PHPBench: Automated Generation of Verifiable and Hierarchical Benchmarks for PHP Web Fuzzing

Youkun Shi
Fudan University
Shanghai, China
The Hong Kong Polytechnic University
Hong Kong, China
youkun.shi@polyu.edu.hk

Yuan Zhang
Fudan University
Shanghai, China
yuanxzhang@fudan.edu.cn

Lei Zhang
Fudan University
Shanghai, China
23210240379@m.fudan.edu.cn

Jiarun Dai
Fudan University
Shanghai, China
jrdai@fudan.edu.cn

Tianhao Bai
Fudan University
Shanghai, China
23212010001@m.fudan.edu.cn

Fengyu Liu
Fudan University
Shanghai, China
23110240120@m.fudan.edu.cn

Bocheng Xiang
Fudan University
Shanghai, China
23210240090@m.fudan.edu.cn

Xiapu Luo
The Hong Kong Polytechnic University
Hong Kong, China
csxluo@comp.polyu.edu.hk

Min Yang
Fudan University
Shanghai Pudong Research Institute of Cryptology
Shanghai, China
m_yang@fudan.edu.cn

Abstract

Despite the rapid development of fuzzing techniques for PHP web applications, the community still lacks a unified, high-quality benchmark for rigorous evaluation. Existing evaluation methods suffer from imbalanced vulnerability distributions, the omission of hard-to-find bugs, and unverified ground truths, leading to skewed and unreliable evaluation results.

To address this, we introduce PHPBench, the first framework for automatically constructing verifiable and hierarchical vulnerability benchmarks for PHP web fuzzers. PHPBench's approach is threefold. First, it generates diverse vulnerability variants from real-world vulnerabilities by systematically weakening existing security protections, creating a varied distribution of exploitation complexity. Second, given a set of target PHP web applications, PHPBench further selects ideal code locations to insert the generated vulnerability variants, ensuring high realism. Finally, PHPBench automatically synthesizes verifiable Proofs-of-Concept (PoCs) to guarantee the exploitability of every inserted vulnerability.

In the evaluation, PHPBench successfully generated 375 unique vulnerability variants from merely 28 high-quality known vulnerability causes and inserted them into 10 popular PHP applications, covering 197 distinct functionalities. Each inserted vulnerability was verified to possess 100.00% exploitability while maintaining functional integrity. Finally, we utilize this curated fuzzing benchmark to evaluate seven existing fuzzing tools and reveal their systemic weaknesses, providing critical insights for future work and highlighting the limitations of current ad-hoc evaluation practices.

CCS Concepts

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

Keywords

Vulnerability Discovery, Web Fuzzing, Fuzzing Benchmark

ACM Reference Format:
Youkun Shi, Yuan Zhang, Lei Zhang, Jiarun Dai, Tianhao Bai, Fengyu Liu, Bocheng Xiang, Xiapu Luo, and Min Yang. 2026. PHPBench: Automated Generation of Verifiable and Hierarchical Benchmarks for PHP Web Fuzzing. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3832672

1 Introduction

Fuzzing is a powerful technique for automatically discovering software vulnerabilities and yielding definitive Proofs-of-Concept (PoCs). Given the widespread adoption of PHP for major websites [2] like Wikipedia [19] and WordPress [20], securing PHP web applications has become a critical concern. Hence, numerous work [4, 17, 18, 21, 33, 42, 50, 55–57, 63] have applied fuzzing to this domain, successfully identifying severe vulnerabilities such as Cross-Site Scripting, SQL Injection, and Command Injection.

Despite this innovation, progress in PHP web fuzzing remains severely hindered by the absence of a unified, high-quality benchmark for evaluating and comparing new techniques. Consequently, researchers resort to ad-hoc methods, including educational testbeds [33, 42, 57, 63], self-selected historical vulnerabilities [42, 55, 56, 63], or



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
CCS '26, The Hague, Netherlands.
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3832672

aggregated baseline results [50, 55, 57]. However, these approaches exhibit three fundamental limitations: First, the distribution of vulnerability difficulty in such datasets is often imbalanced, lacking a graduated hierarchy of flawed security protections. This creates a "capability cliff" where promising fuzzers that make incremental progress—such as bypassing basic but not advanced flawed protections—receive no credit, effectively masking their improvements and stifling innovation. Second, these datasets rarely contain vulnerabilities undetectable by existing tools, thereby obscuring shared technical bottlenecks. Third, dataset reliability is often unverified; even vulnerabilities with assigned CVEs may be invalid [47], which undermines fair comparison. The pivotal role of high-quality benchmarks is evident in the C/C++ fuzzing domain [29, 39, 41, 43, 45, 46, 62], where widely adopted standards like LAVA [29] have significantly accelerated technological progress.

To fill this gap, we aim to create the first framework dedicated to constructing high-quality benchmarks for PHP web fuzzing. A robust benchmark framework must satisfy four critical requirements: **Automation (R1)** for scalability and ease of use; **Realism (R2)** indicates that the causes of vulnerabilities and the applications in which they reside should align with real-world situations. **Hierarchy (R3)** requires providing a graduated spectrum of flawed protections for each vulnerability (e.g., ranging from unprotected to complex combinations of flawed protections). This is crucial to prevent the evaluation bias (i.e., vulnerabilities are either too trivial or impossible to detect) and enables fine-grained assessment of fuzzer capabilities; and **Verifiability (R4)**, ensuring every vulnerability is verifiably exploitable with a concrete PoC.

However, fulfilling these requirements is a non-trivial challenge, and existing benchmark construction techniques fall short. The first line of work [34, 39, 41] relies on manual construction by experts. While this approach ensures realism (R2) and verifiability (R4), it lacks automation (R1) and struggles to establish the required hierarchy (R3). The second line of work attempts to automate this process. Notable approaches like `FixReverter` [62] (reverting patches) and `VulGen` [43] (deep learning synthesis) have advanced the field. However, they are limited by (1) insufficient granularity in controlling the protection hierarchy (violating R3), and (2) failing to provide verifiable PoCs (violating R4). The final line of work [29, 46] aims to satisfy all the requirements. Theoretically, their two-stage strategy (i.e., generation and insertion) allows for flexible control over the hierarchy of generated vulnerabilities (R3) while ensuring their realism once inserted (R2). Furthermore, as the process is automated and provides triggering inputs as existence proofs, it satisfies R1 and R4. Nevertheless, these approaches are tailored to C/C++ programs and are inapplicable to web contexts.

Inspired by this, we design our benchmark construction framework with three core stages: generating hierarchical variants from real vulnerabilities, inserting them into realistic application locations, and verifying them via automatically adapted PoCs. However, realizing this design requires addressing two key challenges.

Challenge I: How to generate a rich hierarchy of vulnerability variants while ensuring our requirements? Automatically generating diverse variants is challenging. Intuitively, modifying vulnerability code pieces would easily compromise the code executability or vulnerability exploitability, rendering it invalid as a ground truth for fuzzing. Beyond this, the core difficulty lies in creating a meaningful hierarchy of challenges—from unprotected variants to those with multiple flawed protections—while ensuring a manageable process of adapting their verifiable PoCs.

Challenge II: How to insert variants into real-world applications to satisfy all of our requirements? Real-world applications often have a large codebase, creating a massive search space for potential insertion points. Identifying an ideal code location presents three main difficulties. First, to achieve realism, an ideal insertion point must be integrated into an application's execution flow—not at a superficial entry point—while remaining reachable. Second, the insertion must be sound, preserving the host application's original functionality to ensure the vulnerability remains accessible to fuzzers. Finally, to ensure the benchmark's correctness, each inserted vulnerability must be verifiably exploitable.

To address these challenges, we introduce `PHPBench`, a framework that automates the construction of high-quality benchmarks. To tackle the first challenge, `PHPBench` generates hierarchical variants by systematically weakening existing security protections identified via dynamic taint analysis. This controlled process preserves executability through minimal code modification and, critically, makes PoC adaptation tractable. To overcome the second challenge, `PHPBench` introduces a novel malicious probe-based dynamic analysis. It first uses dynamic exploration techniques to discover realistic user workflows and then replays requests with malicious probes to pinpoint deep, exploitable insertion points where user input remains active. Finally, `PHPBench` inserts the variants within the function's scope to prevent interference with the original context. It then synthesizes verifiable PoCs, confirming that each vulnerability is genuinely exploitable in its new context.

In the evaluation, we prototyped `PHPBench` and used it to construct a PHP web fuzzing benchmark from 10 popular PHP web applications frequently employed in prior research [31–33, 42, 55–57, 63]. During the vulnerability generation phase, `PHPBench`, using only 28 original and high-quality vulnerability causes, successfully identified 76 flawed security protections and generated 427 hierarchical vulnerability variants, with 375 unique ones. Furthermore, `PHPBench` generates an additional 88 adapted PoCs for variant confirmation. During the vulnerability insertion phase, `PHPBench` successfully identified 524 ideal insertion points within the dataset applications. Quality assessment indicated that these points were distributed across 293 application functionalities, achieving an average execution trace depth of 63.3%, which confirms their high quality. Subsequently, `PHPBench` selected 375 of these points to insert all generated vulnerability variants. Verification confirmed that the automatically generated PoCs ensured 100.00% exploitability. Regarding functional integrity, the inserted vulnerabilities involved 197 application functionalities. Manual verification confirmed 100.00% integrity, validating the effectiveness of our insertion strategy.

To demonstrate the utility of `PHPBench`-constructed benchmark, we evaluated seven well-known and open-source fuzzing (scanning) tools [4, 17, 18, 21, 42, 55, 56]. The evaluation shows that even the best-performing tool achieved only 28.45% recall. This further reveals critical limitations in existing techniques across all stages, including vulnerability exploration, confirmation, and even the criteria of bug oracles used for detection. Furthermore, our results demonstrate the superiority of `PHPBench` over conventional ad-hoc practices, which can produce misleading results. For example, one

fuzzer's recall on a popular ad-hoc testbed (DVWA [5]) can inflate performance metrics by over 20x (4.53% to 100%), highlighting the misleading nature of these practices.

We summarize the contributions of this paper as follows:

- We propose PHPBench, the first framework for automated construction of high-quality PHP web fuzzing benchmarks. Specifically, we feature novel techniques, including weakening-based variant generation with automated PoC adaptation and malicious probe-based deep insertion point identification.
- We evaluate PHPBench on 10 popular PHP applications for constructing a comprehensive benchmark. The evaluation results show that PHPBench successfully and automatically generated and inserted 375 unique, verifiable vulnerability variants, demonstrating its scalability and effectiveness.
- We evaluate seven existing fuzzing (scanning) tools on the PHPBench-constructed benchmark, offering critical insights and exposing the misleading nature of current ad-hoc evaluation practices.

