VUnitFuzz: Decoupling PHP Sink Verification from Monolithic Execution for Taint-Style Vulnerability Detection

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

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

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

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

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

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

Abstract

Taint-style vulnerabilities remain prevalent in PHP applications. Detecting them fundamentally requires determining whether the input space that reaches a security-sensitive sink still admits malicious values. Static approaches provide broad code coverage, but often struggle with path-specific constraints and complex custom sanitizers, forcing them to approximate the sink input space. Dynamic and hybrid approaches execute concrete payloads, but they remain tightly coupled to monolithic application execution, forcing tools to resolve client-side interactions, routing semantics and server-side functional checks before they can exercise the sink.

In this paper, we introduce Vulnerability Unit (V-Unit) fuzzing, a novel detection paradigm that decouples vulnerability-relevant logic from monolithic application execution. A V-Unit is an independently executable artifact that preserves the data-flow sanitizers and control-flow constraints defining a specific sink input space, alongside the minimal execution context required for runtime. Building on this paradigm, we present VUnitFuzz, an automated V-Unit fuzzing framework for PHP applications. By integrating path-insensitive, branch-selective skeleton extraction with interpreter-guided, on-demand context construction, VUnitFuzz enables direct sink fuzzing for taint-style vulnerability detection.

We evaluated VUnitFuzz on 50 popular PHP applications. The results show that VUnitFuzz successfully extracted 12,262 V-Units and detected 357 confirmed taint-style vulnerabilities with 98.89% precision and 95.20% recall, including 232 XSS, 122 SQLi and 3 CMDi cases. Compared with state-of-the-art static, dynamic, and hybrid tools, VUnitFuzz improves precision and recall by up to 370.68% and 2129.51%, respectively. To date, developers have acknowledged 70 newly discovered vulnerabilities, 28 of which have been patched.

CCS Concepts

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

Keywords

Vulnerability Detection, Web Security

ACM Reference Format:
Youkun Shi, Yuan Zhang, Tianhao Bai, Jiarun Dai, Fengyu Liu, and Xiapu Luo. 2026. VUnitFuzz: Decoupling PHP Sink Verification from Monolithic Execution for Taint-Style Vulnerability Detection. 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/10.1145/3830454.3846756



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/10.1145/3830454.3846756

1 Introduction

PHP applications remain a cornerstone of the modern web ecosystem. Recent statistics show that PHP powers over 71% of websites globally [3], such as Wikipedia [4] and Facebook [1]. However, this broad deployment also makes PHP applications a persistent target for attacks. Among the most prevalent threats are taint-style vulnerabilities, particularly Cross-Site Scripting (XSS), SQL Injection (SQLi), and Command Injection (CMDi), which arise when attacker-controlled inputs (i.e., sources) reach security-sensitive operations (i.e., sinks) without sufficient protection.

At the core of these vulnerabilities lies a simple question: **what inputs can actually reach a sink?** In real PHP applications, the answer is dictated by both data-flow sanitizers (e.g., encoding or escaping) and control-flow constraints (e.g., predicates validating inputs before execution) encountered along the way. Together, these protections define the **sink input space**: the set of transformed source values that survive security checks and reach the sink. A taint-style vulnerability exists if this input space still retains any value capable of triggering a security violation. Therefore, the central task of vulnerability detection is to evaluate the sink input space accurately and efficiently.

Existing approaches struggle to address this task effectively. ❶ Static-based analysis [11, 13, 25, 33, 41, 42, 45, 49] scans entire codebases without execution, but must statically approximate the sink input space. To remain scalable, these tools often sacrifice control-flow sensitivity and therefore miss path-specific guards. They also struggle to reason precisely about complex, developer-defined string transformations that shape the final sink value. As a result, coarse heuristics or heavyweight string reasoning often lead to both false positives and false negatives. Conversely, ❷ dynamic [14–16, 20, 37, 39, 44, 46, 47] and ❸ hybrid techniques [48, 52] validate candidate vulnerabilities through concrete execution. This

design is attractive because it naturally evaluates branch predicates and runtime string semantics. However, existing dynamic and hybrid designs tightly couple sink verification with monolithic application execution. Before testing a target sink, a detector must navigate client-side interactions, resolve flexible framework routing, and satisfy workflow-specific server-side checks. Much of this effort is orthogonal to whether the sink itself is sufficiently protected. Consequently, these approaches spend most of their efforts merely reaching the sink, leaving many sinks practically unverified.

This mismatch suggests a different design. For sink-level detection, it is often unnecessary to reproduce the full end-to-end execution required for exploitation. To determine whether a sink remains a threat, a detector only needs to execute the vulnerability-relevant logic that defines the sink input space, namely the data-flow sanitizers and control-flow constraints along source-to-sink paths. Much of the surrounding monolithic application, including routing dispatchers, UI rendering, and unrelated business logic, mainly serves as a vehicle for reaching that logic rather than defining it.

Guided by this observation, we propose **Vulnerability Unit** (V-Unit) fuzzing. Inspired by unit testing, V-Unit fuzzing decouples the verification of vulnerability-relevant logic from monolithic application execution. A V-Unit is an independently executable artifact that preserves the sanitizers and constraints defining a specific sink input space, together with the minimal execution context required to run them. By fuzzing V-Units directly, we execute real protection logic with concrete inputs while bypassing security-irrelevant reachability barriers such as routing and deep application states.

However, realizing V-Unit fuzzing in practice introduces two main challenges.

- *Challenge I: How to accurately and scalably extract the logic that defines the sink input space?* The extraction must preserve all and only the data-flow transformations and control-flow guards that shape the sink input space. In real PHP applications, these sink-defining elements may be distributed across multiple paths of the same source-sink pair and intertwined with unrelated business code. This creates a fundamental tension between precision and scalability: coarse-grained extraction introduces irrelevant code, whereas path-sensitive collection quickly leads to path explosion.
- *Challenge II: How to soundly and minimally recover the execution context?* The extracted logic is not self-executable. It still depends on missing execution context, such as callee definitions, yet identifying exactly which context is required for the current sink-relevant execution is difficult in PHP, as many reference targets are resolved only at runtime. Moreover, naively importing full definitions can pull large amounts of sink-irrelevant code back into the unit: real-world functions are often multi-purpose, and their transitive dependencies can trigger cascading context expansion, undermining the efficiency gained by decoupling.

