effective is XSSky in converting source-sink

Table 3: Breakdown of our evaluation dataset and the XSS vulnerabilities detected by XSSky.

| Applications       | # Stars  | # LoCs           | # Vulns (w/. san) |
|--------------------|----------|------------------|-------------------|
| Matomo             | 20,079   | 346,773          | -                 |
| WordPress          | 19,772   | 334,247          | -                 |
| Firefly III        | 17,005   | 102,536          | -                 |
| Grav CMS           | 14,651   | 110,109          | -                 |
| October            | 11,056   | 39,986           | -                 |
| Dolibarr           | 5,629    | 1,119,801        | 3 (2)             |
| GLPI               | 4,447    | 476,190          | 5 (3)             |
| OpenEMR            | 3,249    | 663,962          | 4 (4)             |
| EasyImage2.0       | 2,934    | 17,207           | 17 (16)           |
| phpIPAM            | 2,309    | 137,478          | -                 |
| AdminLTE           | 2,083    | 9,152            | 1 (0)             |
| Live helper chat   | 2,002    | 304,883          | 7 (4)             |
| Vmphp              | 1,840    | 103,023          | -                 |
| Mantis Bug Tracker | 1,675    | 79,315           | -                 |
| ProjectSend        | 1,460    | 25,391           | 1 (1)             |
| Jstolpe Blog       | 239      | 24,506           | 1 (1)             |
| Chyrp              | 231      | 18,966           | 2 (0)             |
| Typesetter CMS     | 228      | 86,077           | 10 (0)            |
| LiteCart           | 207      | 31,772           | 3 (0)             |
| Tiny Tiny RSS      | 195      | 39,760           | 6 (0)             |
| <b>Total</b>       | <b>/</b> | <b>4,082,064</b> | <b>60 (31)</b>    |

paths into locally executable PUTs from real-world applications? (in §5.1)

- RQ2: How effective is XSSky in detecting XSS vulnerabilities in real-world applications? (in §5.2)
- RQ3: How accurate is XSSky compared to the state-of-the-art approaches? (in §5.3)
- RQ4: How efficient is XSSky in performing the end-to-end analysis? (in §5.4)

**Dataset.** Our dataset consists of 20 popular PHP web applications, and the selection criteria are as follows. First, we filtered applications on GitHub [1] using the PHP language keyword. We then selected these applications with more than 100 stars to ensure their popularity. Finally, to ensure the representativeness of the dataset, we randomly chose 5 applications with over 10,000 stars, 10 applications with over 1,000 stars, and 5 applications with over 100 stars to comprise the dataset. The detailed dataset is presented in Table 3.

**Setup.** All the experiments in this section are run on a Ubuntu 20.04 machine with an Intel Xeon Gold 6242 processor (64 cores) and 512 GB memory. Furthermore, both our PUTs and the fuzzer engine were run under Apache 2.4.41 and PHP 7.4.0. All baselines in §5.3 were executed according to their provided guidelines [8–10, 15], using default configurations.

## 5.1 RQ1: PUT conversion

In this phase, we evaluated the effectiveness of XSSky in converting the source-sink paths reported by TChecker [41]

Table 4: Statistical analysis of code elements of each PUT in Average, Median, and Median Absolute Deviation (MAD).

| Code Elements       | Average | Medium | MAD |
|---------------------|---------|--------|-----|
| Function Definition | 17      | 4      | 3   |
| Temporary Variables | 12      | 4      | 2   |
| Lines of Code       | 1,321   | 396    | 182 |

into locally executable PUTs within our dataset. Overall, for 7,005 source-sink paths, XSSky successfully converted 6,997 (99.89%) of them into locally executable PUTs.

**Unsuccessful Path Conversion Analysis.** For the 8 paths that could not be successfully converted, we conducted a detailed analysis and found that the root cause was the same: *there are function calls to third-party libraries on the paths.* Since XSSky analyzes the source code of the target application, which lacks the definitions for third-party library functions, it is unable to relocate these function calls along the path, leading to unsuccessful conversion of the PUTs.

**Statistical Analysis.** We also conducted a statistical analysis of the PUT conversion by XSSky. As shown in Table 4, each PUT contains an average of 1,321 lines of code, including 17 function definitions and 12 temporary variables. Notably, the median and MAD indicate that the number of temporary variables and function definitions per PUT is significantly lower than the average. This discrepancy is mainly due to the complex code logic of two applications in our dataset (i.e., WordPress and Mantis Bug Tracker), which utilize more function calls and externally or dynamically assigned variables, thereby inflating the average values.