2 Research Background

This section presents the research background, including existing PHP web fuzzing techniques (in §2.1), ad-hoc evaluation methods (in §2.2) and our vulnerability scope (in §2.3).

2.1 PHP Web Application Fuzzing

Fuzzing is a mainstream approach [4, 17, 18, 21, 32, 33, 42, 55–57, 63] to detect vulnerabilities in PHP web applications. These techniques are renowned for their capability to generate PoCs, ensuring high precision in vulnerability detection. We can broadly categorize them into the following two main lines.

Black-box Fuzzing (Scanning). This line of work [4, 17, 18, 21, 32] assumes that internal information (e.g., source code or database) about the target application is inaccessible. Instead, it starts with initial requests (e.g., the homepage of the web application) or requests targeting specific functionality. These requests are then mutated following predefined patterns to test for potential web vulnerabilities. Once a specific signal is triggered (i.e., bug oracle), it reports the vulnerability along with the PoC.

White/Grey-box Fuzzing. In contrast to black-box fuzzing, these techniques assume access to internal information of the target application. Several approaches [33, 42, 55, 56] implement instrumentation to collect code coverage information during the fuzzing process, using this feedback to guide code exploration. Additional approaches [50, 57, 63] expand on this by incorporating preliminary static analysis to identify potential source-sink paths, facilitating more effective directed fuzzing. Due to access to more internal information, these techniques can typically confirm vulnerabilities by observing crash signals from the interpreter.

2.2 Ad-hoc Evaluation Methods

The absence of a standardized benchmark for PHP web application fuzzing compels researchers to construct ad-hoc evaluation sets [33, 42, 50, 55–57, 63]. These methods, however, suffer from fundamental flaws that can lead to unreliable evaluation conclusions.

Using Educational-Purpose Testbeds. Researchers [33, 42, 57, 63] often employ open-source vulnerable applications like DVWA [5] for evaluation. However, these testbeds are designed for educational purposes and commonly have limited scale. Obviously, the patterns (i.e., root causes, triggering conditions, vulnerability locations, etc.) of implied vulnerabilities are quite simplifications of those found in real-world applications, causing inflated evaluation conclusions.

Using Self-Selected Historical Vulnerabilities. This method [42, 55, 56, 63] is also an ad-hoc practice that substitutes for a standard benchmark, where researchers select historical vulnerabilities from their application datasets for evaluation. The evaluation results are influenced by inconsistent selection factors such as the choice of applications and their versions, and selection criteria. This also makes it difficult to guarantee a balanced distribution of vulnerability causes, which can lead to over-representation.

Aggregating the Results of All Baselines. This approach [50, 55, 57] introduces bias as it creates a benchmark skewed toward common vulnerabilities that tools can find. Critically, it omits vulnerabilities missed by all baselines—which represent key challenges—and may include invalid reports [47], undermining evaluation accuracy.

2.3 Vulnerability Scope

We reviewed the vulnerability scope addressed by academic PHP web fuzzing techniques[32, 33, 42, 50, 55–57, 63] and focused on the three most representative vulnerability types, i.e., Cross-site Scripting (XSS) [10], SQL Injection (SQLi) [11], and Command Injection (CMDi) [9] vulnerabilities. These types are not only prevalent in real-world applications, but each surveyed literature also targets at least one of them, making them an ideal benchmark.

3 Research Consideration

In this section, we first introduce the requirements for a fuzzing benchmark construction framework (in §3.1) and then discuss the limitations of existing work (in §3.2).

3.1 Requirement Guidelines

A high-quality benchmark for PHP web fuzzing must be built upon a framework that meets four critical requirements.

R1: Automation. Manual benchmark construction is too labor-intensive to scale against evolving threats, the construction process must be automated to ensure scalability and minimize manual effort.

R2: Realism. To approximate real-world threats, the benchmark should prioritize realism in two aspects: (1) *Realistic Causes.* Vulnerabilities should originate from realistic developer mistakes rather than purely synthetic code [48, 54]. (2) *Realistic Locations.* Vulnerabilities should be embedded deep within application workflows rather than at superficial entry points or in isolated snippets [1, 43], thereby challenging a fuzzer's ability to explore the application.

R3: Hierarchy. As previously noted, an imbalanced distribution of vulnerabilities can skew evaluation conclusions. To enable a more accurate comparison of fuzzer capabilities, a benchmark must contain a hierarchical set of vulnerability instances. These instances should span a range of complexities, from being entirely unprotected to being guarded by multiple, yet flawed security protections.

R4: Verifiability. Each vulnerability in the benchmark must be explicitly verifiable. Prior research [47, 49] has highlighted that incorrect labeling—where non-exploitable bugs are marked as vulnerabilities—remains a critical issue in existing datasets. Thus, we require that each vulnerability be guaranteed via a verifiable PoC.

3.2 Limitation of Existing Work

Building a benchmark that satisfies all of our requirements is a non-trivial task. Existing work [29, 34, 39, 41, 43, 45, 46, 62], while informative, falls short and can be classified into three types.

Expert-Dependent Manual Construction. These methods rely on security experts to use applications with known bugs [39, 41] or to manually backport historical vulnerabilities [34]. While ensuring realism (R2) and verifiability (R4), this process inherently lacks automation (R1), cannot produce a hierarchy of challenges (R3), and scales poorly due to the extensive manual effort.

Automated Vulnerability Introduction or Synthesis. These methods attempt to automate vulnerability introduction, exemplified by heuristics like code reverting [45, 62] or deep learning-based synthesis [43]. While these approaches advance the field, they lack the granularity to control the hierarchy of flawed protections (violating R3). Crucially, they fail to synthesize verifiable PoCs to guarantee exploitability (violating R4).

Application-Targeted Vulnerability Insertion. This category, separating benchmark construction into *vulnerability (bug) generation* and *application insertion*, is the most flexible and promising. However, as these works [29, 46] target C/C++ programs, they possess significant limitations for web applications. (1) For generation, LAVA [29] supports only simple memory-corruption errors, while Apocalypse [46] focuses on functional bug synthesis (e.g., triggering `assert(false)`) rather than generating security vulnerabilities. (2) For insertion, both works rely exclusively on low-level instruction traces (e.g., via symbolic execution), which are inherently blind to high-level dynamic web behaviors (e.g., routing or UI interactions).

4 Overview

In this section, we will first introduce our high-level ideas (in §4.1), and then summarize the key challenges (in §4.2).

4.1 High-level Ideas

As discussed in §3.2, the high-level idea of application-targeted vulnerability insertion is more favorable for achieving our goals and requirements. Therefore, our approach can be divided into three main stages: (1) generating hierarchical vulnerability variants from original ones, (2) appropriately inserting them into target applications, and (3) verifying their exploitability using PoCs during both the generation and insertion stages. Specifically, the first stage involves automatically generating hierarchical variants based on the root causes of known vulnerabilities that reflect real developer mistakes. The second stage identifies ideal insertion points within real-world applications and seamlessly integrates the generated vulnerabilities into their workflows. The third stage validates the preceding phases by adapting and synthesizing verifiable PoCs, thereby ensuring exploitability. In conclusion, the end-to-end process is fully automated, satisfying R1 (Automation). The first and second stages ensure R2 (Realism) and R3 (Hierarchy) are met, while the third stage guarantees R4 (Verifiability).

4.2 Challenges

Although the high-level idea is appealing, we need to address the following two key challenges to achieve it.

Challenge I: How to generate a rich hierarchy of vulnerability variants while ensuring our requirements?

(1) *Executability Preservation:* since the variant generation involves code modifications from the original ones, it is essential to ensure the syntactical integrity of the variants. This is particularly critical during code deletion operations, as they may introduce issues (e.g., undefined variables) that can impede executability.
(2) *Exploitability Preservation:* the variant generation inevitably involves modifications to existing security protections, which should be handled carefully to avoid affecting the exploitability.
(3) *Verifiable PoC Adaptation:* the data/control-flow of vulnerability variants may have changed compared to the original ones, potentially rendering the original PoC ineffective. This requires that the proposed approach automate the PoC adaptation to accommodate these changes, which is a challenging task [28].

Challenge II: How to insert variants into real-world applications to satisfy all of our requirements?

(1) *Appropriate Insertion Point Identification:* A primary challenge is searching the vast code space for insertion points that balance both realism and exploitability, as these goals often conflict. For example, placing a bug near a program's entry point (e.g., after `$_GET` in PHP), ensures exploitability but lacks realism.
(2) *Sound Vulnerability Insertion:* The insertion process must be semantically sound. A flawed insertion can disrupt the application's logic, potentially making the new vulnerability unreachable by the fuzzing tools under evaluation. This can lead to false negatives and skew the conclusions.
(3) *Verifiable PoC Synthesis:* The framework must automatically synthesize a verifiable PoC for each inserted vulnerability. The PoC must verify exploitability by demonstrating a path to both reach and trigger the vulnerability.

5 Approach of PHPBench

In this section, we introduce PHPBench, an automated benchmark construction framework, which addresses the aforementioned challenges through two key modules: (1) hierarchical variants generation module (§5.1), and (2) application-targeted vulnerability insertion module (§5.2). Figure 1 presents the overall architecture.

5.1 Hierarchical Variants Generation

To address challenge I, this part first introduces the definition of hierarchical variants. Then, we elaborate on how this module ensures the executability and exploitability of the variants and automatically adapts the PoCs for validation.

5.1.1 Hierarchical Variant Definition. First, we define a hierarchical vulnerability variant. Generally, our in-scope vulnerabilities occur when untrusted input (i.e., source) flows to a security-sensitive function (i.e., sink) with flawed security protections along the path, including inadequate control-flow constraints and data-flow sanitizers. We formally define their causes V with the tuple:

$$V = \langle S, K, \mathbb{P} \rangle$$

- S is the program point representing the *Source* where untrusted data controlled by users.
- K is the program point representing the *Sink* where a security-sensitive function is performed.

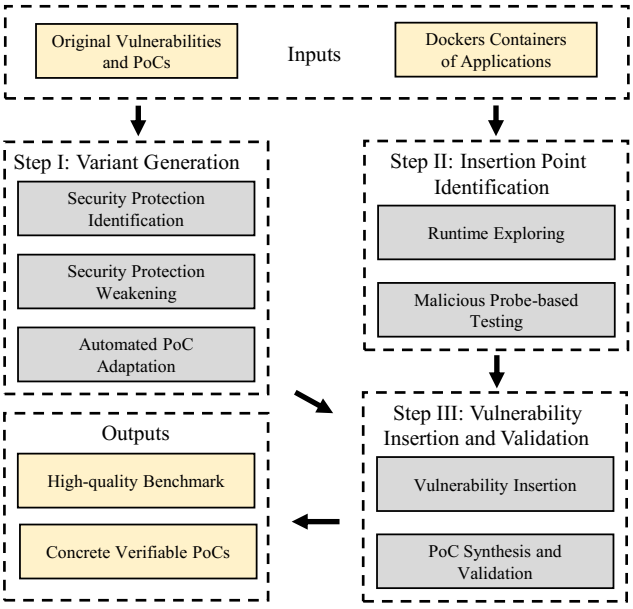

Figure 1: The workflow of PHPBench.