To address these challenges, we design `VUnitFuzz`, an automated V-Unit fuzzing framework for PHP web applications. For Challenge I, `VUnitFuzz` introduces sink-input-space-oriented Skeleton extraction. Instead of statically solving branch semantics for every source-to-sink path, `VUnitFuzz` directly preserves the constraint structure that shapes the sink input space for each source-sink pair. Because later fuzzing can exercise different executions within the same pair through different inputs. `VUnitFuzz` therefore combines targeted backward data-dependency analysis, path-insensitive branch-selective forward traversal, and fine-grained AST-based materialization to preserve sink-defining logic while avoiding exhaustive path enumeration and the introduction of irrelevant code. For Challenge II, `VUnitFuzz` cleverly leverages the interpreted nature of PHP to perform interpreter-guided, on-demand execution-context construction. Rather than preloading a large transitive dependency graph, `VUnitFuzz` starts from the extracted Skeleton and recovers missing definitions only when the PHP interpreter demands them during execution. This design resolves runtime-dependent symbols precisely when they are exercised and keeps the resulting V-Unit focused on the logic needed for fuzzing. On top of this execution model, `VUnitFuzz` directly reuses state-of-the-art vulnerability-oriented fuzzing techniques and bug oracles [41, 46, 48] to verify XSS, SQLi, and CMDi vulnerabilities on V-Units.

We implemented `VUnitFuzz` and evaluated it on 50 popular PHP applications. Our results show that `VUnitFuzz` successfully extracted 12,262 executable V-Units, and detected 357 confirmed taint-style vulnerabilities with 98.89% precision and 95.20% recall. Compared with state-of-the-art static, dynamic, and hybrid baselines, `VUnitFuzz` improves precision by up to 370.68% and recall by up to 2129.51%. To date, developers have acknowledged 70 newly discovered vulnerabilities, 28 of which have been patched.

In summary, this paper makes the following contributions:

- We introduce the concept of the Vulnerability Unit, shifting taint-style vulnerability detection from monolithic application analysis to focused, unit-level verification of sink input spaces.
- We propose `VUnitFuzz` that combines structure-aware, path-insensitive, branch-selective Skeleton extraction with interpreter-guided, on-demand execution-context construction, preserving the logic that defines the sink input space while avoiding both path and context explosion.
- We implement `VUnitFuzz` and evaluate it on 50 real-world PHP applications, confirming 357 taint-style vulnerabilities and showing that V-Unit fuzzing improves the effectiveness and efficiency of vulnerability detection over state-of-the-art baselines.
- We release the source code of `VUnitFuzz`[1], together with the evaluation dataset and supporting artifacts, to facilitate the reproduction of our results and future research.

2 Problem Statement

This section frames taint-style vulnerability detection as a sink-input-space verification problem. We first define the vulnerability model used throughout the paper (§2.1). We then revisit existing detection workflows through this lens (§2.2), present a motivating example that exposes their limitations (§2.3), and distill the design takeaway that motivates V-Unit fuzzing (§2.4).

2.1 Taint-Style Vulnerability Scope

Taint-style vulnerabilities arise when untrusted data reaches a security-sensitive operation without sufficient protection. This paper focuses on three prevalent classes in PHP web applications: Cross-Site Scripting (XSS), SQL Injection (SQLi), and Command

[1] https://zenodo.org/records/21659116

Injection (CMDi). We model each of them using three components:

$$\langle Source, Sink, Protections \rangle$$

Source is an entry point through which untrusted data enters an execution trace. In PHP applications, common sources include superglobal variables that receive user inputs, such as `$_GET`, `$_POST`.
Sink is a security-sensitive operation whose argument may trigger a security violation when influenced by attacker-controlled data. Common sinks include PHP built-in functions, such as `echo` for XSS, `mysql_query` for SQLi, and `system` for CMDi.
Protections are program logic that constrain the propagation or value of source-derived data before it reaches a sink, including:

- *Data-flow Sanitizers:* Transformations $\mathcal{D}(x) \rightarrow x'$ that rewrite source-derived data by encoding, escaping, filtering, or removing dangerous characters. Examples include PHP built-in functions such as `htmlentities()` and developer-defined sanitizers implemented with string operations such as `preg_replace()`.
- *Control-flow Constraints:* Predicates $C(x) \rightarrow \{True, False\}$ that guard whether execution can proceed toward a sink based on its parameter value. For example, a branch such as `if(is_numeric($id))` restricts the values that can reach a SQL query by terminating executions with non-numeric inputs. Other constraints may select different sanitization branches or block specific input patterns.

Definition 1 (Sink Input Space). For a target sink, let $\Pi_{src \rightsquigarrow sink}$ denote the set of source-to-sink paths that may reach it. For each path $\pi \in \Pi_{src \rightsquigarrow sink}$, let $\mathcal{D}_\pi$ denote the data-flow transformations that produce the source-derived value observed at the sink, and let C_π denote the sink-related control-flow constraints that must hold along π. The Sink Input Space, denoted Ω_{sink}, is defined as:

$$\Omega_{sink} = \bigcup_{\pi \in \Pi_{src \rightsquigarrow sink}} \{\mathcal{D}_\pi(x) \mid x \in Source \wedge C_\pi(x)\}.$$

Definition 2 (Taint-Style Vulnerable Sink). Let Ω_{mal} denote the set of malicious values that can trigger a sink-specific security violation, such as executing JavaScript in an XSS sink or altering query semantics in a SQLi sink. A sink is taint-style vulnerable if its input space overlaps with the malicious space:

$$\Omega_{sink} \cap \Omega_{mal} \neq \emptyset.$$

2.2 Existing Detection Workflows

Existing taint-style vulnerability detection techniques generally fall into three categories: static, dynamic, and hybrid analysis. Although they differ in how they inspect applications, they all face difficulty in accurately and efficiently verifying the sink input space.
Static Analysis-Based Methods. Static analysis scans application code without executing it, enabling broad code coverage. However, without concrete runtime values, static tools must approximate Ω_{sink}. This approximation introduces two major limitations.

(1) *Control-flow abstraction.* Determining the feasibility and effect of path-specific branch conditions is expensive without runtime values. To remain scalable, many static techniques [11, 13, 25, 33, 41, 42, 45, 49] prioritize data-flow propagation and treat path-specific branch constraints coarsely or ignore them. This over-approximates Ω_{sink} when guards restrict the values that can reach a sink, producing false positives. Conversely, fully path-sensitive reasoning can quickly become impractical in branch-heavy web applications.

(2) *String-protection modeling.* Taint-style vulnerabilities often depend on structured string payloads, and security protections frequently involve string encoding, escaping, or custom rewriting. Existing static tools typically handle these protections through coarse sanitizer models [11, 13, 25, 33, 42, 49] or heavyweight string reasoning [28, 45, 49]. Coarse models can miss flawed sanitizers and developer-defined protections, while string reasoning is costly and remains difficult for complex custom logic [27, 41].
Dynamic Analysis-Based Methods. Dynamic approaches, including web scanners [14–16, 39, 44] and fuzzers [20, 37, 46, 47], execute the application with concrete payloads and monitor vulnerability oracles. This naturally evaluates branch predicates and string operations, but existing designs verify sinks only through monolithic application execution. Before a payload can test Ω_{sink}, the tool must first reach the target sink through two kinds of barriers.