## 5.2 RQ2: XSS Vulnerability Detection

In this phase, we evaluated the effectiveness of XSSky in detecting XSS vulnerabilities within our dataset through PUT fuzzing. In all, XSSky reported 74 distinct potential vulnerabilities. The quality of these reports is discussed as follows.

**Report Verification.** First, we manually investigated the reported vulnerabilities to confirm their exploitability in the runtime environments of real-world applications. Overall, we confirmed that 60 / 74 (81.08%) reports are indeed XSS vulnerabilities, with details presented in Table 3. Attackers can exploit these vulnerabilities to compromise the corresponding applications, including stealing sensitive user information, redirecting users to malicious websites, or even executing unauthorized actions on behalf of users.

**False Positives Analysis.** For the remaining 14 false positives, we conducted a thorough examination of their causes and found that they mainly stem from two aspects.

- *Restricted by control flow constraints (6 FPs).* Given that the static tool TChecker, used by XSSky, employs a forward data-flow analysis to locate source-sink paths, it results in

Table 5: Breakdown of the sink contexts and sanitizers of XSS vulnerabilities detected by XSSky.

| Sink Context            | w/. san | w/o. san | Total |
|-------------------------|---------|----------|-------|
| HTML Context            | 9       | 19       | 28    |
| URL Context             | 2       | 3        | 5     |
| HTML Attributes Context | 14      | 5        | 19    |
| JavaScript Context      | 6       | 2        | 8     |
| <b>Total</b>            | 31      | 29       | 60    |

reported paths lacking control flow constraints. This limitation means that XSSky cannot account for these constraints during the conversion and fuzzing of the PUT. However, we believe that these false positives can be eliminated in the future by incorporating advanced static analyzers.

- *Dead code (8 FPs)*. These false positives were found in the dead code of the target applications. Notably, XSSky is capable of confirming vulnerabilities in PUTs of these cases. Unfortunately, static detector failed to identify that these paths were not potential vulnerable paths and thus provided them to XSSky, resulting in false positives.

**Vulnerability Causes Analysis.** We further analyzed the sink contexts and root causes of the confirmed vulnerabilities, as detailed in Table 5. The causes of these vulnerabilities can be attributed to the following three categories:

- *Lack of sanitizer protection (29 cases)*. These vulnerabilities are caused by the lack of sanitizer protection along the source-sink path, allowing attackers to exploit XSS vulnerabilities using the most basic exploits.
- *Improper sanitizer usage (24 cases)*. Although sanitizers were deployed on these vulnerable paths, we found that they were evaded due to misuse (as discussed in Figure 2).
- *Inadequate sanitizer policies (7 cases)*. For these vulnerabilities, we found that the developers had correctly deployed sanitizers for the corresponding sink context. However, due to inadequate detection of dangerous characters, attackers were able to craft exploits using alternative keywords, resulting in XSS vulnerabilities.

For the 31 sanitizers that are evaded due to improper use or insufficient sanitization, we present their detailed information in Appendix-Table 8. These sanitizers comprise 8 PHP built-in functions and 23 developer-defined sanitization rules. Additionally, there are 6 developer-customized sanitizers, each composed of at least one or more built-in functions or sanitization rules. On average, with averaging 67.3 LoC, illustrating their complexity.

Furthermore, we observed an interesting finding: vulnerabilities in the HTML Attributes Context had the highest rate of evasion due to sanitizer failures (14/19, 73.68%), while those in the HTML Context had the lowest rate (9/28, 32.14%). Sanitizers like `strip_tags()`, `htmlspecialchars()`, or

`htmlentities()` are effective in the HTML Context but can be bypassed in the HTML Attributes Context with payloads like `"' onclick=alert(1)//"`. Deploying sanitizers during development without considering the sink context will increase the risk of evasion.