- $\mathbb{P} = \{p_1, p_2, \ldots, p_n\}$ is the set of flawed security protections applied to the path from S to K. This set can be empty ($\mathbb{P} = \emptyset$), indicating a complete lack of protection.

Let V_A and V_B be two vulnerabilities with root causes $V_A = \langle S_A, K_A, \mathbb{P}_A \rangle$ and $V_B = \langle S_B, K_B, \mathbb{P}_B \rangle$ respectively. We state that V_A **is a hierarchical variant of** V_B, denoted as $V_A \preceq V_B$, if they share the same source and sink, and the security protections for V_A are a subset of those for V_B. Formally:

$$V_A \preceq V_B \iff (S_A = S_B) \wedge (K_A = K_B) \wedge (\mathbb{P}_A \subseteq \mathbb{P}_B)$$

5.1.2 Hierarchical Variant Generation. Next, we introduce how PHPBench generates hierarchical variants from original causes.

Design Considerations. In fact, there are two ways to generate hierarchical variants from an original vulnerability: (1) strengthening the security protections, and (2) weakening the existing protections. Here, PHPBench only selects the latter. The rationale is that controlling the exploitability of variants created by strengthening protections is challenging. This method requires precisely controlling the interaction between new and existing protections to ensure the vulnerability persists. In contrast, variants generated by weakening existing protections have deterministic exploitability, which also makes the automated PoCs adaptation more manageable.

General Procedure. Taking the code snippets of the original vulnerabilities and their corresponding PoCs, PHPBench automatically generates hierarchical vulnerability variants with PoCs in three steps. First, PHPBench performs dynamic taint propagation through diverse malicious probes to automatically and accurately identify security protections within the code snippets of the original vulnerabilities. Then, for different types of the identified protections (i.e., on data- or control-flow), PHPBench applies different weakening strategies to ensure the executability of variants. Finally, PHPBench uses the original PoCs to validate the exploitability of the variants. For those that fail, PHPBench attempts to perform automated adaptation to make them work. The details are as follows.

Step I: Security Protection Identification. First, we describe how PHPBench automatically identifies security protections within original vulnerabilities. Unlike efforts that rely on manual modeling [23, 24, 36, 40, 52], we employ dynamic analysis. The rationale is that the manual approach is not only labor-intensive but also prone to false negatives with developer-defined methods [51, 60], which can reduce the hierarchy of the original vulnerabilities.

Specifically, our method analyzes the original vulnerability's code snippet in a runnable PHP environment. We designate the source as a tainted variable and perform dynamic taint propagation [6] using both normal and malicious inputs from a comprehensive test suite [8]. During execution, we identify two types of protections. First, we record data-flow statements that alter the tainted variable only under malicious input, flagging them as data-flow sanitizers (D_p). Second, we identify conditional statements where the execution path diverges between normal and malicious inputs, marking them as control-flow protections (C_p). Since triggering protections is far simpler than bypassing them, we consider such an approach to be sufficiently accurate and effective in identifying the full set of flawed protections.

Step II: Security Protection Weakening. To ensure the executability of the code variants, PHPBench carefully weakens the identified protections. It generates a hierarchy of variants by performing permutations and combinations on the statements within C_p and D_p. This process creates a spectrum of variants, ranging from the original vulnerability with all protections intact (no statements removed) to a basic version with all statements in C_p and D_p removed.

Since any code removal has the potential to undermine executability, PHPBench adopts different strategies for removing security protections within C_p and D_p. ❶ *For* C_p, PHPBench adopts a minimal-modification strategy, inverting the logic of the conditional statements by adding `|| True` or `&& False`, depending on the correct flow direction of normal inputs. ❷ *For* D_p, a more fine-grained Abstract Syntax Tree (AST) modification is required, as sanitizers are often commingled with essential formatting functions (as shown in Figure 2), and their naive removal could lead to execution failures. Thanks to dynamic taint propagation analysis, PHPBench can perform function-level analysis. By comparing the values of a tainted variable before and after a function call, PHPBench can differentiate security sanitizers. It then uses PHP-parser [7] to convert the target code into an AST, surgically removes the nodes corresponding to the identified sanitizers, and finally reconstructs the weakened statement from the modified AST.

Step III: PoC Adaptation and Validation. Finally, we detail how PHPBench automates the adaptation of original PoCs to verify the exploitability of vulnerability variants. As discussed before, generating a PoC from scratch is a non-trivial task, particularly for vulnerabilities protected by multiple flawed security protections. Our approach, by contrast, makes this process tractable by strategically focusing on the specific protections that were weakened. This design choice reduces the verification task to reusing or adapting the existing PoC for the modified protection, thus bypassing the complexity of generating a new exploit from scratch.

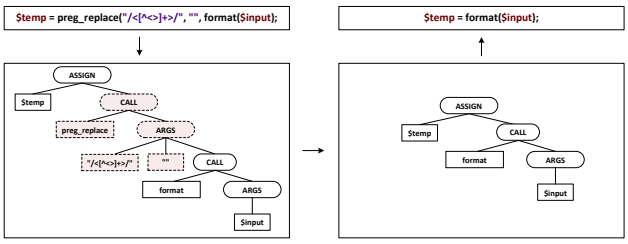

Figure 2: Example of AST-based modification strategy.

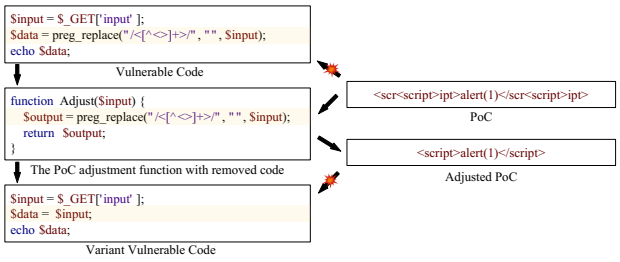

Figure 3: Example of the automated PoC adaptation.

Specifically, we can categorize PoCs into two main types based on how they evade security protections:

- *Exploiting via Edge Cases:* These PoCs exploit blind spots in security protections by crafting payloads that do not match predefined rules. For example, an attacker can circumvent a keyword-based blacklist by using an equivalent but unlisted term.
- *Exploiting via Implementation Flaws:* These PoCs target unintended behaviors in the implementation of security protections. Attackers submit a specially crafted input that the flawed protection inadvertently transforms into a malicious payload. A classic example is bypassing a flawed XSS sanitizer with a double-encoded payload like `<scr<script>ipt>`. The sanitizer's single-pass removal of the inner `<script>` tag unintentionally reconstructs the malicious outer tag.

Clearly, the two types of PoCs respond differently when their targeted protection is removed. An edge-case PoC, designed to evade the protection, remains effective. In contrast, an implementation-flaw PoC becomes ineffective and must be adjusted, as its exploit depends on the very flaw in the now-absent protection.

Therefore, for PoCs that exploit implementation flaws, PHPBench uses a novel adaptation strategy after the targeted security protection is removed. Rather than analyzing the remaining code, our approach leverages the logic of the removed protection itself. The rationale is that the original PoC was crafted to be transformed into a malicious payload by the flawed protection. PHPBench simulates this transformation by processing the original PoC through the removed code; the output becomes the new, adapted PoC for the vulnerability variant. This process is illustrated in Figure 3. In the example, a variant is created by removing a custom `preg_replace()` sanitizer, which renders the original double-encoded PoC ineffective. To adapt the exploit, PHPBench encapsulates the removed sanitizer code within a temporary function that simulates the original flaw. It then passes the original PoC to this function, and the return value serves as the new, valid PoC for the variant.

Finally, PHPBench places the vulnerability variant code into a runnable PHP environment and verifies its exploitability with the PoCs and bug oracles [31, 55]. This ensures that each vulnerability variant passed to subsequent steps is executable and exploitable.

5.2 Application-Targeted Insertion

To address Challenge II, this part details how PHPBench identifies ideal insertion points, inserts hierarchical variants into real-world applications, and synthesizes PoCs for verification.

Design Considerations. As discussed in §4.2, this process entails three key tasks: identifying ideal insertion points, soundly inserting vulnerability variants, and synthesizing verifiable PoCs to ensure exploitability. Specifically, ❶ for the first task, static analysis is ill-suited due to the complex routing and dynamic nature of web applications [24, 35, 40]. Instead, we employ a novel dynamic method. This approach uses web crawlers to simulate user interaction, discovering reachable code locations in realistic workflows, and then applies malicious probe-based dynamic tracking to pinpoint deep, exploitable insertion points. ❷ For the second task, to ensure realism and "fuzzer-visibility", the vulnerability's source must be an original application variable. However, direct modification could disrupt application functionality. We therefore adopt a minimal modification strategy that encapsulates the vulnerability logic within a separate function, passing original variables as parameters. This confines data flow changes to a local scope, preserving the application's integrity. ❸ For the last task, a verifiable PoC must contain both a request to navigate to the vulnerability and a payload to trigger it. Our dynamic approach inherently captures the required requests during insertion point identification, while the variant generation stage provides the necessary payloads. By combining these two components, we construct a verifiable PoC to validate the exploitability of each inserted vulnerability.

General Procedure. The overall process can be divided into five main steps. First, PHPBench performs code instrumentation on the target application to facilitate the collection of necessary traces. Second, PHPBench employs existing web crawling techniques to simulate user interaction, collecting requests and corresponding traces under realistic scenarios. Third, for these collected requests, PHPBench uses malicious input as probes for their parameters. By performing dynamic taint propagation analysis, it monitors their value within the trace to identify exploitable and ideal insertion locations. Fourth, PHPBench randomly selects insertion points for each hierarchical variant. It then inserts the vulnerability by encapsulating the code within a function, following a minimal modification strategy. Finally, PHPBench synthesizes the final concrete PoC by incorporating the variant's PoC into the relevant request, which is used to verify the exploitability of each inserted vulnerability.

Step I: Instrumentation. To identify reachable code regions and exploitable request parameters, PHPBench performs malicious probe-based dynamic taint analysis. Specifically, it first employs basic block-level AST instrumentation [56] to capture execution code traces. Second, to enable dynamic taint propagation, PHPBench extends the Taint extension [6] by hooking PHP assignment opcodes (`ZEND_ASSIGN`, `ZEND_ASSIGN_DIM`, `ZEND_ASSIGN_OBJ`). When a

right-hand operand is tainted, PHPBench records the operand, the variable name, and the corresponding file and line number.