(1) *Client-side state complexity.* Modern web applications often rely on client-side JavaScript to construct requests and manage interaction states [16, 39]. Triggering a server-side sink may require executing specific JavaScript event sequences or reaching particular UI states [21, 44]. If the tool fails to discover the corresponding request, the server-side sink remains untested regardless its input space is vulnerable, leading to false negatives.

(2) *Server-side functional checks.* Even when a valid request is available, execution may still need to satisfy application-specific functional checks, such as session-state checks before it can reach the vulnerable code. Existing techniques [20, 37, 46, 47] often cannot easily satisfy such checks, causing execution to be blocked before reaching the sink and leaving the vulnerability undetected.
Hybrid Analysis-Based Methods. Hybrid methods use static guidance for concrete payload validation [48, 52]. However, they still need to drive the full application to the target sink. In addition to inheriting the server-side functional check barrier discussed above, directed fuzzers must also construct valid entry URLs for statically identified code locations. This is difficult for modern PHP applications, where routing rules are often distributed across configuration files, annotations, controller logic, and framework-specific dispatchers. State-of-the-art tools such as Predator [48] still rely on coarse file-path mapping and often fail to construct valid entry URLs, preventing validation from even starting.

2.3 A Motivating Example

We next use a concrete example to illustrate why current approaches struggle to verify sink input space effectively.

Figure 1 shows a simplified example from a modern e-commerce application. The vulnerable controller logic appears in Figure 1 (a), while the helper functions that further shape the sink input space are defined in Figure 1 (b). The target functionality is reachable through the annotation-based routed URL */orderService/postComment*, as indicated by the route declarations at lines 2–4. After entering the controller, execution first checks functional session state (lines 6–7). It then processes the attacker-controlled input `$comm` (line 8) through several string operations and a regex-based validation guard (lines 9–14). Inputs accepted by this guard are emitted through the benign sink at line 15, whereas the remaining ones are rendered in `"xss"` mode and eventually emitted at

```php
1  include "functions/utility.php";
2  /*@Route("/orderService")*/
3  class OrderController extends UserController {
4      /*@Route("/postComment", methods={"POST"})*/
5      public function postComment() {
6          if (!search_purchase_history($_SESSION["uid"], $_POST["order_id"]))
7              http_response(403);
8          $comm = $_SESSION["uid"] . ":" . $_POST["comment"];
9          if (strlen($comm) > 200)
10             $comm = substr($comm, 0, 200) . "...";
11         if (strpos($comm, "http") !== false)
12             $comm = preg_replace("#https?:\/\/[^\s]+#i", "[Link Removed]", $comm);
13         $safe_pattern = "/^[a-zA-Z-_])+$/s";
14         if (preg_match($safe_pattern, $comm, $matches)) {
15             echo render_comment_card($comm); // Secure XSS sink
16         } else {
17             echo render_comment_card($comm, "xss"); // Vulnerable XSS sink
18         }
19     }
20 }
```

a) adapter/customer/orderController.php

```php
21 function render_comment_card($comm, $mode="default") {
22     $sanitizers = ["xss"=>"esc", ...];
23     if ($mode === "default") {
24         return "<span title='new_post'>" . $comm . "</span>";
25     }
26     elseif ($mode === "json") {
27         ... // Vulnerability-irrelevant and unexecuted code
28     }
29     elseif ($mode === "custom") {
30         ... // Vulnerability-irrelevant and unexecuted code
31     }
32     elseif ($mode === "xss") {
33         $output = $sanitizers["xss"]($comm);
34         return "<span title='new_post'>" . $output . "</span>";
35     }
36     ...
37 }
   // Vulnerable and can be evaded by <div onmouseover=alert(1)>
38 function esc($str) {
39     if (!is_numeric($str)) {
40         $str = preg_replace("#<script[^>]*>.*?</script>#is", "", $str);
41         $str = preg_replace("#<a[^>]*>.*?</a>#is", "", $str);
42         $str = preg_replace("#<img[^>]*>#is", "", $str);
43         $str = preg_replace("#(<[^>]*)onerror=onload=(.+>)#Uis", "$1$2", $str);
44     }
45     return $str;
46 }
```

b) functions/utility.php

Figure 1: A motivating example illustrating why existing detection workflows struggle to verify the sink input space.

line 17. Along this branch, `render_comment_card()` dynamically dispatches the sanitizer `$sanitizers["xss"]` at line 33, which resolves to `esc()`. Because `esc()` is vulnerable, malicious payloads (e.g., `<div onmouseover=alert(1)>`) can bypass the sanitization logic and still reach the vulnerable sink at line 17.

This example exposes the core limitations of recent state-of-the-art techniques across detection paradigms. Details are as follows:
Why are static approaches ineffective? Detecting this vulnerability requires path-sensitive reasoning about the guard at line 14, handling PHP dynamic dispatch at line 33, and evaluating developer-defined regular-expression protections at lines 12–14 and lines 40–43. Recent analyzers such as ZIPPER [49] and XSSky [41] still struggle with this combination. First, both tools ignore the full effect of the control-flow constraint (line 14), over-approximate the sink input space, and therefore produce a false positive (line 15). Second, neither tool reconstructs the full sink-defining logic behind the dynamic call (line 33). Due to these dynamic features, XSSky fails to instantiate the required execution context, leading to a false negative (line 17), whereas ZIPPER abstracts it symbolically. Consequently, ZIPPER's report of the true vulnerability is coincidental rather than a faithful verification of the custom protections.
Why are dynamic and hybrid approaches ineffective? Detecting this vulnerability through monolithic execution poses two key obstacles: (1) discovering or constructing a valid entry URL that reaches the vulnerable functionality, and (2) satisfying the functional checks along the execution path before the sink-defining logic is exercised. Recent state-of-the-art techniques, such as Witcher [46] and Predator [48], still fail to overcome these obstacles on this example. Witcher relies on web crawling to harvest valid entry URLs, but it is blocked by the state-based functional check at line 6 and therefore never verifies the downstream sink at line 17. Predator faces an even earlier obstacle: it must infer routing semantics well enough to construct a valid entry URL, yet its file-path-based request construction fails on the routed endpoint in this example. Even if a correct entry URL were given, Predator would still be blocked by the same state-based functional check. In short, barriers that lie outside the sink input space itself can completely prevent monolithic techniques from validating the vulnerability.