### 5.3 RQ3: Comparison

**Baseline Setup.** We compare the effectiveness of XSSky against two baseline types in detecting XSS vulnerabilities.

- *Static Detectors*. Defining source and sink and then performing taint analysis to identify potential vulnerabilities is currently the mainstream approach for static vulnerability detectors. Therefore, we have chosen to include the SOTA work TChecker [41] as one of our baselines.
- *Dynamic Testers*. Dynamic analysis is another mainstream approach for detecting XSS vulnerabilities. Given that our work focuses on the source code of target applications, without a runtime environment. To ensure a fair comparison, we compared this line of work on XSSky-generated PUTs, guided by the following considerations: XSSky employs static code exploration, while baselines use dynamic methods. PUT-based comparison helps avoid code-coverage-induced false negatives and focuses on comparing vulnerability confirmation capability. Since PUTs provide executable environments compliant with baseline requirements, we believe this setup doesn't hinder their performance. Therefore, we have disabled the fuzzer engine in XSSky and have equipped XSSky with these dynamic testers. In all, we selected four open-source and well-known techniques aimed at detecting XSS vulnerabilities, namely w3af [9] (employed by several renowned crawlers [23, 24]), Burp Suite [15], Black widow [24] and webFuzz [48], and integrated them into XSSky, referring to these integrations as XSSky-w3af, XSSky-Burp, XSSky-bw and XSSky-webFuzz.

**Benchmark.** Comparing the accuracy of each work requires a comprehensive enumeration of all vulnerabilities within our dataset, which is impractical. Therefore, to ensure a fair comparison, we followed the mainstream approach [38, 45, 51], which involves constructing a ground truth by aggregating all vulnerabilities identified by both XSSky and the baseline tools in our dataset. It is important to note that each vulnerability involved in the ground truth underwent careful scrutiny. This was accomplished by manually confirming them as true positives. In total, the ground truth consists of 60 vulnerabilities. It is worth noting that XSSky can detect all of these cases, showcasing its remarkable capability by combining the strengths of both dynamic and static analysis.

**Results Overview.** Overall, the results presented in Table 6 show that XSSky outperforms several baselines in detecting XSS vulnerabilities by improving precision by 11.48%~642.49% and recall by 87.51%~172.70%. Notably, it not only identifies all the vulnerabilities detected by the

Table 6: Comparison of the effectiveness between XSSky and state-of-the-art (SOTA) techniques.

| Baselines     | TP (w/. san) | FP  | FN | Prec(%) | Recall(%) |
|---------------|--------------|-----|----|---------|-----------|
| XSSky         | 60 (31)      | 14  | 0  | 81.08%  | 100.00%   |
| TChecker      | 32 (3)       | 261 | 28 | 10.92%  | 53.33%    |
| XSSky-w3af    | 22 (2)       | 10  | 38 | 68.75%  | 36.67%    |
| XSSky-bw      | 24 (2)       | 9   | 36 | 72.73%  | 40.00%    |
| XSSky-webFuzz | 25 (10)      | 11  | 35 | 69.44%  | 41.67%    |
| XSSky-Burp    | 29 (2)       | 11  | 31 | 72.50%  | 48.33%    |

baselines but also discovers 18 additional vulnerabilities that none of the baselines could detect.

**Compared with Static Detector.** Compared to TChecker, XSSky improves the precision of XSS vulnerability detection by 624.49% and the recall rate by 87.51%. Specifically, the 261 false positives reported by TChecker mainly stem from two main aspects. (1) 179 false positives caused by custom sanitizers present on the source-sink path, which are not included in TChecker’s pre-built models, leading to incorrect identification of unprotected paths. (2) 73 false positives due to TChecker’s inability to analyze the semantics of typecasting operations (e.g., `int()` operation casts strings to integers). Comparably, XSSky benefits from its novel fuzzing design, which does not rely on sanitizer modeling or static semantic analysis, thereby effectively avoiding these false positives. For the remaining 9 false positives reported by TChecker, they share the same cause as those identified by XSSky, which we have discussed in §5.2. Regarding the 28 false negatives from TChecker, these were due to hasty assumptions about the effectiveness of sanitizers. Specifically, TChecker assumed that the presence of a sanitizer on a path guarantees security, whereas these false negatives resulted from sanitizer evasion. While XSSky does not rely on these assumptions, allowing it to successfully detect these vulnerabilities. Interestingly, TChecker still detected three vulnerabilities caused by sanitizer evasion. This was due to its insufficient sanitizer modeling, which failed to identify these sanitizers, thereby inadvertently reporting them as vulnerabilities.