Step II: Runtime Requests and Traces Collection. Next, PHPBench employs existing crawling techniques to simulate user interactions, recording requests and code traces under realistic scenarios. However, dynamic exploration inevitably generates some duplicate or invalid requests [31, 44]. Given that these collected requests must be replayed in subsequent steps, PHPBench filters out this low-quality data to enhance overall efficiency:

- *Invalid Requests:* PHPBench removes requests without any execution traces. These typically occur because dynamic exploration may construct requests for common directories that may not exist in the target application.
- *Duplicate Requests:* PHPBench removes duplicate requests that have exactly identical execution traces. These requests typically originate from the same URLs but different parameter values (e.g., `?NewsID=1` and `?NewsID=2`).
- *Parameter-Useless Requests:* PHPBench removes these requests as their parameters are uncontrollable. For example, the request that contains parameters not utilized within the execution trace.

Step III: Malicious Probe-Based Insertion Point Identification. In this step, PHPBench analyzes the filtered requests and their execution traces to identify ideal insertion points for vulnerability insertion. An ideal insertion point is a statement in the code that meets three criteria: (1) a variable influenced by external input remains tainted after its execution; (2) this tainted variable has not yet been sanitized or validated by the application's security mechanisms; and (3) it is the last statement within the trace that satisfies the preceding two conditions. The final criterion is crucial, as it ensures that vulnerabilities are embedded deep within the application's execution flow rather than at superficial locations like the program entry.

To locate such points, PHPBench performs dynamic taint analysis using malicious probes. For each execution trace, it injects PoCs of vulnerability variants as malicious inputs for the request parameters. PHPBench then taints these inputs and tracks their propagation. The taint propagation for any variable is halted if its value is modified or sanitized, as this conservatively indicates the end of a potential exploit path. Finally, from all statements that meet the criteria within a trace, PHPBench selects the one that occurs latest in the execution sequence as the definitive insertion point.

Step IV: Vulnerability Insertion. In this step, PHPBench inserts the generated hierarchical variants into the target application. As discussed before, the insertion process must preserve the original application functionality to prevent any failures that could impede the exploration of fuzzing tools. Therefore, we chose to encapsulate the vulnerable code into a function and then inserted the function into the selected insertion points. This approach allows us to not only utilize the application's native variables (by passing them as function parameters) but also ensure that all inserted code remains confined to the function's scope. This prevents any interference with the value space of the original program's variables.

As an illustration, Figure 4 (a) shows how PHPBench encapsulates a hierarchical vulnerability variant into a function. This function consists of three parts: (1) Function Name: the name is generated by concatenating the prefix `vuln_function` with the first six characters

```
1  function vuln_function_e9db56($var) {
2  -   $input= $_GET['input'];
3  +   $input= $var;
4      $data= preg_replace("/<[^<>]+>/", "", $input);
5      echo $data;
6  }
```

(a) Encapsulation function for hierarchical vulnerability variant

```
1  <?php
2  +include_once 'vuln_function_e9db56.php';
3   $keyword= (isset($_GET['keyword'])) ? $_GET['keyword'] : '';
4   $action= (isset($_GET['action'])) ? $_GET['action'] : '';
5   $args= array('action' => $action, 'keyword' => $keyword, ...);
6   ... // Processing
7   if ($args['action'] == 'search') {
8      $args['keyword'] = (string)$args['keyword'];
9      $sql = "SELECT * FROM comments".
10                          "WHERE content LIKE ? LIMIT 10";
11     $stmt = $pdo->prepare($sql);
12 +   vuln_function_e9db56($args['keyword']);
13     $searchKeyword= "%" . stripchars($args['keyword']) . "%";
14     $stmt->execute([$searchKeyword]);
15     ... // Fetch and display results
16  }
17  ?>
```

(b) Vulnerability variant insertion with minimal modifications

Figure 4: Example of vulnerability variant insertion with function-based minimal modification.

of the MD5 hash of the variant's generation timestamp. (2) Function Parameters: these correspond to the source variables within the vulnerability variant (e.g., `$var` in line 1). (3) Function Body: the body contains the variant's code, which PHPBench slightly modifies for encapsulation. Specifically, original source variables are replaced with function parameters (e.g., replacing `$_GET['input']` with `$var` in lines 2 and 3). Each encapsulated function is saved as a separate file named after the function itself (e.g., `vuln_function_e9db56.php`).

Then, PHPBench randomly pairs the encapsulated vulnerability variant functions with the selected insertion points and performs the insertion. Specifically, as shown in Figure 4 (b), the preceding step selected line 12 as an insertion point, where the corresponding controllable parameter is `$args['keyword']`. This parameter is tainted from `$_GET['keyword']` in line 3, and the malicious data propagation is terminated at line 13 by being sanitized by `stripchars()`. To perform the insertion, PHPBench makes two key modifications to the program: (1) It adds a file include statement (e.g., line 2) at the beginning of the target file to import the encapsulated function. (2) It inserts the encapsulated function at the insertion points, passing the controllable variable as a parameter (e.g., line 12).

Step V: Verifiable PoC Synthesis and Validation. In the final step, PHPBench synthesizes a concrete PoC to verify the exploitability of the inserted vulnerability. As we previously discussed, PHPBench creates these concrete PoCs by combining the reachable requests obtained during the insertion point collection phase with the PoCs of the vulnerability variants achieved in §5.1.2. This synthesis results in a final PoC that can both navigate to the vulnerability's location and successfully trigger the exploit. Figure 5 illustrates a detailed example of PoC synthesis. Subsequently, PHPBench uses

these concrete PoCs along with bug oracles [31, 55] to verify the exploitability of the inserted vulnerabilities.

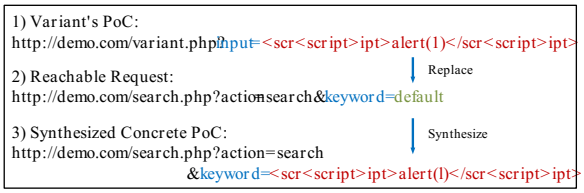

Figure 5: Example of the final verifiable PoC synthesis.

6 Evaluation

This section evaluates PHPBench's effectiveness and efficiency in benchmark construction, and shows its utility and superiority.

Implementation. We have implemented a prototype of PHPBench. Overall, it comprises 3,580 lines of Python code for the main program, 331 lines of C code for dynamic taint analysis instrumentation, and 1,090 lines of PHP code for AST-based operations and target application code instrumentation. Moreover, for the insert point collection phase, to ensure comprehensive exploration, we employed eight well-known and easily set up open-source dynamic tools. These include four tools from the open-source community (i.e., w3af [17], ZAP [21], wapiti [18] and Arachni [4]) and four academic prototypes (i.e., webFuzz [56], Black Widow, Witcher [55] and PHUZZ [42]). We configured their crawling phase according to their guidelines, with details presented in Appendix-Appendix A. Additionally, for tools supporting automatic authentication, we provided valid credentials for each application. The runtime for each crawling was limited to 8 hours per application.

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

- RQ1: How effective is using PHPBench to construct the benchmark? (in §6.1)
- RQ2: How efficient is using PHPBench to construct the benchmark? (in §6.2)
- RQ3: What is the utility of the PHPBench-constructed benchmark for evaluating existing fuzzing tools? (in §6.3)
- RQ4: How superior is the PHPBench-constructed benchmark compared to ad-hoc practices? (in §6.4)

Original Vulnerability Cause Collection. To ensure the quality of the generated vulnerability variants, we aim to select high-quality, known vulnerability causes. We began by crawling the National Vulnerability Database (NVD) for patched PHP vulnerabilities (XSS, SQLi, CMDi) reported between 2019 and 2024, yielding 236 CVEs. Using an existing technique [53], we then automatically extracted root-cause code snippets for 189 of these. This set was further refined to 83 CVEs by excluding cases where security protections were absent, retaining only those with flawed implementations. From this pool, we manually selected a final, balanced set of 28 CVEs (listed in Appendix-Appendix B). These causes were chosen to be mutually exclusive—ensuring no cause's set of flawed protections was a subset of another's—a crucial property that enables PHPBench to generate a wider range of unique variants. Each selected cause was then prepared as an executable code snippet with a corresponding PoC. The entire process required about five hours.

Table 1: Distribution of hierarchical variants by number of flawed security protections and vulnerability types.

# Flawed Protections Num.	# XSS Num.	# SQLi Num.	# CMDi Num.
$\mathbb{P} = \emptyset$	10	10	8
$0 < \lvert\mathbb{P}\rvert \leq 2$	108	36	36
$2 < \lvert\mathbb{P}\rvert \leq 4$	155	16	8
$\lvert\mathbb{P}\rvert \geq 5$	39	1	0
Total (Uniques)	312 (278)	63 (52)	52 (45)

Real-world Application Dataset. For insertion purposes, we selected projects commonly utilized in the evaluation of prior work [31–33, 42, 55–57, 63]. Specifically, we chose the top 10 applications based on their usage frequency, as detailed in Table 2.

Environment 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 236 GB memory. Each target application was run in a Docker container under Apache 2.4.41 and PHP 7.4.3.

6.1 RQ1: Effectiveness of PHPBench

In this experiment, we evaluate the effectiveness of PHPBench in constructing a benchmark across our dataset. This experiment consists of three key parts: (1) the effectiveness of vulnerability variant generation (in §6.1.1), (2) insertion point identification (in §6.1.2) and (3) vulnerability insertion with verification (in §6.1.3).

Result Overview. Regarding the hierarchical variant generation stage, PHPBench successfully generated 427 vulnerability variants, of which 375 were unique, as detailed in Table 1. During the exploitability verification process, for 88 of these variants where the original PoCs failed, PHPBench automatically adjusted the PoCs and completed the verification. Regarding the vulnerability insertion stage, PHPBench first successfully identified 524 insertion points within the 10 applications, and then randomly selected 375 of these for inserting the unique variants. The details are presented in Table 2, PHPBench successfully finished the vulnerability insertion and verified them using synthesized concrete PoCs.

6.1.1 Hierarchical Variants Generation Details. PHPBench automatically identified 76 flawed security protections stemming from 28 original vulnerability causes. We manually confirmed that all identified protections were correct, of which 71 were unique, comprising 46 insecure data-flow sanitizers and 25 insecure control-flow constraints. Among these unique flawed protections, we found that 14 utilized PHP's built-in sanitizer functions (e.g., `htmlspecialchars()`), while 57 were developer-defined (e.g., custom rules implemented via `preg_replace()`). This underscores the importance of PHPBench's use of automated security protection identification rather than experience-based modeling.

Then, by applying weakening-based variant strategies, PHPBench successfully generated 427 hierarchical vulnerability variants. During this process, all generated variants were verified to be executable (i.e., capable of running in a runtime environment), which highlights the effectiveness of our minimal-modification strategy. Furthermore, we analyzed the number of flawed protections in the

Table 2: Breakdown of results from each step of the application-targeted insertion stage across our dataset.