2.4 Rethinking Existing Workflows

Fundamentally, taint-style vulnerability detection seeks to decide whether Ω_{sink} overlaps with Ω_{mal}. Existing workflows address this problem inadequately. Static analysis discards concrete execution semantics; although this broadens the code scope, it necessitates approximating the protections that define Ω_{sink}. Conversely, dynamic and hybrid approaches execute the real application and therefore observe concrete semantics, but they need to pass client-side interactions, routing resolution, and server-side functional checks before the sink-defining logic can be exercised. These requirements are important for full exploit reproduction, but they are often orthogonal to the detection question: *whether the sink input space itself is capable of unsafe behavior?* This mismatch motivates a design that isolates and verifies sink-defining logic directly, while treating full exploit reproduction as a related but distinct concern.

3 Overall Idea

Building on §2.4, this section presents the high-level idea behind our approach. We first introduce the Vulnerability Unit (V-Unit) in §3.1, then summarize the two core challenges of V-Unit fuzzing in §3.2, and finally present guiding design insights in §3.3.

3.1 Key Idea: Vulnerability Unit

In software engineering, validating a target function need not require bootstrapping the entire environment. A standard practice is *unit testing* [6, 35, 36, 53, 55]: developers isolate the logic of interest, provide the dependencies required for execution, and check its behavior in a controlled scope. We apply the same principle to taint-style vulnerability detection. Rather than driving a full PHP web application to a target sink, we isolate the logic that defines the sink input space Ω_{sink} and verify that logic directly with concrete payloads. Code that only helps reach the sink but does not affect its input space, such as routing logic or unrelated business workflows, is kept outside the verification loop.

Definition 3 (Vulnerability Unit). To operationalize this idea, we define the Vulnerability Unit (V-Unit). A V-Unit is a synthesized, independently executable artifact that preserves the logic defining Ω_{sink}. Unlike a static code snippet, a V-Unit can be executed with concrete inputs. Unlike a full application, a V-Unit excludes sink-irrelevant reachability barriers (e.g., routing mechanisms and functional checks). Formally, a V-Unit u is defined as $u = \langle \mathcal{S}, \mathcal{E} \rangle$:

- *The Skeleton ($\mathcal{S}$):* The Skeleton is the structural backbone of the V-Unit. It is a focused program slice that preserves the protections defining the sink input space Ω_{sink}. As established in Definition 1, Ω_{sink} is shaped by data-flow sanitizers ($\mathcal{D}$) and control-flow constraints ($\mathcal{C}$) along source-to-sink paths. Accordingly, the Skeleton retains these sink-defining statements and their structural relationships, while excluding reachability logic irrelevant to the sink input space. However, it cannot be executed in isolation because referenced symbols may remain unresolved.
- *The Execution Context ($\mathcal{E}$):* $\mathcal{E}$ provides the definitions required to execute $\mathcal{S}$ independently. Because the Skeleton intentionally avoids whole-program expansion, $\mathcal{E}$ supplies the missing context needed to resolve referenced symbols. Together, $\mathcal{S}$ and $\mathcal{E}$ form a self-contained executable unit for exercising the sink-defining logic outside the full application.

```
1  include "execution_context.php"; // Including minimal execution context
2  $comm = $_GET["temp_mWpseP"] . ":" . $_POST["comment"];
3  if (strlen($comm) > 200)                    // Fuzzer-controllable source
4      $comm = substr($comm, 0, 200) . "...";
5  if (strpos($comm, "http") !== false)
6      $comm = preg_replace("#https?:\/\/[^\s]+#i", "[Link Removed]", $comm);
7  $safe_pattern = "/^(?:[^<]+|<(?![\x00-\x20]*[a-zA-Z\/!?]))*$/s";
8  if (preg_match($safe_pattern, $comm, $matches)) {}
9  else {                                  // Selectively omitting sink-irrelevant branch body
10     echo render_comment_card($comm, "xss"); // Target sink
11 }
```

Figure 2: A V-Unit synthesized from the motivating example.

Benefits. Figure 2 illustrates a concrete V-Unit based on the motivating example in Figure 1. The synthesized V-Unit retains the source processing, validation guards, and branch structure that jointly define the target sink input space Ω_{sink}, while excluding route declarations and functional checks that affect reachability rather than sink verification. The symbol definitions required for execution are externalized into a lightweight file (line 1), making the target sink directly fuzzable without its surrounding monolithic scaffold. Compared to traditional detection workflows, V-Unit fuzzing offers two key advantages:

- *Compared with Static Analysis:* V-Unit fuzzing executes the actual sink-defining logic with concrete values instead of approximating Ω_{sink}. It therefore evaluates path-sensitive control-flow constraints and complex string operations through execution, avoiding precision loss from control-flow abstraction, coarse heuristics, and incomplete sanitizer modeling without relying on heavyweight static reasoning over string transformations.
- *Compared with Dynamic and Hybrid Analysis:* V-Unit fuzzing removes sink input space-irrelevant reachability barriers from the testing loop, including client-side interactions, routing resolution, and server-side functional checks. As a result, sinks that are difficult to reach under monolithic execution become directly testable. By restricting execution to sink-related inputs, V-Unit fuzzing reduces noise and substantially improves testing focus.

Scope of Claims. V-Unit fuzzing distinguishes between sink-level unsafe behavior (where Ω_{sink} admits malicious values) and end-to-end exploits (which are executable through the original application). Because VUnitFuzz targets the former, vulnerabilities exposed at the V-Unit level might appear to be false positives if external barriers currently prevent end-to-end exploitation. However, these cases differ from protection-related false positives, where the sink is protected and its actual input space does not admit malicious values. Their sink input spaces remain unsafe, and exploitation is prevented only by current external reachability barriers. We therefore characterize this subset of false positives as **dormant threats** [5, 9, 23, 38]. From the perspective of a secure software development lifecycle, the root cause remains the unsafe code pattern itself [34, 43]. External mitigations may be bypassed, misconfigured, or removed during future maintenance, which would reactivate these dormant threats [29, 40]. Therefore, unit-level exposure remains highly relevant to security, even when surrounding external barriers temporarily limit full exploitability.

3.2 Challenges

While the concept of V-Unit fuzzing is appealing, realizing it in a practical system introduces two core challenges.