**Compared with Dynamic Testers.** Among dynamic testers, although XSSky-Burp achieved the best results (i.e., discovered 29 vulnerabilities), we found that its recall was still only 48.33%. Compared to them, XSSky has a significant advantage in detecting vulnerabilities arising from both the absence and the evasion of sanitizers. Specifically, we found that XSSky-w3af, XSSky-Burp, XSSky-bw, and XSSky-webFuzz all lack sink context analysis, leading to ineffective attempts and failure to construct test cases that align with the syntactic structure of the target sink context, thus missing many vulnerabilities caused by absent sanitizers. Regarding vulnerabilities arising from sanitizer evasion, we found that all baselines detect only a few cases (2-10). This is mainly because identifying vulnerable sanitizers usually requires a targeted mutation and

Table 7: Performance (seconds) of XSSky for analyzing each application is broken down by different modules in Average, Median, and Median Absolute Deviation (MAD).

| XSSky Module     | Average    | Medium  | MAD     |
|------------------|------------|---------|---------|
| Path Location    | 2,013.51s  | 539.45s | 537.39s |
| PUT Conversion   | 9,807.49s  | 106.96s | 105.68s |
| PUT Fuzzing      | 1,651.48s  | 45.63s  | 43.62s  |
| End-to-end Total | 13,472.48s | 692.04s | 686.69s |

evolution process. Existing dynamic testers fall short in this aspect; although webFuzz employs a more diverse mutation strategy, it lacks an effective feedback mechanism for guidance, failing to discover 21 vulnerabilities caused by evaded sanitizers. In contrast, XSSky’s novel feedback mechanism enables it to effectively understand sanitizers and guide targeted mutations, thus identifying these vulnerabilities. Regarding false positives, given that dynamic testers have corresponding bug oracles, no false positives emerged during the testing of the PUTs. However, when confirming vulnerabilities within the application, they encounter the same causes of false positives as XSSky, as discussed in §5.2.

## 5.4 RQ4: Efficiency

In this phase, we evaluated the efficiency of XSSky by performing end-to-end analysis on our dataset. In total, XSSky spent an average of 3.74 hours detecting XSS vulnerabilities for a target application from our dataset. The details are presented in Table 7. Specifically, in the *Path Location* module, XSSky employed TChecker to complete the task, with an average time of approximately 0.56 hours per application. Meanwhile, the *PUT conversion* module appears to be a relatively time-consuming task, averaging 2.72 hours per application. However, when considering the Median and MAD, its time cost is actually lower than that of the path location. This indicates that aside from a small number of cases involving large-scale path conversions, the PUT conversion for most paths is efficient. Regarding the *PUT Fuzzing* module, it takes an average of 0.46 hours to test all PUTs in a target application. Additionally, we observed that both the Median and MAD for these modules are significantly lower than the average values. This is because our dataset includes two particularly large applications (i.e., Dolibarr and OpenEMR), whose total lines of code are comparable to the combined total of the other 18 applications, thereby inflating the overall average analysis time. Overall, given that vulnerability detection is usually an offline task, we believe that having a stronger capability to detect vulnerabilities is more favorable.

## 6 Discussion

**Inherent Limitations of Static Analysis.** Although XSSky significantly outperforms several baselines in precision and recall, it still reports some false positives. The key reason lies in that, XSSky is built upon existing static detectors (e.g., TChecker [41]) to identify XSS vulnerabilities (i.e., take as inputs the source-sink paths reported by existing tools). Hence, the inherent limitations of static detectors (e.g., ignoring control flow constraints when reporting source-sink paths, reporting source-sink paths among dead code, etc.) consequently hurt the effectiveness of XSSky. This issue can be potentially addressed by incorporating more advanced static detectors.

**Defense-in-depth Mechanism.** Apart from preventing XSS through sanitizers in the code, another approach is to prevent XSS by configuring defense-in-depth mechanisms, such as Content Security Policy (CSP). The developers can define a policy (via HTTP headers) specifying allowed sources for scripts, styles, images, etc. This, in turn, helps prevent malicious behavior. Given that XSSky is a technique focused on code security, such mechanisms are outside its scope.