Applications	LoC	Insertion Point Identification			Quality of Insertion Points		Vulnerability Insertion & Verification	
		Collected Requests	Qualified Requests	Insertion Points	Functional Diversity	Execution Trace Depth (Avg.)	Vulnerability Insertion	<Functionality, PoC> Verification (Pass Rate)
WordPress	587,187	8,412	435	145	80	3560.53 (71.87%)	80 / 145	<46, 80> (100.00%)
OpenEMR	1,788,910	101,036	226	96	41	2318.01 (55.81%)	89 / 96	<37, 89> (100.00%)
Joomla	564,737	12,521	642	40	21	1856.92 (63.66%)	27 / 40	<12, 27> (100.00%)
PrestaShop	2,044,183	25,551	332	56	39	4909.09 (74.97%)	19 / 56	<10, 19> (100.00%)
Drupal	56,847	21,730	963	27	20	50.11 (92.10%)	26 / 27	<19, 26> (100.00%)
Piwigo	244,270	5,104	226	33	22	414.62 (56.13%)	20 / 33	<12, 20> (100.00%)
MyBB	161,351	7,314	386	46	28	572.36 (75.76%)	42 / 46	<26, 42> (100.00%)
Hosp. Mgmt.	14,352	4,649	73	44	20	11.39 (95.71%)	36 / 44	<14, 36> (100.00%)
Doctor Appt.	4,530	5,274	33	15	9	6.73 (72.60%)	14 / 15	<8, 14> (100.00%)
Login Mgmt.	11,074	4,319	30	22	13	12.5 (96.53%)	22 / 22	<13, 22> (100.00%)
Total	5,477,441	195,910	3,346	524	293	1972.33 (63.33%)	375 / 524	<197, 375> (100.00%)

causes of these vulnerability variants, with the distribution detailed in Table 1. We observed that XSS vulnerabilities in the original code tend to involve a greater number of flawed security protections. Consequently, our strategy generated a larger set of variants for this vulnerability type. Finally, we identified 375 unique variants, each with a distinct set of flawed protections. These results highlight the effectiveness of our hierarchical design, which addresses the issue of over-representation caused by the uneven distribution of vulnerability causes. During the vulnerability variant validation, PHPBench discovered that 88 variants could not be confirmed by the bug oracle using the original PoCs. Therefore, PHPBench constructed corresponding adjustment functions to modify the PoCs based on the removed flawed protections. It successfully generated 88 adjusted PoCs, which triggered the vulnerabilities in these variants for confirmation.

6.1.2 Insertion Point Identification Details. As detailed in Table 2, PHPBench first explored the target applications using dynamic exploration techniques and successfully collected a total of 195,910 requests. Subsequently, these requests were filtered, resulting in 3,346 qualified requests after removing 122,410 invalid, 62,318 duplicate, and 7,836 parameter-useless requests. For the 3,346 qualified requests, PHPBench used the PoCs from the previously generated variants as malicious input probes, individually assigning them to the parameters and replaying the requests. Using dynamic taint analysis, this process identified a total of 828 insertion points. Upon further deduplication, 524 of these were unique, indicating that different requests resulted in the same insertion points due to overlaps in their traces. Additionally, we performed a sampling analysis on requests lacking ideal insertion points (i.e., sampled 100 requests). We found that the reasons were twofold: (1) controllable parameters were globally sanitized (11 cases); (2) The values of controllable parameters were constrained by control flow expectations and could not be assigned malicious values, which included 20 cases related to functionality checks and 69 cases for security checks.

Furthermore, as noted in prior work [29], assessing the realism of insertion locations is difficult. Consequently, we adopt an alternative approach by evaluating the quality of insertion points based on two criteria: Functional Diversity and Execution Trace Depth. The former measures the breadth of application functionalities that the insertion points reside in, while the latter quantifies the insertion depth within the execution trace of each functionality request (i.e., the relative position of the basic block containing the insertion point within the execution trace sequence). The evaluation details are presented in Table 2. The results show that the 524 insertion points identified by PHPBench cover 293 functionalities across 10 applications. Feature-rich applications like WordPress achieved coverage of 80 functionalities, while simpler applications, such as Doctor Appt., still reached 9. Regarding insertion depth, the average depth of these insertion points within the traces was 63.33%. This demonstrates that our strategy avoids selecting shallow locations and instead integrates more deeply into the application functionalities. Besides, we observe that the absolute trace depths for 3 applications (Hosp. Mgmt., Doctor Appt., and Login Mgmt.) were relatively low, which can be attributed to the inherent nature of their limited LoC and complexity.

6.1.3 Vulnerability Insertion & Verification Details. In the next step, PHPBench encapsulates each variant of a unique vulnerability cause into a function for application-targeted insertion. Specifically, PHPBench generated 375 encapsulated functions, totaling 23,352 lines of code (LoC) and averaging 62 LoC per function. The largest of these contains 107 LoC, as its cause involves five developer-defined functions, while the smallest has 8 LoC, representing a vulnerability variant with no security protections. Then, PHPBench randomly selected 375 out of 524 unique insertion points for vulnerability insertion, the distribution is detailed in Table 2. Finally, PHPBench automatically synthesizes a verifiable PoC for each inserted vulnerability. Specifically, it selects the HTTP requests that reach the insertion points, and then replaces the controllable parameters in these requests (totaling 256 POST and 119 GET) with the corresponding vulnerability variant's PoCs. Subsequently, by executing the synthesized PoCs, PHPBench verified that all 375 inserted vulnerabilities were successfully triggered, confirming 100% exploitability for the benchmark.

Besides, we also manually verified the preservation of the application functionality of each inserted vulnerability. The results indicate that the 375 vulnerabilities inserted by PHPBench cover 197 functionalities across 10 applications. Following [52], we replayed benign requests against these functionalities post-insertion and compared the responses to those from pre-insertion requests. While 45 responses were identical, the remaining 152 differences were confirmed as benign deviations stemming from expected vulnerability contexts (46 cases, e.g., XSS-introduced HTML tags) or dynamic values (106 cases, e.g., CSRF tokens and timestamps). This confirms that PHPBench maintains 100% functional integrity for the inserted vulnerability and highlights the effectiveness of our function-based minimal modification insertion strategy.

6.2 RQ2: Efficiency of PHPBench

In this experiment, we evaluate the end-to-end efficiency of PHPBench in constructing the benchmark. Overall, excluding the 8-hour time limit per application for each dynamic exploration technique, PHPBench spent a total of 2.39 hours to complete the benchmark construction across our dataset. Specifically, during the vulnerability variant generation phase, PHPBench spent 13.37 minutes, including 5.65 minutes for identifying security protections, 1.05 minutes for variant generation, and 6.67 minutes for PoC adaptation and vulnerability validation. In the vulnerability insertion phase, PHPBench spent 130.12 minutes, including 114.88 minutes for insertion point identification and 15.24 minutes for vulnerability insertion and verification. In conclusion, we consider that efficiency is acceptable for constructing a high-quality benchmark.

6.3 RQ3: Utility of PHPBench

In this experiment, we evaluate existing techniques on the PHPBench-constructed benchmark to demonstrate its utility.

Under-test Tools. We selected seven open-source black-box or gray-box dynamic techniques[1] capable of detecting (or being extended to detect) XSS, SQLi, and CMDi, as detailed in Table 3. Among them, we extended webFuzz [56] (namely webFuzz+), which natively supported XSS and SQLi detection, to also identify CMDi vulnerabilities by incorporating the testcase generation and oracle from Witcher [55]. We also employ Witcher+, which natively supports SQLi and CMDi detection and was enhanced for XSS detection by an existing work [57]. Furthermore, Wapiti+ is a modified version of Wapiti [18] in which the bug oracles for SQLi and CMDi are replaced with those from Witcher. The specific reasons are detailed in §6.3.3. Each technique was evaluated on each application for 3 rounds, with each round limited to 24 hours (follows [37, 47]).

Result Overview. The detailed comparison results are presented in Table 3. In terms of insertion point coverage, Witcher+ performs best, successfully exploring a three-round average of 214.33 points with a coverage rate of 57.15%. Regarding vulnerability detection, Figure 6 illustrates the Time-to-Exposure (TTE) for each baseline. In contrast, black-box scanners employ a parallel exploration and vulnerability confirmation strategy, whereas gray-box fuzzers adopt an explore-then-confirm approach, prioritizing URL crawling before sequentially fuzzing the collected URLs. In conclusion, all tools converged within 24 hours, Wapiti+ identified the highest average number of vulnerabilities (i.e., 106.67) across three trials, yet its recall rate remained below 30%.

[1]We excluded Atropos as its repository confirms it's "hard to set up currently" [14].

Table 3: Performance comparison of existing work on the constructed benchmark. Values denote mean (± standard).

Tool	Reached Insertion Points		Detected Vulnerabilities	
	Num. (± std)	Recall (%)	Num. (± std)	Recall (%)
webFuzz+	130.00 (±2.00)	34.67%	26.67 (±3.21)	7.11%
Witcher+	214.33 (±10.97)	57.15%	41.33(±3.51)	11.02%
PHUZZ	90.67 (±2.52)	24.18%	17.00 (±1.00)	4.53%
Arachni	151.67 (±10.69)	40.45%	43.00 (±6.93)	11.47%
w3af	166.33 (±1.15)	44.35%	33.33 (±0.58)	8.89%
ZAP	119.67(±4.62)	31.91%	13.00 (±0.00)	3.47%
Wapiti	199.67 (±6.66)	53.25%	72.67 (±0.58)	19.38%
Wapiti+	199.67 (±6.66)	53.25%	106.67 (±2.08)	28.45%
Avg.	159.00	42.40%	44.21	11.79%

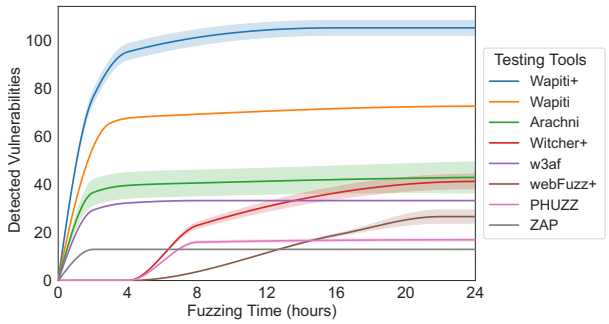

Figure 6: Time-to-Exposure analysis of existing work on PHPBench-constructed benchmark. Colored bands denote the range of detected vulnerabilities across three 24-hour runs.