Challenge I: *How to precisely and scalably extract the Skeleton?* The goal of Skeleton extraction is to isolate the code that defines the sink input space Ω_{sink} from the surrounding monolithic application. This challenge has two coupled aspects. ❶ *Accuracy.* The extracted Skeleton should preserve all and only the logic that shapes Ω_{sink}. Including sink-irrelevant code bloats the V-Unit and wastes subsequent fuzzing effort, while omitting relevant logic yields an incomplete sink input space and can admit values that would not reach the sink in the original application. This is difficult because sink-defining logic is not captured by taint propagation alone: it includes both data-flow statements and control-flow structures that validate, transform, or gate sink-bound data. In real-world PHP applications, this logic is often interleaved with unrelated business logic and may even occupy only part of a statement or branch condition. ❷ *Scalability.* For an identified source-sink pair, relevant security protections may be distributed across multiple source-to-sink paths. Naively traversing each path to collect these protections can quickly degenerate into path explosion, consuming substantial analysis time and reducing extraction efficiency. As a result, Skeleton extraction may time out on large applications, ultimately leading to false negatives during vulnerability detection.

Challenge II: *How to soundly and minimally construct the Execution Context?* Even a precise Skeleton is not executable by itself. It still depends on external definitions that must be recovered into the Execution Context ($\mathcal{E}$). This challenge also has two aspects. ❶ *Soundness.* The recovered context must provide all definitions required for the Skeleton to execute under fuzzing. Otherwise, unresolved references will terminate execution before reaching the sink, halting the V-Unit fuzzing process. Achieving this statically is particularly difficult in PHP because the required definitions can be invoked through diverse call-site forms (e.g., line 33 in Figure 1),

making analyzing their concrete targets both complex and time-consuming. ❷ *Minimality.* The recovered context should contain only what is needed for the exercised sink-relevant logic. In practice, referenced symbols are often defined within multi-purpose functions (e.g., lines 23, 26 and 29 in Figure 1). Blindly recovering entire function bodies and their transitive dependencies can trigger a cascading context expansion, pulling vast amounts of irrelevant code into the V-Unit and significantly degrading efficiency.

3.3 Key Insights

Key Insight I: ***Preserve sink-defining structure instead of solving all path semantics.*** To extract a Skeleton, `VUnitFuzz` does not need to prove the feasibility of every path or statically decide which values each guard admits. It only needs to preserve the program elements whose runtime behavior defines Ω_{sink}, namely, the data-flow statements that propagate sink-related variables and the control-flow structures whose conditions or bodies constrain them. This reframes Skeleton extraction from path-wise semantic reasoning into sink-directed slicing and materialization, directly addressing both accuracy and scalability. Specifically, for accuracy, `VUnitFuzz` identifies sink-defining program elements and materializes them through fine-grained AST-to-code synthesis, rather than copying coarse line ranges. This is especially important in PHP, where a single line may contain multiple statements and a branch condition may mix relevant and irrelevant variables. Moreover, AST-level materialization avoids pulling unrelated code into the Skeleton while still preserving node-level metadata (e.g., namespace) from the original application, which directly benefits the subsequent Execution Context recovery phase. For scalability, `VUnitFuzz` need not enumerate source-to-sink paths one by one. A single Skeleton can retain the relevant control-flow structure for a source-sink pair, while subsequent fuzzing concretely exercises different paths. This avoids the path explosion induced by path-wise semantic analysis while still preserving the protections that shape Ω_{sink}.

Key Insight II: ***The interpreted nature of PHP enables an on-demand context oracle.*** Unlike compiled languages, PHP does not require the V-Unit to be fully linked before execution. A partially completed Skeleton can begin executing, and missing definitions are exposed only when execution dynamically reaches the corresponding call site. We exploit this property to turn incompleteness into a recovery oracle: rather than predicting the entire Execution Context statically, `VUnitFuzz` lets fuzzing drive execution, collects interpreter feedback on the first missing definition encountered, recovers that definition, adds it to the Execution Context, and replays the same request. This execution-driven design helps achieve both soundness and minimality. For soundness, the interpreter reveals the concrete definition demanded by the current runtime path, while enriched runtime feedback disambiguates candidates across diverse call-site forms. For minimality, only definitions exercised by the current fuzzing path are recovered. When a recovered definition resides in a multi-purpose function, only the portions actually reached by the current execution trigger further dependency recovery; unrelated code paths within that function introduce no additional dependencies. Consequently, `VUnitFuzz` constructs an Execution Context that is sufficient for sink-directed fuzzing while remaining lightweight enough to avoid context explosion.

4 `VUnitFuzz` Design

This section presents the design of `VUnitFuzz`, whose architecture is shown in Figure 3.

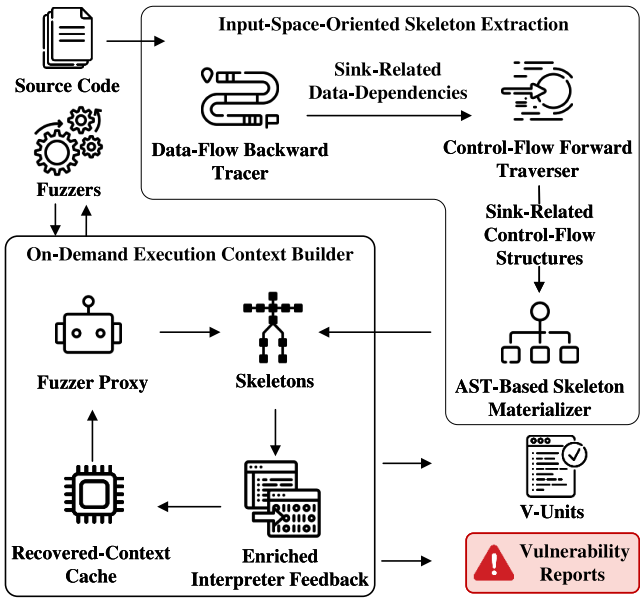

Figure 3: The architecture of `VUnitFuzz`.

4.1 Input-Space-Oriented Skeleton Extraction

This part describes how `VUnitFuzz` extracts a lightweight yet faithful Skeleton for each target source-sink pair.

General Procedure. Guided by Key Insight I, `VUnitFuzz` treats Skeleton extraction as sink-directed symbol preservation rather than path-wise semantic reasoning. It first traces backward from each sink parameter to recover sources, sink-related variables (i.e., variables on which the sink parameter directly or indirectly depends), and dependency statements, thereby forming source-sink pairs and their data-flow backbones. For each pair, `VUnitFuzz` then traverses control flow forward in a path-insensitive but branch-selective manner to retain only the constraints over sink-related variables. Finally, it materializes the retained statements and structures through fine-grained AST-to-code synthesis, avoiding line-level noise while preserving node-level metadata needed later for Execution Context recovery. The details are as follows:

Step I: Sink-Related Data-Dependency Identification. For a target sink, this step begins with targeted backward data-flow analysis from its parameter. As the analysis traverses from the sink toward one or more sources, `VUnitFuzz` maintains two registries: a variable map for tracking sink-related variables, and a statement map for recording the statements that establish such dependencies. In this way, `VUnitFuzz` isolates the data-flow transformations that shape the sink input space. Specifically, starting from the variables appearing in the sink parameter, `VUnitFuzz` iteratively locates each variable's defining statement, records that statement, and continues tracing the variables on its right-hand side. Every variable encountered along this backward chain is marked as sink-related. The trace

stops when it reaches a recognized source or no further definition can be found. The recovered sources, sink-related variables, and dependency statements together form the source-sink pair-specific data-flow backbones for the subsequent control-flow analysis.