**Realism of Vulnerability Reports.** Another common concern for local fuzzing of source-sink paths is the realism of the identified vulnerabilities (i.e., whether the vulnerability confirmed with the PUT can also be triggered in the original application). Although our adaptations on undefined symbols may indeed lead to false positives of vulnerability detection, after carefully analyzing the experimental results, we confirmed that no false positives were caused by this reason. Despite the aforementioned shortcomings, we believe that XSSky represents a step forward in advancing XSS vulnerability detection.

**Future Work.** In the future, the capabilities of XSSky can be improved in two aspects: (1) **equip with more XSS sanitizer evasion techniques**. Currently, XSSky is an academic prototype and does not cover all evasion techniques. In the future, end-users can customize a wider range of techniques to strengthen its detection capabilities; (2) **expand XSSky's ability to detect stored XSS vulnerabilities**. Following the high-level design of XSSky, **end-users only need to define the data retrieval function as the source**, and XSSky can acquire this ability.

## 7 Related Work

**Sanitizer-targeted XSS Detection.** These works can be classified into the following two types. One line of work aims to improve existing static analyzers by verifying the security of sanitizers through Satisfiability Modulo Theories (SMT) solvers [18, 27, 35, 37]. Lathouwers *et al.* [37] used Symbolic Finite Transducers to verify sanitizer implementations against a given security specification. Alhuzali *et al.* [18] applied the Z3-str solver, while Klein *et al.* [35] utilized Deterministic Finite Automata to generate exploits to evade sanitizers.

Eriksson *et al.* [27] adopted a two-way alternating finite-state automaton for generating testing payloads that meet client-side format requirements. Despite their effectiveness, these methods require extensive modeling by security experts, and inevitably face challenges like memory explosion when handling sanitizers intertwined with complex business logic.

Another line of work focuses on a specific type of sanitizer based on HTML parsers. For instance, MutaGen [36] revealed that such sanitizers and user browsers could potentially produce divergent results when parsing HTML fragments, thus potentially causing bypasses. In comparison, XSSky focuses on general types of XSS vulnerabilities, rather than a specific type of sanitizer evasion. Besides, such types of sanitizers have not yet been found in our dataset. Furthermore, given that MutaGen requires end-users to inspect and annotate HTML parser-based sanitizers for testing, it lacks some flexibility compared to XSSky.

**Vulnerable Code Clone Detection.** Another technique that could aid static tools in identifying vulnerable sanitizers is vulnerable code clone detection, which matches unknown vulnerabilities by extracting features from a large set of known vulnerabilities. Jang *et al.* [32] introduced a token-based approach called ReDeBug to locate unpatched code clones at the line-level granularity. Kim *et al.* [34] presented VUDDY, a scalable approach for detecting vulnerable code clones using several vulnerability-preserving abstraction techniques. Xiao *et al.* [51] proposed MVP and Shi *et al.* [45] introduced ReCurScan; both approaches take security patches as input and extract signatures from both pre-patch and post-patch code for vulnerability detection. However, these techniques require the presence of a reference vulnerability, and thus are unable to detect vulnerable code that appears for the first time.

## 8 Conclusion

In this paper, we propose XSSky, a static sanitizer-tolerant detector for accurately detecting XSS vulnerabilities within PHP web applications. XSSky converts source-sink paths identified by the static detector into locally executable PUTs and employs a path-persistent fuzzing strategy guided by feedback from the PHP interpreter. This novel design avoids the burdensome sanitizer modeling and enhances the ability of static detectors to detect vulnerable sanitizers. Our evaluation results demonstrate that XSSky can detect 60 previously unknown XSS vulnerabilities in 20 popular PHP web applications, outperforming several baselines by improving precision by 11.48%~642.49% and recall by 87.51%~172.70%.

## Acknowledgement

We would like to thank our shepherd and the anonymous reviewers for their helpful comments and feedback. This work was supported in part by the National Natural Science Foun-

dation of China (U2436207, 62172105, 62402116) and Hong Kong RGC Projects (PolyU15224121, PolyU15231223). Yuan Zhang, Xiapu Luo 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.

## 9 Open Science

The source code of XSSky is available upon reasonable request via the Zenodo [17]. To mitigate potential misuse (detailed in §10), we have implemented a controlled access policy. Readers can request access by contacting us through their institutional email, stating their name, affiliation, and intended use in the email. We will then vet the provided information and grant or deny access to the source code. Our goal is to grant access to academic and industry researchers for purposes of building upon our research or advancing the field of cybersecurity, while ensuring XSSky isn't exploited for any malicious purposes.