6.3.1 Covered Insertion Points. PHUZZ demonstrated the lowest coverage at 24.18% (averaging 90.67 insertion points). Its basic crawler, limited to parsing <a> and <form> tags, fails to navigate modern web applications that rely on JavaScript events tied to other elements like and . Conversely, Witcher+ achieved the highest coverage at 57.15%. This superior performance stems from its advanced exploration phase and a crawling mechanism simulating user behaviors to uncover dynamically generated URLs.

However, despite three rounds of testing per tool, 96 insertion points remained undetected by any tested tools. Notably, 73.96 % of them were identified by Black Widow [31] during benchmark construction. These points are typically located deep within a page and require specific event sequences to generate access URLs (e.g., in Figure 7). Their discovery was enabled by Black Widow's traversing design, a feature absent in the tested tools.

Takeaway 1: Our evaluation shows that the tested tools only found 159 insertion points (42.40%) on average. Notably, all tools failed to discover 71 points found by Black Widow [31]. We thus recommend that future efforts leverage or extend state-of-the-art crawlers over simplistic, self-implemented ones. This will enhance the discovery of more dynamic and in-depth URLs, thereby uncovering more potential attack surfaces.

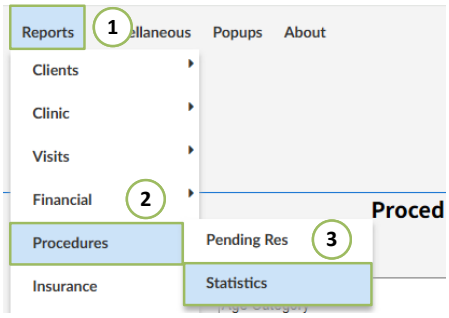

Figure 7: Insertion point in the statistics functionality of OpenEMR. All tools failed for the required click sequences.

6.3.2 Covered Inserted Vulnerabilities. During vulnerability detection, all tools demonstrated low detection rates, with recall ranging from 3.47% to just 28.45%. Notably, tools struggled to confirm vulnerabilities even after successfully reaching their insertion points. For example, averaged over three runs, webFuzz+ discovered 130.00 insertion points but only confirmed vulnerabilities in 26.67 of them (20.52%). More broadly, the average confirmation rate across all tools was merely 27.81%, indicating that limited confirmation capabilities are a major bottleneck for existing fuzzers. Our analysis of these unconfirmed vulnerabilities reveals three primary causes:

❶ *Context-unaware test case generation.* Tools often fail to generate payloads that meet strict input-specific formatting (e.g., an SQLi payload needs to conform to a valid email address format) or context constraints (e.g., an XSS payload must align with the surrounding HTML structure). We found that under-test tools rarely consider these contextual requirements, with the notable exceptions of Wapiti [18], which designs 55 payload strategies for five types of HTML contexts to detect XSS vulnerabilities. This design contributed significantly to its vulnerability detection rate. For instance, Listing 1 illustrates an XSS variant of CVE-2021-40541 inserted into Piwigo's album configuration functionality. Successfully detecting this vulnerability requires escaping the `$text` variable's context (line 8) to enable JavaScript injection; otherwise, the input renders as a literal string. Wapiti actively analyzes the client-side context to select appropriate breakout strategies. Consequently, it successfully escaped the context in all five attempts, and detected the vulnerability. In contrast, context-unaware fuzzers like webFuzz+ failed in all 12 attempts by neglecting context escaping requirements, resulting in every test case being processed as a literal string.

```
1 // Inserted into the album configuration functionality of Piwigo
2 function vuln_function_c31b44($var){
3     $input = $var;
4     ...
5     $preg_patterns = ['#<script(.*?)>(.*?)</script>#is'=>'', ...];
6     foreach ($preg_patterns as $pattern => $replacement)
7         $text = preg_replace($pattern, $replacement, $input);
8     echo '<a href="/base/path?' . $text . '">My Link</a>';
9 }
```

Listing 1: Fuzzing context awareness: webFuzz+ failed but Wapiti succeeded in an XSS variant of CVE-2021-40541.

❷ *Simplistic or ineffective mutation strategies.* The under-test tools often lack highly effective test case mutation strategies capable of evading flawed security protections. They either rely on predefined and often limited payload dictionaries [4, 17, 18, 21] (e.g., w3af for SQLi vulnerabilities has only two test cases) or adopt random mutation strategies [42, 55, 56] from binary fuzzing [3]. This leads to a high volume of futile attempts irrelevant to specific flawed protection or rendered syntactically invalid by random mutations, hindering vulnerability confirmation within the allotted time. For instance, when detecting an XSS vulnerability in the PrestaShop profile modification function, PHUZZ generated only 1/142 (0.70%) valid fuzzing attempts on a single parameter, despite requiring no context escaping. The remaining attempts were invalid mutations, such as random character insertion or deletion (e.g., "test", "teLst", "tst"), which were ineffective in triggering the XSS. Consequently, PHUZZ failed to confirm this case.

❸ *Unfocused fuzzing of non-critical request parameters.* When a request contains multiple parameters, existing tools typically cannot infer which specific parameter is more likely to be vulnerable. Consequently, they waste resources by fuzzing all parameters randomly, which vastly inflates the search space and reduces the number of effective tests on the actual critical parameter. For example, within the MyBB private message management feature, a request leading to an inserted vulnerability involved 16 parameters. webFuzz+ generated 140 tests for this request, yet only one of them (0.71%) effectively targeted the vulnerable parameter.

> ***Takeaway 2:*** Our evaluation shows that the tested tools performed poorly in vulnerability confirmation, detecting an average of only 11.79% of vulnerabilities, which is significantly lower than the reached insertion points (averaging 42.40%). This demonstrates that ineffective vulnerability confirmation is a major bottleneck. We thus recommend that future research focus on improving three key areas: 1) context-aware fuzzing, 2) mutation strategies capable of evading flawed protections, and 3) targeted fuzzing of critical request parameters.

6.3.3 Vulnerability Criteria Differences. We observed a significant disparity in vulnerability types detected by Witcher+ and Wapiti. Specifically, SQLi and CMDi constituted 90.38% for Witcher+, whereas 100% of those found by Wapiti were XSS.

Upon investigation, we found that this discrepancy arises from their different criteria for confirming SQLi and CMDi vulnerabilities. As a gray-box tool, Witcher can confirm vulnerabilities by monitoring for runtime syntax errors. In contrast, Wapiti, being a black-box tool, can only rely on observable consequences such as injection-induced time delays—a significantly more stringent criterion. For example, as shown in Listing 2, when detecting a SQLi variant of CVE-2021-3286 (inserted into the search functionality of MyBB), the flawed protection bans the `SLEEP` function used to introduce time delays in SQL injection (line 5). Consequently, Wapiti fails to trigger its overly strict oracle and misses this vulnerability, even though the injection causes a SQL syntax error, while Witcher succeeds due to its more appropriate oracle.

```
1 // Inserted into the search functionality of MyBB
2 function vuln_function_3d3fcf($var) {
3     $input = $var;
4     ...
5     $blacklist = ['DELETE', 'DROP', 'SLEEP', ...];
6     if (preg_match('/' . implode('|', $blacklist) . '/i', $input))
```

Table 4: A comparative performance analysis of existing techniques on two ad-hoc alternatives versus our benchmark. The detection number for each tool represents the average of three runs (same configuration as against our benchmark).

Ad-hoc Methods	Vulns	Metrics	webFuzz+	Witcher+	PHUZZ	Arachni	w3af	ZAP	Wapiti	Wapiti+
DVWA	12	Num.	0.00	10.00	12.00	6.00	5.00	7.00	8.00	10.00
		Recall (%)	0.00%	83.33%	100.00%	50.00%	41.67%	58.33%	66.67%	83.33%
		Diff. (%)	↓ -7.11%	↑ +72.31%	↑ +95.47%	↑ +38.53%	↑ +32.78%	↑ +54.86%	↑ +47.29%	↑ +54.88%
Aggregated Ground Truth	145	Num.	26.67	41.33	17.00	43.00	33.33	13.00	72.67	106.67
		Recall (%)	18.39%	28.50%	11.72%	29.66%	22.99%	8.97%	50.12%	73.57%
		Diff. (%)	↑ +11.28%	↑ +17.48%	↑ +7.19%	↑ +18.19%	↑ +14.10%	↑ +5.50%	↑ +30.74%	↑ +45.12%

```
7         die("Malicious keywords detected");
8     $db->query("SELECT * FROM items WHERE id =" . $input);
9 }
```

Listing 2: Vulnerability criteria differences: Wapiti failed but Witcher+ succeeded in a SQLi variant of CVE-2021-3286.

To measure the impact of this difference, we constructed Wapiti+, which integrates Witcher's bug oracle for SQLi and CMDi into Wapiti. The evaluation results showed that, based on the same set of insertion points reached by Wapiti, Wapiti+ reported an additional 34 SQLi and CMDi vulnerabilities (improved 46.79% than Wapiti). This indicates that different vulnerability criteria have a significant impact on results.

> ***Takeaway 3:*** The evaluation shows that vulnerability detection recall for even the best-performing tool varied by 46.79% before and after unifying the bug oracle. Therefore, we encourage future web fuzzing efforts, when conducting comparative experiments, to account for the criteria differences in bug oracles across various techniques, as these can significantly impact the vulnerability detection results of the tested tools.

6.4 RQ4: Compared to Ad-Hoc Practices

In this experiment, we evaluate existing work using two ad-hoc methods. The detailed comparison results are presented in Table 4.

Setup. To compare our benchmark against ad-hoc practices, we employed two of the three methods discussed in §2.2. First, we evaluated the tools on the popular educational testbed Damn Vulnerable Web Application (DVWA) [5], a common ground truth in prior studies [33, 42, 57, 63], which contains 12 XSS, SQLi, and CMDi vulnerabilities. Second, we used the aggregated results from three runs of all baselines as presented in §6.3. We excluded the third method, using self-selected historical vulnerabilities, due to its ambiguous selection criteria and the inconsistent availability of such vulnerabilities across different application versions.

On DVWA. We found that most tool performance on DVWA was significantly inflated. For example, PHUZZ's recall surged to 100%—a 95.47% increase over its performance on our benchmark—misleadingly suggesting its superiority. This discrepancy stems from: (1) DVWA's vulnerabilities lie in shallow locations that do not require robust exploration; (2) its requests contain only a single parameter for mutation, simplifying the search space; (3) its small, static nature makes it highly susceptible to overfitting. In contrast, webFuzz+ exhibited anomalous behavior by detecting no vulnerabilities in DVWA. This failure likely stems from design or implementation flaws. For instance, the tool consistently attempted to visit inaccessible pages and failed to initiate vulnerability confirmation, even on shallow requests. Consequently, it was unable to confirm any vulnerabilities within the allotted time.