Step II: Sink-Related Control-Flow Structure Identification. After recovering the data-flow backbone of a source-sink pair, VUnitFuzz identifies only the control-flow structures that further constrain the sink input space. To avoid path-wise semantic enumeration, it performs a path-insensitive but branch-selective forward traversal over the pair-specific sink-relevant slice. As illustrated in Figure 4, this design avoids redundant traversal while preserving the structures that define the sink input space.

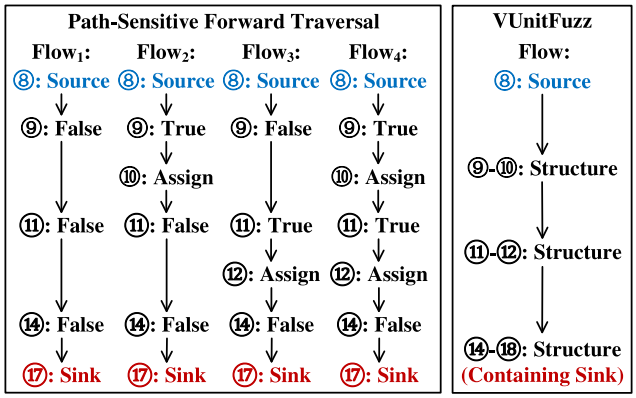

Figure 4: Sink-related control-flow traversal: path-sensitive analysis vs. VUnitFuzz on the motivating example.

Specifically, VUnitFuzz initiates forward traversal from the most upstream source within the sink-relevant slice. When multiple sources contribute to the same sink via connected logic, this starting point naturally subsumes any downstream sources. This choice prevents redundant structural exploration from those downstream sources, thereby enhancing overall efficiency. Besides, branches reachable in control flow do not always shape the sink input space. VUnitFuzz therefore preserves only branches whose conditions or bodies involve sink-related logic. During forward traversal, an encountered control-flow structure is deemed relevant only if one of the following holds:

- *Branch-condition relevance.* The branch condition explicitly validates a sink-related variable and thus constrains how source-derived values are processed before reaching the sink.
- *Branch-body relevance.* At least one branch body contains a data-dependency statement recorded in the statement map and therefore affects the sink input space.

Once such a control-flow structure is selected, VUnitFuzz retains its branch condition but preserves only those branch bodies that satisfy one of the following conditions, avoiding the introduction of irrelevant branches (e.g., the branch body at line 8 in Figure 2).

- *Branch-body modification of sink-related variables.* The branch body modifies a sink-related variable and thus determines how sink-bound data is transformed.
- *Branch-body early exit.* The branch body contains an early-exit statement (e.g., `return` or `die`) that validates sink-related variables to dictate whether execution proceeds to the sink.

For a preserved branch body, VUnitFuzz keeps only early exits and the sink-related statements already recorded in the statement map. This avoids pulling unrelated logic into the Skeleton and improves both precision and subsequent fuzzing efficiency. If a preserved branch body contains nested control-flow structures, VUnitFuzz applies the same procedure recursively.

Note that forward traversal may introduce new variables that are not sink-related by themselves but are still required to interpret a preserved control-flow constraint. We term these *constraint-supporting variables*. For example, in `if(in_array($sink_var,$list))`, `$list` does not flow into the sink but is still needed to interpret how `$sink_var` is validated. In such cases, VUnitFuzz applies the aforementioned backward tracing (in Step I) to recover the data-flow statements of these variables and preserves them in the Skeleton.

Step III: AST-Based Skeleton Materialization. Once the sink-defining symbols and structures of the current source-sink pair have been identified, this step materializes them into a local Skeleton.

Intuitively, the most straightforward idea is a line-based extraction strategy, which directly copies the lines of code corresponding to the identified data-flow statements and control-flow structures. However, given the flexibility of PHP syntax, this method is too coarse-grained. Because a single line in PHP may contain multiple statements, copying the entire line could introduce unrelated code. Furthermore, directly copying a branch condition may retain irrelevant variables, e.g., conjuncts or disjuncts introduced by `"&&"` and `"||"`, that are not actually part of the logic defining the sink input space. Therefore, VUnitFuzz adopts a finer-grained AST-to-code synthesis strategy that preserves only the relevant nodes while retaining useful node-level metadata from the original application.

Specifically, VUnitFuzz materializes a local Skeleton from the identified elements in three steps: ❶ *Root-node localization.* VUnitFuzz queries the program representation graph to locate the root node for each identified data-flow statement or control-flow structure. ❷ *Statement-level AST-to-code synthesis.* For each extracted statement root, VUnitFuzz performs a bottom-up traversal of the node and its subtree, converting nodes into a three-address-code intermediate form. This linearizes the AST and propagates child code to parents, ultimately emitting a reconstructed statement-level code fragment. Consequently, VUnitFuzz synthesizes isolated, vulnerability-relevant code without needing irrelevant surrounding context. Moreover, anchoring synthesis to the original AST nodes also preserves node-level metadata (e.g., namespaces) to facilitate subsequent definition resolution. ❸ *Scope-aware control-flow reconstruction.* To preserve the original syntactic structure of vulnerability-relevant logic, VUnitFuzz reconstructs the surrounding control-flow blocks. During extraction, it records the influence scope of selected control-flow nodes (i.e., branch-controlled statement ranges). During materialization, a block dynamically opens when encountering the first retained statement within its scope, and closes upon reaching an outside statement or exhausting the node set. This preserves essential branch structures representing the sink input space while omitting sink-irrelevant statements.

Table 1: PHP call-site categories, their native diagnostics, and the corresponding argument-enriched interpreter hooks.