## 10 Ethical considerations

This research was conducted with careful attention to ethical responsibilities, which are detailed below.

**Vulnerability Verification Environment.** In our evaluation part, all of the applications are open-source, and we downloaded and conducted analysis on them locally through XSSky. This process did not involve any data related to real user privacy. Then, after XSSky reported potential vulnerabilities, we built these applications on a local server for vulnerability verification. This process also did not involve any user privacy.

**Vulnerability Disclosure.** In terms of vulnerability disclosure, all our vulnerability reports have strictly adhered to the timeline of the CVE Numbering Authorities (CNA), along with proactive communication with all developers. Specifically, after we manually confirmed the vulnerabilities, we immediately contacted the developers, including raising issues in the Github repository and via email. We carefully explained to the developers the cause of the vulnerability, the details of vulnerability exploitation, and the corresponding recommended solutions for fixing the issues. Although the detected vulnerabilities were still being fixed at the time we submitted this paper, we did not mention any information about these vulnerabilities in the paper. Therefore, the release of this paper will not cause any harm to real-world users.

**Artifact Availability and Potential for Misuse.** We have carefully considered the ethical implications of releasing the XSSky artifact.

- *Potential Risks:* As a tool that proved effective in discovering numerous previously unknown XSS vulnerabilities, with many involving sanitizer bypasses, XSSky could be repurposed by malicious actors to find and exploit vulnerabilities in live web applications. We acknowledge that while other XSS testing tools are publicly available, XSSky's novel path-persistent fuzzing and interpreter-feedback mechanisms may grant it an advantage in discovering vulnerabilities that other tools miss, heightening the risk of misuse.
- *Mitigating Factors in Real-World Systems:* We recognize that modern applications often deploy defense-in-depth mechanisms. Security measures like a well-configured Content Security Policy (CSP), Web Application Firewalls (WAFs), or Trusted Types can mitigate or block the exploitation of XSS vulnerabilities. However, these defenses are not universally adopted, can be misconfigured, and are sometimes overlooked.
- *Trade-offs and Our Approach:* We weighed the risks of unrestricted release against the scientific benefits of open access. Releasing the tool without any controls could empower attackers, but restricting it entirely would prevent the community from building on our work to create better defenses. We have therefore chosen a middle ground: controlled distribution. By making the artifact available upon request to identified researchers, we aim to support legitimate scientific inquiry while adding a layer of accountability that deters casual misuse. This approach balances our commitment to open science with our ethical duty to prevent foreseeable harm.

## References

- [1] Github. <https://github.com>.
- [2] Programming Language Security. <https://thehackernews.com/2015/12/programming-language-security.html>, 2015.
- [3] Baidu. <https://www.baidu.com>, 2024.
- [4] CVE-2020-14475. <https://nvd.nist.gov/vuln/detail/CVE-2020-14475>, 2024.
- [5] Python library - lxml. <https://lxml.de>, 2024.
- [6] Python library - Selenium. <https://www.selenium.dev>, 2024.
- [7] Sink Context of XSS vulnerabilities. [https://cheatsheetseries.owasp.org/cheatsheets/Cross\\_Site\\_Scripting\\_Prevention\\_Cheat\\_Sheet.html#output-encoding](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#output-encoding), 2024.
- [8] Source Code - TChecker. <https://github.com/cuhk-seclab/TChecker>, 2024.