On Aggregated Ground Truth. We similarly observed inflated detection capabilities across all tools, particularly among the top performers, where inflated metrics masked their flaws. For instance, the detection recall of the previously leading tool, Wapiti+, appeared to soar from 28.45% to 73.57%, leading to the following limitations: (1) *It yields unreliable conclusions.* The performance of tools like Wapiti+ is overstated because the ground truth is incomplete—230 vulnerabilities in the dataset are not accounted for, which impedes an accurate assessment. (2) *It obscures the challenge of vulnerability confirmation.* The methodology fails to show that tools may reach many potential vulnerabilities but cannot confirm them, potentially misdirecting research toward improving code coverage rather than the more pressing need to enhance vulnerability confirmation.

7 Discussion

Scalability and Extensibility. Although our evaluation used an initial set of 28 high-quality vulnerability seeds, PHPBench's high degree of automation makes the benchmark construction process highly scalable. The primary manual effort is confined to curating these seeds. This approach allows researchers to easily expand the benchmark to reflect evolving threats. The framework is also extensible because its core methodology—weakening security protections along control and data flows—is not limited to current vulnerability types. End-users can easily adapt it to support other injection-based vulnerabilities to suit their research objectives.

Threats to Validity. A potential threat to validity exists in our dynamic exploration phase. The framework's ability to identify deep and realistic vulnerability insertion points is bounded by the code coverage of the underlying web exploration techniques it employs. Nevertheless, we found this sufficient for evaluating existing tools, as the highest coverage achieved by any fuzzer in our experiments was only 57.15%. We also anticipate that this limitation will be mitigated over time by advancements in web crawling technology, which will inherently improve our insertion point selection.

8 Related Work

Web Vulnerability Detection. Detecting vulnerabilities in PHP web applications, a persistent security threat, primarily relies on

two approaches: static and dynamic analysis. Static analysis [23–27, 36, 40, 51, 60, 61] inspects source code for vulnerabilities using predefined source-sink models and heuristic rules. However, the lack of runtime verification often results in a high false positive rate. In contrast, dynamic analysis [4, 17, 18, 21, 30, 32, 33, 38, 42, 44, 55–58, 63] offers higher precision by executing the application and observing its runtime behavior. Nonetheless, improving the exploratory and confirmatory capabilities of dynamic techniques remains an open research challenge. We propose PHPBench to address this gap by facilitating fairer comparisons, clarifying bottlenecks, and encouraging the development of novel solutions.

PHP Vulnerability Benchmark for Static Analysis. There are several PHP vulnerability benchmarks designed for static vulnerability detection techniques [22, 48, 54, 59]. Similar to PHPBench, these techniques reference known vulnerability causes but generate vulnerability variants in different ways. Given their goals are to evaluate the bottlenecks of static detection techniques, these methods focus on introducing diverse features of the vulnerable code (e.g., callback functions or dynamic calls that hinder static analysis). However, these "tarpits" are often trivial for dynamic techniques that perform runtime analysis rather than static code analysis.

9 Conclusion

In this paper, we introduced PHPBench to resolve the critical lack of a high-quality benchmark for evaluating PHP web fuzzing tools. Our framework meets the core requirements for a robust benchmark through a novel, automated (R1) process. To ensure realism (R2), PHPBench generates vulnerability variants from real-world vulnerability causes and inserts them into deep, functionally-relevant application workflows. It establishes a hierarchy (R3) of complexity by systematically weakening existing security protections. Crucially, PHPBench verifies every vulnerability by synthesizing a verifiable PoC (R4), guaranteeing 100% exploitability. By evaluating seven fuzzers on our benchmark, we highlight the systemic weaknesses of existing tools and the misleading nature of ad-hoc evaluations, paving the way for more rigorous and impactful future research.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (U2436207), Hong Kong RGC Project (No. PolyU15231223), Hong Kong RGC Grant for Theme-based Research Scheme Project (No. T41-517/25-N), PolyU Distinguished Postdoctoral Fellowship Scheme (P0059445), JC STEM Early Career Research Fellowship for Translation and Application (P0059443 & P0063736). Yuan Zhang, Xiapu Luo and Min Yang are the corresponding authors. The work described in this paper was conducted in part by Dr SHI Youkun, JC STEM Early Career Research Fellow supported by The Hong Kong Jockey Club Charities Trust. Yuan Zhang was supported in part by the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012).

References

[1] 2015. PHP Vulnerability Test Suite. https://samate.nist.gov/SARD/test-suites/103.
[2] 2024. W3Techs - Usage statistics of PHP for websites. https://w3techs.com/technologies/details/pl-php.
[3] 2026. American Fuzzy Lop. https://github.com/google/AFL.
[4] 2026. Arachni. https://github.com/Arachni/arachni.
[5] 2026. Damn Vulnerable Web Application. https://github.com/digininja/DVWA.
[6] 2026. Dynamic Taint Extension. https://github.com/laruence/taint.
[7] 2026. Github-PHP-Parser. https://github.com/nikic/PHP-Parser.
[8] 2026. Github-Testing Payloads. https://github.com/payload-box.
[9] 2026. OWASP-Command Injection. https://owasp.org/www-community/attacks/Command_Injection.
[10] 2026. OWASP-Cross Site Scripting. https://owasp.org/www-community/attacks/xss/.
[11] 2026. OWASP-SQL Injection. https://owasp.org/www-community/attacks/SQL_Injection.
[12] 2026. Source Code - PHUZZ. https://github.com/gehaxelt/phuzz.
[13] 2026. Source Code - Witcher. https://github.com/sefcom/Witcher.
[14] 2026. Source code of Atropos. https://github.com/CISPA-SysSec/atropos-legacy.
[15] 2026. Source Code of Black Widow. https://github.com/1N3/BlackWidow.
[16] 2026. Source Code of webFuzz. https://bitbucket.org/srecgrp/webfuzz-fuzzer/src/v1.2.1/.
[17] 2026. w3af. https://github.com/andresriancho/w3af.
[18] 2026. Wapiti. https://wapiti-scanner.github.io/.
[19] 2026. Wikipedia. https://www.wikipedia.org.
[20] 2026. WordPress. https://wordpress.com.
[21] 2026. ZAP. https://github.com/zaproxy/zaproxy.
[22] Feras Al Kassar, Giulia Clerici, Luca Compagna, Davide Balzarotti, and Fabian Yamaguchi. 2022. Testability Tarpits: the Impact of Code Patterns on the Security Testing of Web Applications.. In *NDSS*.
[23] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. 2018. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*.
[24] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy*.
[25] Johannes Dahse and Thorsten Holz. 2014. Simulation of Built-in PHP Features for Precise Static Code Analysis.. In *NDSS*, Vol. 14. 23–26.
[26] Johannes Dahse and Thorsten Holz. 2014. Static Detection of Second-order Vulnerabilities in Web Applications. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*. 989–1003.
[27] Johannes Dahse and J Schwenk. 2012. Rips-a static source code analyser for vulnerabilities in PHP scripts (2010).
[28] Jiarun Dai, Yuan Zhang, Hailong Xu, Haiming Lyu, Zicheng Wu, Xinyu Xing, and Min Yang. 2021. Facilitating vulnerability assessment through poc migration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 3300–3317.
[29] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. Lava: Large-scale automated vulnerability addition. In *2016 IEEE symposium on security and privacy*.
[30] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2012. Enemy of the state: A state-aware black-box web vulnerability scanner. In *21st USENIX Security Symposium (USENIX Security 12)*. 523–538.
[31] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. 2021. Black widow: Blackbox data-driven web scanning. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1125–1142.
[32] Benjamin Eriksson, Amanda Stjerna, Riccardo De Masellis, Philipp Rüemmer, and Andrei Sabelfeld. 2023. Black Ostrich: Web Application Scanning with String Solvers. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 549–563.
[33] Emre Güler, Sergej Schumilo, Moritz Schloegel, Nils Bars, Philipp Görz, Xinyi Xu, Cemal Kaygusuz, and Thorsten Holz. 2024. Atropos: Effective fuzzing of web applications for server-side vulnerabilities. In *USENIX Security Symposium*.
[34] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29.
[35] Rasoul Jahanshahi, Babak Amin Azad, Nick Nikiforakis, and Manuel Egele. 2023. Minimalist: Semi-automated debloating of php web applications through static analysis. In *32nd USENIX Security Symposium (USENIX Security 23)*.
[36] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P)*. IEEE.
[37] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2123–2138.
[38] Taekjin Lee, Seongil Wi, Suyoung Lee, and Sooel Son. 2020. FUSE: Finding File Upload Bugs via Penetration Testing.. In *NDSS*.
[39] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. 2021. UNIFUZZ: A holistic and pragmatic Metrics-Driven platform for evaluating fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*. 2777–2794.