Call-Site Category	Code Example	Native Diagnostic Message	Argument-Enhanced Interpreter Hook
Direct Function call	`foo()`	Call to undefined function <namespace>\<func>	zend_undefined_function_helper_SPEC
Static method invocation	`A::foo()`	Call to undefined method <namespace>\<class>::<method>	zend_undefined_method
Instance method invocation	`$a->foo()`	Call to undefined method <namespace>\<class>::<method>	zend_undefined_method
Dynamic function call	`$f()`	Call to undefined function <namespace>\<func>	zend_init_dynamic_call_string
Dynamic method invocation	`$a->$f()`	Call to undefined method <namespace>\<class>::<method>	zend_undefined_method
Dynamic static method invocation	`A::$f()` or `$a::foo()`	Call to undefined method <namespace>\<class>::<method>	zend_undefined_method
Indirect callable invocation	`call_user_func([$a, "foo"])`	Function <namespace>\<func> not found	zend_is_callable_ex

4.2 On-Demand Execution Context Builder

After Skeleton extraction, the artifact may still contain unresolved references, especially function calls and method invocations whose concrete definitions are difficult to recover statically under PHP's diverse call-site forms (Table 1). Moreover, blindly recovering candidates risks cascading context expansion from unrelated transitive dependencies. VUnitFuzz therefore makes Execution Context Builder execution-driven, recovering and injecting only the external definitions demanded by the current V-Unit execution.

General Procedure. Guided by Key Insight II, VUnitFuzz uses PHP's interpreted execution as a recovery oracle. Instead of predicting all definitions before fuzzing begins, VUnitFuzz follows a concrete-execution-driven workflow: it executes the current Skeleton, lets the PHP interpreter expose missing definitions at runtime, locates and injects the corresponding definition into an auxiliary context file, and then replays the same request. As an optimization, recovered contexts are cached and reused across different Skeletons. This completion–replay loop continues until the current request can execute without unresolved-reference errors.

Design I: Execution-Driven Missing-Definition Recovery. We first describe how VUnitFuzz resolves missing definitions once execution exposes them. These runtime recovery tasks mainly involve unresolved callees, variables, and environment-dependent values.

(1) *Unresolved Callee Recovery.* VUnitFuzz avoids statically inferring the concrete receiver objects, callable values, or target classes hidden behind diverse PHP call-site categories. Instead, it relies on execution to reveal these values precisely at runtime using the existing variables within the Skeleton. When the interpreter reaches an unresolved callee, its native diagnostics, summarized in Table 1, expose the concrete function name or class–method pair demanded by the current execution. For instance, processing the function call at line 17 in Figure 1 involves a dynamic function call (line 33), a notorious challenge for static analysis. However, during dynamic execution, the variable defined at line 22 is automatically concatenated and executed. This eliminates the need to infer the semantics of the dynamic call at line 33; instead, direct interpreter feedback enables VUnitFuzz to accurately identify the target function (line 38), thereby facilitating further recovery while avoiding expensive static reasoning. Furthermore, this design effectively prevents the recovery of entire multi-purpose functions. Since only the `"xss"` mode branch of `render_comment_card()` (line 32) is executed within the current sink context, VUnitFuzz does not recover contexts for any other branches (lines 23, 26, 29), thereby avoiding substantial irrelevant context expansion and significantly improving efficiency.

However, name-level feedback alone may still match multiple definitions in a large codebase, e.g., when different functions share the same name. Traditional static analysis handles such ambiguity conservatively by retaining multiple candidates and analyzing them approximately [33, 41, 49], introducing unnecessary noise. VUnitFuzz instead combines runtime and structural cues: (1) the namespace of the call site and (2) the number of arguments observed at runtime. When querying the program-representation graph, VUnitFuzz therefore requires the function name and namespace to match, and then filters the remaining candidates to signatures compatible with the observed argument count.

Unfortunately, native PHP diagnostics do not expose all of this information, especially the runtime argument count. To bridge this gap, VUnitFuzz instruments the argument-enriched interpreter hooks in Table 1, augmenting native diagnostics with runtime metadata. In contrast, namespace information requires no extra instrumentation because AST-based Skeleton materialization preserves the original AST node, allowing the call-site namespace to remain in the Skeleton and surface through native diagnostics. With this enriched feedback, VUnitFuzz follows a structured resolution procedure. For method invocations, it extracts the target class and method definitions into the auxiliary context file. For functions calls, it matches the name and namespace, utilizing the observed argument count to disambiguate and inject the correct definition.

(2) *Unresolved Variable Recovery.* Compared with unresolved callee recovery, variable recovery is substantially simpler. The preceding Skeleton inherently preserves most variable definitions identified through backward data-dependency analysis. Nevertheless, a few variables may remain unresolved due to incomplete data-flow edges in the program-representation graph, e.g., global variables. Therefore, when a variable read lacks an in-unit definition, VUnitFuzz performs a namespace-aware lookup across the graph to retrieve matching global-scope definitions when available.

(3) *Temporary Variable Placeholder.* When no suitable variable definition can be found, or when execution reaches environment-dependent operations (e.g., filesystem accesses), reconstructing the missing environment would pull the V-Unit back toward monolithic execution. VUnitFuzz therefore follows existing practice [10, 32, 41] and replaces such values with temporary placeholders (e.g., replacing `$_SESSION['uid']` in line 2 of Figure 2). These placeholders are not new attacker-controlled sources and are not used for payload mutation; they only keep sink-defining logic executable.

Design II: On-Demand Completion Loop During Fuzzing. The recovery tasks above define how VUnitFuzz resolves missing definitions once execution exposes them. We now describe how these recovery actions are orchestrated during fuzzing. Overall, VUnitFuzz dynamically constructs the Execution Context to the exact extent required by the fuzzing requests.

Specifically, VUnitFuzz places a proxy between the fuzzer and the Skeleton to intercept responses and monitor for recovery-relevant runtime events (i.e., interpreter diagnostics). If no such event is present, the proxy forwards the response directly to the fuzzer. Otherwise, it invokes the aforementioned recovery methods, updates the auxiliary context file included by the Skeleton, and immediately replays the same fuzzing request against the updated V-Unit. Because PHP is interpreted, this update requires no recompilation or relinking. This completion–replay loop continues until the current request executes without unresolved-reference errors, at which point the proxy returns the final response to the fuzzer.

In real-world applications, developers frequently reuse the same custom-defined functions to perform security validation across different parts of the application. As a result, the same unresolved references often recur in different Skeletons. Repeatedly recovering these shared definitions, along with the dependencies they transitively induce, is unnecessary and only increases analysis overhead. VUnitFuzz therefore reuses recovered symbols across different Skeletons instead of repeatedly resolving the same dependencies from scratch. Specifically, after each recovery step, VUnitFuzz uses the corresponding interpreter diagnostic to construct a lightweight fingerprint, i.e., the MD5 hash of the reported namespace, function name/class–method pair, and argument count. It then stores the recovered context in a cache using this fingerprint as the key. Later, if another Skeleton triggers the same runtime event, VUnitFuzz can directly reuse the cached context instead of repeating the graph query and extraction, thereby significantly improving efficiency.