- [9] Source Code - w3af. <https://github.com/andresriancho/w3af>, 2024.
- [10] Source Code - webFuzz. <https://bitbucket.org/sregrp/webfuzz-fuzzer>, 2024.
- [11] Tumblr. <https://www.tumblr.com>, 2024.
- [12] W3Techs - Usage statistics of PHP for websites. <https://w3techs.com/technologies/details/pl-php>, 2024.
- [13] Wikipedia. <https://www.wikipedia.org>, 2024.
- [14] WordPress. <https://wordpress.com>, 2024.
- [15] Burp Suite. <https://portswigger.net/burp>, 2025.
- [16] CVE Details - Vulnerabilities By Types. <https://www.cvedetails.com/vulnerabilities-by-types.php>, 2025.
- [17] XSSky - Source Code. <https://zenodo.org/records/15580726>, 2025.
- [18] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. NAVEX: Precise and scalable exploit generation for dynamic web applications. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 377–392, 2018.
- [19] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. Fudge: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 975–985, 2019.
- [20] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE european symposium on security and privacy (EuroS&P)*, pages 334–349. IEEE, 2017.
- [21] Johannes Dahse and Thorsten Holz. Simulation of built-in php features for precise static code analysis. In *NDSS*, volume 14, pages 23–26, 2014.
- [22] Johannes Dahse and J Schwenk. Rips-a static source code analyser for vulnerabilities in php scripts (2010), 2012.
- [23] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 523–538, 2012.
- [24] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. Rescan: A middleware framework for realistic and robust black-box web application scanning. In *Network and Distributed System Security (NDSS) Symposium*, 2023.
- [25] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. Kameleonfuzz: evolutionary fuzzing for black-box xss detection. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 37–48, 2014.
- [26] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. Black widow: Blackbox data-driven web scanning. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1125–1142. IEEE, 2021.
- [27] Benjamin Eriksson, Amanda Stjerna, Riccardo De Masellis, Philipp Rüemmer, and Andrei Sabelfeld. Black ostrich: Web application scanning with string solvers. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 549–563, 2023.
- [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 *USENIX Security Symposium*, 2024.
- [29] Xiangyu Guo. Evocrawl: Exploring web application code and state using evolutionary search. Master’s thesis, University of Toronto (Canada), 2023.
- [30] Mark Hills, Paul Klint, and Jurgen Vinju. An empirical study of php feature usage: a static analysis perspective. In *Proceedings of the 2013 international symposium on software testing and analysis*, pages 325–335, 2013.
- [31] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed grey-box fuzzing with provable path pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 36–50. IEEE, 2022.
- [32] Jiyong Jang, Abeer Agrawal, and David Brumley. Re-DeBug: Finding Unpatched Uode Clones in Entire OS Distributions. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.
- [33] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P’06)*, pages 6–pp. IEEE, 2006.
- [34] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *Proceedings of the 38th IEEE Symposium on Security and Privacy*, 2017.

- [35] David Klein, Thomas Barber, Souphiane Bensalim, Ben Stock, and Martin Johns. Hand sanitizers in the wild: A large-scale study of custom javascript sanitizer functions. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 236–250. IEEE, 2022.
- [36] David Klein and Martin Johns. Parse me, baby, one more time: Bypassing html sanitizer via parsing differentials. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 203–221. IEEE, 2024.
- [37] Sophie Lathouwers, Maarten Everts, and Marieke Huisman. Verifying sanitizer correctness through black-box learning: A symbolic finite transducer approach. In *Proceedings of the 6th International Conference on Information Systems Security and Privacy: Volume 1: ForSE*, pages 784–795. SciTePress, 2020.
- [38] Fengyu Liu, Youkun Shi, Yuan Zhang, Guangliang Yang, Enhao Li, and Min Yang. Mocguard: Automatically detecting missing-owner-check vulnerabilities in java web applications. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 10–10. IEEE Computer Society, 2024.
- [39] Miao Liu, Boyu Zhang, Wenbin Chen, and Xunlai Zhang. A survey of exploitation and detection methods of xss vulnerabilities. *IEEE access*, 7:182004–182016, 2019.
- [40] Yuwei Liu, Yanhao Wang, Xiangkun Jia, Zheng Zhang, and Purui Su. Afdgen: Whole-function fuzzing for applications and libraries. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1901–1919. IEEE, 2024.
- [41] Changhua Luo, Penghui Li, and Wei Meng. Tchecker: Precise static inter-procedural analysis for detecting taint-style vulnerabilities in php applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2175–2188, 2022.
- [42] Sebastian Neef, Lorenz Kleissner, and Jean-Pierre Seifert. What all the fuzz is about: A coverage-guided fuzzer for finding vulnerabilities in php web applications. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 1523–1538, 2024.
- [43] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. jak: Using dynamic analysis to crawl and test modern web applications. In *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015. Proceedings 18*, pages 295–316. Springer, 2015.
- [44] Seemanta Saha, Laboni Sarker, Md Shafiuazzaman, Chaofan Shou, Albert Li, Ganesh Sankaran, and Tefik Bultan. Rare path guided fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1295–1306, 2023.
- [45] Youkun Shi, Yuan Zhang, Tianhao Bai, Lei Zhang, Xin Tan, and Min Yang. Recurscan: Detecting recurring vulnerabilities in php web applications. In *Proceedings of the ACM on Web Conference 2024*, pages 1746–1755, 2024.
- [46] Aleksei Stafeyev, Tim Recktenwald, Gianluca De Stefano, Soheil Khodayari, and Giancarlo Pellegrino. Yurascanner: Leveraging llms for task-driven web app scanning. 2024.
- [47] Erik Trickle, 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 *2023 IEEE symposium on security and privacy (SP)*, pages 2658–2675. IEEE, 2023.
- [48] Orpheas van Rooij, Marcos Antonios Charalambous, Demetris Kaizer, Michalis Papaevripides, and Elias Athanasopoulos. webfuzz: Grey-box fuzzing for web applications. In *Computer Security—ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I 26*, pages 152–172. Springer, 2021.
- [49] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512. IEEE, 2010.
- [50] Seongil Wi, Sijae Woo, Joyce Jiyoung Whang, and Soeul Son. HiddenCPG: Large-Scale Vulnerable Clone Detection using Subgraph Isomorphism of Code Property Graphs. In *Proceedings of the ACM Web Conference 2022*, 2022.
- [51] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. MVP: Detecting vulnerabilities using Patch-Enhanced vulnerability signatures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1165–1182, 2020.