[40] Changhua Luo, Penghui Li, and Wei Meng. 2022. 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*. 2175–2188.
[41] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 1393–1403.
[42] Sebastian Neef, Lorenz Kleissner, and Jean-Pierre Seifert. 2024. What all the phuzz 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*. 1523–1538.
[43] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2023. Vulgen: Realistic vulnerability generation via pattern mining and deep learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*.
[44] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. 2015. jäk: Using dynamic analysis to crawl and test modern web applications. In *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015. Proceedings 18*. Springer.
[45] Jannik Pewny and Thorsten Holz. 2016. EvilCoder: automated bug insertion. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*.
[46] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. 2018. Bug synthesis: Challenging bug-finding tools with deep faults. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 224–234.
[47] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. Sok: Prudent evaluation practices for fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1974–1993.
[48] Felix Schuckert, Hanno Langweg, and Basel Katt. 2022. Systematic generation of XSS and SQLi vulnerabilities in PHP as test cases for static code analysis. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 261–268.
[49] Ze Sheng, Zhicheng Chen, Shuning Gu, Heqing Huang, Guofei Gu, and Jeff Huang. 2025. LLMs in Software Security: A Survey of Vulnerability Detection Techniques and Insights. *arXiv e-prints* (2025), arXiv–2502.
[50] Youkun Shi, Yuan Zhang, Tianhao Bai, Feng Xue, Jiarun Dai, Fengyu Liu, Lei Zhang, Xiapu Luo, and Min Yang. 2025. XSSky: Detecting XSS Vulnerabilities through Local Path-Persistent Fuzzing. In *34th USENIX Security Symposium (USENIX Security 25)*. 8255–8272.
[51] Youkun Shi, Yuan Zhang, Tianhao Bai, Lei Zhang, Xin Tan, and Min Yang. 2024. RecurScan: Detecting Recurring Vulnerabilities in PHP Web Applications. In *Proceedings of the ACM on Web Conference 2024*. 1746–1755.
[52] Youkun Shi, Yuan Zhang, Tianhan Luo, Xiangyu Mao, Yinzhi Cao, Ziwen Wang, Yudi Zhao, Zongan Huang, and Min Yang. 2022. Backporting security patches of web applications: A prototype design and implementation on injection vulnerability patches. In *31st USENIX Security Symposium (USENIX Security 22)*.
[53] Youkun Shi, Yuan Zhang, Tianhan Luo, Xiangyu Mao, and Min Yang. 2022. Precise (un) affected version analysis for web vulnerabilities. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*.
[54] Bertrand Stivalet and Elizabeth Fong. 2016. Large scale generation of complex and faulty PHP test cases. In *2016 IEEE International conference on software testing, verification and validation (ICST)*. IEEE, 409–415.
[55] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. 2023. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities. In *2023 IEEE symposium on security and privacy*. IEEE, 2658–2675.
[56] Orpheas van Rooij, Marcos Antonios Charalambous, Demetris Kaizer, Michalis Papaevripides, and Elias Athanasopoulos. 2021. webfuzz: Grey-box fuzzing for web applications. In *Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I 26*. Springer, 152–172.
[57] Chenlin Wang, Wei Meng, Changhua Luo, and Penghui Li. 2024. Predator: Directed Web Application Fuzzing for Efficient Vulnerability Validation. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 66–66.
[58] Enze Wang, Jianjun Chen, Wei Xie, Chuhan Wang, Yifei Gao, Zhenhua Wang, Haixin Duan, Yang Liu, and Baosheng Wang. 2024. Where URLs Become Weapons: Automated Discovery of SSRF Vulnerabilities in Web Applications. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 239–257.
[59] Lin Wang, Yuan Zhang, Xin Tan, Shengke Ye, and Min Yang. 2024. New PHP Language Features Make Your Static Code Analysis Tools Miss Vulnerabilities. In *2024 IEEE International Conference on Software Maintenance and Evolution*.
[60] Seongil Wi, Sijae Woo, Joyce Jiyoung Whang, and Sooel Son. 2022. HiddenCPG: Large-Scale Vulnerable Clone Detection using Subgraph Isomorphism of Code Property Graphs. In *Proceedings of the ACM Web Conference 2022*.
[61] Yichen Xie and Alex Aiken. 2006. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the 15th USENIX Security Symposium*.
[62] Zenong Zhang, Zach Patterson, Michael Hicks, and Shiyi Wei. 2022. FIXREVERTER: A realistic bug injection methodology for benchmarking fuzz testing. In *31st USENIX Security Symposium (USENIX Security 22)*. 3699–3715.
[63] Jiazhen Zhao, Yuliang Lu, Kailong Zhu, Zehan Chen, and Hui Huang. 2022. Cefuzz: An directed fuzzing framework for php rce vulnerability. *Electronics* 11, 5 (2022), 758.

Ethical Considerations

This paper adheres to strict ethical guidelines and introduces no risks to users or online systems. Our methodology is grounded in the following key principles:

- *Use of Public Known Vulnerabilities.* All vulnerability variants generated and inserted by PHPBench are derived exclusively from publicly known and documented vulnerabilities (CVEs). The objective of our work is to create benchmark challenges from these known security flaws, not to discover or exploit new, undisclosed vulnerabilities in real-world applications.
- *Isolated Experimental Environment.* All experimental procedures, including application instrumentation, vulnerability insertion, and subsequent fuzzer evaluations, were conducted within a controlled and isolated local environment. We used popular open-source applications deployed in Docker containers, ensuring no interaction with or impact on any live, production systems.

Open Science

To promote transparency and facilitate the reproduction of our work, we have made our research artifacts publicly available[2]. The artifacts consist of (1) the source code of the PHPBench prototype, (2) the entire PHPBench-constructed benchmark for our evaluation (composed of 10 Docker images, one for each target application with the inserted vulnerabilities, and the corresponding verifiable PoCs), and (3) the detailed results from our evaluation of existing techniques on our benchmark (comprehensive data on the specific insertion points and the vulnerabilities detected by each tested tool).

A Setup of Existing Dynamic Techniques

A.0.1 Black Widow Setup. We follow the guidelines [15] and conducted its setup. First, we installed its dependencies:

```
sudo apt install -y google-chrome-stable chromedriver
pip install -r requirements.txt
```

We then executed it with the following command, imposing a time limit of 8 hours during the benchmark construction phase:

```
timeout 8h python crawl.py --url=target_application_seed_URLs
```

A.0.2 Arachni Setup. We employed the publicly available Docker image for Arachni (i.e., `hercules9/arachni`) [4]. During the benchmark construction phase, we then executed it with the following command, imposing a time limit of 8 hours:

```
docker run -v $(pwd)/arachni-reports:\
  /opt/arachni/reports hercules9/arachni /usr/local/arachni/bin/arachni \
    target_application_seed_URLs --timeout=8:00:00
```

During the evaluation phase, we evaluated it on our benchmark:

```
docker run -v $(pwd)/arachni-reports:\
    /opt/arachni/reports hercules9/arachni \
    /usr/local/arachni/bin/arachni target_application_seed_URLs \
    --checks="active/*"  --timeout=24:00:00
```

[2]https://zenodo.org/records/20603989, https://github.com/SheltonShi/WebVulnBench, and https://hub.docker.com/r/sheltonshi/webvulnbench

A.0.3 w3af Setup. We directly utilized the publicly available Docker image for w3af (i.e., `andresriancho/w3af`) [17]. During the benchmark construction phase, we then executed it with the following command, imposing a time limit of 8 hours:

```
docker run andresriancho/w3af bash
./w3af_console
target set target target_application_seed_URLs
plugins crawl web_spider
start
```

During the evaluation phase, we evaluated it on our benchmark:

```
docker run andresriancho/w3af bash
./w3af_console
target set target target_application_seed_URLs
plugins crawl web_spider
plugins audit xss,sqli,os_commanding
plugins output csv_file
start
```

A.0.4 ZAP Setup. We directly utilized the publicly available Docker image for ZAP (i.e., `zaproxy/zap-stable`) [21]. During the benchmark construction phase, we then executed it with the following command, imposing a time limit of 8 hours:

```
docker run -v /opt/zap-wrk:/zap/wrk/:rw \
 zaproxy/zap-stable zap-full_scan.py -t target_application_seed_URLs -T 480
```

During the evaluation phase, we evaluated it on our benchmark:

```
docker run -v /opt/zap-wrk:/zap/wrk/:rw \
 zaproxy/zap-stable zap-full_scan.py -t target_application_seed_URLs -T 1440
```

A.0.5 Wapiti Setup. Given that Wapiti [18] is directly packaged as a Python library, we used the following command for installation:

```
pip install wapiti
```

During the benchmark construction phase, we then executed it with the following command, imposing a time limit of 8 hours:

```
wapiti --url target_application_seed_URLs -t 28800
```

During the evaluation phase, we evaluated it on our benchmark:

```
wapiti --url target_application_seed_URLs -t 86400
```

A.0.6 webFuzz Setup. We follow the guidelines of webFuzz [16] and conducted its setup. First, we installed its dependencies:

```
pip install --upgrade -r web_fuzzer/requirements.txt
```

Then, we conducted the necessary instrumentation:

```
cd instrumentor/src
php7.4 instrumentor.php --verbose --method file \
    --policy edge --dir target_application_path \
    --instrpath /var/log --mappath instrument_meta_path
```

After that, we executed it with the following command, imposing a time limit of 8 hours for the benchmark construction phase:

```
timeout 8h python webFuzz.py \
    --driver ./geckodriver -m instrument_meta_path \
    -w 1 -r simple "target_application_seed_URLs"
```

During the evaluation phase, we evaluated it on our benchmark:

```
timeout 24h python webFuzz.py --driver ./geckodriver \
   -m instrument_meta_path -w 1 -r simple "target_application_seed_URLs"
```

A.0.7 PHUZZ Setup. We follow the guidelines of PHUZZ [12] and conducted its setup. First, we installed its dependencies:

```
pip install -r requirements.txt
```

Then, we conducted the necessary instrumentation:

```
mv target_application_path ./web/applications/
sed -i 's/dockerfile: Dockerfile/dockerfile: \
        Dockerfile.php7/' docker-compose.yaml
docker-compose build web
```

After that, we then executed it with the following command, imposing a time limit of 8 hours during the benchmark construction:

```
python crawler.py --baseurl target_application_base_URL \
 --entrypoint target_application_seed_URLs --timeout 28800
```

During the evaluation phase, we first execute the command to convert the collected URLs into fuzzing configuration files, and then evaluated it on our benchmark using the following command:

```
./hargen.sh
python fuzzer.py
```

A.0.8 Witcher Setup. We follow the guidelines [13] and conducted its setup. First, we installed its dependencies with the commands:

```
sudo apt install -y nodejs chromium-browser
```

Then, following the instructions in README.md, we configured its default settings:

```
mkdir -p /opt/witcher-conf
touch /opt/witcher-conf/witcher_config.json
vi /opt/witcher-conf/witcher_config.json
```

After that, we conducted the necessary instrumentation:

```
vi Dockerfile
# Inside application's Dockerfile
from witcherfuzz/php7run
# configurations specific to application

# Outside application's Dockerfile
docker build -t target_application:instrumented
```

We then executed it with the following command, imposing a time limit of 8 hours during the benchmark construction phase:

```
cd base/helpers/request_crawler
timeout 8h node main.js request_crawler \
    target_application_seed_URLs /opt/witcher-conf --no-headless
```

During the evaluation phase, we first move the URL configuration file to the working directory, and then evaluated it on our benchmark using the following command:

```
mv ./request_data.json /witcher/tests/test_data/
cd /witcher/tests/test_data/
su wc
python3 -m witcher --testver WICHR
```

B Original Vulnerabilities

Table 5 presents the original vulnerabilities used as input for PHPBench, which consist of 10 XSS, 10 SQLi, and 8 CMDi vulnerabilities.

Table 5: Distribution of 28 selected original vulnerabilities.

CVE-ID	Vuln Types	CVE-ID	Vuln Types
CVE-2020-14475	XSS	CVE-2021-35437	SQLi
CVE-2020-16145	XSS	CVE-2021-41155	SQLi
CVE-2021-40541	XSS	CVE-2021-42235	SQLi
CVE-2022-0268	XSS	CVE-2022-23337	SQLi
CVE-2022-4421	XSS	CVE-2024-27299	SQLi
CVE-2022-40704	XSS	CVE-2024-57151	SQLi
CVE-2023-0606	XSS	CVE-2020-10789	CMDi
CVE-2023-3191	XSS	CVE-2020-11016	CMDi
CVE-2023-5842	XSS	CVE-2020-12078	CMDi
CVE-2024-12183	XSS	CVE-2020-21785	CMDi
CVE-2019-10762	SQLi	CVE-2023-0671	CMDi
CVE-2020-15176	SQLi	CVE-2023-3368	CMDi
CVE-2021-3286	SQLi	CVE-2023-32073	CMDi
CVE-2021-28245	SQLi	CVE-2023-38886	CMDi