Table 8: Detailed information on all evaded sanitizers and rules from the vulnerabilities identified by XSSky.

| Types                         | Details                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Built-in Functions            | strip_tags, strpos, json_encode, json_decode, is_array, html_entity_decode, htmlspecialchars, filter_var                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Sanitization Rules            | <pre> preg_replace('/([\\])*\\"/', '\\\\', \$var), preg_replace('/([&lt;&gt;])([+]?\\d)/', '\\1 \\2', \$var), preg_replace(array('/^[^\\?]*%/'), preg_replace(array('/^[a-z]\\s\\s+/i'), '', \$var), preg_replace('/\\\\\\\\([0-9xu])/','/\\1', \$var), preg_replace('/&lt;br[^&gt;]*&gt;(\\n \\r)+/ims', '&lt;br&gt;', \$var), preg_replace('/&lt;br[^&gt;]*&gt;/i', '\\n', \$var), preg_replace('/&lt;[^&gt;]+&gt;/',' ', \$var), preg_replace('/&lt;+([a-z]+)/i', '\\1', \$var), str_replace('\\\\', '/', \$var), str_replace(array(':', ';', '@', '\\t', ' '), ' ', \$var), str_replace(array("\\r\\n", "\\r", "\\n"), " ", \$var), str_replace('&lt;&gt;', ' ', \$var), str_replace(" ", " ", \$var), str_replace('&lt; ', '__ltspace__', \$var), str_replace('&lt;:', '__ltspacepoints__', \$var), str_replace('__ltspace__', '&lt; ', \$var), str_replace('__ltspacepoints__', '&lt;:', \$var), str_replace(array('"', '/', '\\', ':', '*', '?', '\\n', '&lt;', '&gt;', ' ', '[', ']', ',', ';', '=', '°', '\$', '%', '&amp;#39;'), '\\', \$str), str_ireplace(array('javascript', 'vbscript', '&amp;colon', '&amp;#39;'), '', \$var), str_ireplace(array('..', '\\', '&amp;#38;', '&amp;#0000038', '&amp;#x26', '&amp;quot', '\\', '&amp;#34', '&amp;#0000034', '&amp;#x22', '&amp;#47', '&amp;#0000047', '&amp;#x2F', '&amp;#92', '&amp;#0000092', '&amp;#x5C'), '\\', \$var), strtr(\$var, array('&amp;#39;' =&gt; '__andamp__', '&amp;lt;' =&gt; '__andlt__', '&amp;gt;' =&gt; '__andgt__', '&amp;quot;' =&gt; '__dquot__')), strtr(\$var, array('__andamp__' =&gt; '&amp;', '__andlt__' =&gt; '&lt;', '__andgt__' =&gt; '&gt;', '__dquot__' =&gt; '"')), </pre> |
| Custom Functions <sup>1</sup> | outputResponse, js_escape, fixquotes, GETPOST, dol_html_entity_decode, sanitizerVal                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |

<sup>1</sup> Custom functions consist of at least one or more PHP built-in functions and sanitization rules.