4.3 Vulnerability Unit Fuzzing

V-Unit fuzzing runs on top of the on-demand completion loop described above: fuzzing drives execution, and execution-triggered recovery incrementally prepares the V-Unit for the current input. With this execution model, VUnitFuzz reuses existing state-of-the-art vulnerability-oriented fuzzing strategies and bug oracles to test whether the sink input space represented by the V-Unit overlaps with the malicious input space. Because the V-Unit has already eliminated sink-irrelevant request parameters, reachability barriers, and unrelated business logic, the fuzzer can remain focused on the sink-input-space verification.

XSS-Oriented Fuzzing. For XSS-oriented V-Units, VUnitFuzz instantiates the above execution model with the state-of-the-art mutation strategy and bug oracle from XSSky [41]. Concretely, the fuzzer mutates the sink-related parameters preserved in the V-Unit using context-aware XSS payload primitives, while interpreter feedback from string comparison and string transformation operations is reused to guide further mutation. A vulnerability is reported only when the XSS oracle confirms concrete JavaScript execution.

SQLi- and CMDi-Oriented Fuzzing. For SQLi- and CMDi-oriented V-Units, VUnitFuzz instantiates the same execution model with the fuzzing strategies of Predator [48] and fault-based bug oracles of Witcher [46]. The core detection mechanism is *fault escalation*: if a fuzzer-generated input causes the SQL parser to raise a syntax error (e.g., a malformed query due to an injected quote), or causes the shell parser (i.e., dash) to report a parsing error (e.g., a malformed command due to injected shell metacharacters), the bug oracle escalates the error into a segmentation-fault signal that the fuzzer can directly detect. A vulnerability is reported only when such a fault-escalation event is triggered during V-Unit fuzzing.

5 Evaluation

Implementation. We implemented VUnitFuzz as a prototype targeting PHP web applications. Skeleton extraction builds on PHPJoy [42], and source/sink models are directly reused from ZIPPER [49]. The Execution-Context Builder instruments the PHP interpreter to capture missing-symbol feedback and hot-load recovered definitions during fuzzing. For SQLi-oriented V-Units, constructing exact database schemas, including queried tables and columns, is impractical at scale. Since the SQLi oracle only needs to determine whether an injected payload escapes the query context and triggers a syntax error, VUnitFuzz rewrites SQL queries into schema-independent templates that preserve the original query structure for dynamic execution. The prototype consists of 8,853 lines of code: 4,507 for Skeleton extraction, 3,418 for execution-context building, and 928 for fuzzing reuse.

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

- RQ1: How effectively does VUnitFuzz extract V-Units from popular PHP web applications? (in §5.1)
- RQ2: How effective is VUnitFuzz in detecting vulnerabilities in popular PHP web applications? (in §5.2)
- RQ3: How does VUnitFuzz compare with state-of-the-art vulnerability detection approaches? (in §5.3)
- RQ4: How do the different components of VUnitFuzz contribute to its success? (in §5.4)

Dataset. To build a representative dataset, we consolidate the evaluation targets used by recent taint-style vulnerability detection techniques, including static analysis (ZIPPER [49] and XSSky [41]), dynamic analysis (Witcher [46] and Atropos [20]), and hybrid analysis (Predator [48]). This yields 50 PHP applications, as detailed in the Appendix-Table 5. The dataset is broad and fair because it unifies the target spaces of recent state-of-the-art techniques.

Setup. All experiments were conducted on an Ubuntu 20.04 virtual machine allocated with 24 cores from an Intel Xeon Gold 6242 processor and 200 GB of memory. All V-Units and fuzzers ran under Apache 2.4.52 and PHP 7.4.0.

5.1 RQ1: Vulnerability Unit Extraction

In this experiment, we evaluate whether VUnitFuzz can extract executable and faithful Vulnerability Units (V-Units).

Metrics. To better quantify and evaluate the effectiveness of V-Unit extraction, we established four metrics:

(1) *Extraction Success:* The percentage of identified source-to-sink paths for which VUnitFuzz successfully extracts a Skeleton.
(2) *Executability:* The percentage of extracted Skeletons that can be completed into executable V-Units via on-demand recovery.
(3) *Compactness:* The proportion of code retained in the V-Unit relative to the original source-to-sink path context.
(4) *Completeness:* The proportion of sink-input-space-relevant protections in the original source-to-sink path code that are preserved in the extracted V-Unit, including both data-flow sanitizers and control-flow constraints.

Table 2: Evaluation of V-Unit extraction effectiveness.

Phase	Metric	Value
Full Dataset	# Source-Sink Pairs (All)	12,306
	# Extracted Skeletons (All)	12,306
	Extraction Success (%)	100.00%
	# Executable V-Units	12,262
	Executability (%)	99.64%
Sampled V-Units (n=90)	# Orig. LoC (Avg.)	732.24
	# V-Unit LoC (Avg.)	310.29
	Compactness (%)	42.38%
	# Orig. Protections (All)	74
	# V-Unit Protections (All)	74
	Completeness (%)	100.00%

Extraction and Executability. As shown in Table 2, VUnitFuzz identifies 12,306 source-to-sink pairs across the 50 applications and extracts a Skeleton for every pair, achieving 100.00% extraction success. It further completes 12,262 Skeletons into executable V-Units, yielding 99.64% executability. Manual analysis of the remaining 44 failures shows a single cause: *required execution context resides in external third-party libraries outside the analyzed codebase.* Such definitions are absent from the local application scope and are typically resolved only during deployment.

Compactness and Completeness. To evaluate compactness and completeness, we manually inspect a representative sample of 90 V-Units, selecting 45 units immediately below the median size and 45 immediately above it to avoid size bias. For each sampled V-Unit, we measure code size and identify sink-input-space-relevant protections, including data-flow sanitizers and control-flow constraints, in the original application. Specifically, as shown in Table 2, each sampled V-Unit contains 310.29 lines of code on average, compared with 732.24 lines in the original source-to-sink context. Thus, VUnitFuzz removes 57.62% of code irrelevant to defining the sink input space and retains a compactness rate of 42.38%. The sampled V-Units also preserve all 74 sink-input-space-relevant protections observed in the original code, yielding 100.00% completeness. This result shows that VUnitFuzz does not discard the data-flow sanitizers or control-flow constraints that define Ω_{sink}.

5.2 RQ2: Vulnerability Detection

In this experiment, we evaluate the vulnerability detection effectiveness of VUnitFuzz on the full dataset. For each extracted V-Unit, VUnitFuzz performs vulnerability-oriented fuzzing and reports a vulnerability only when the corresponding bug oracle is triggered. We manually inspect all reports against the original applications.

Result Overview. Overall, VUnitFuzz reports 361 potential vulnerabilities, 357 of which are manually confirmed as true positives, yielding 98.89% precision. These confirmed vulnerabilities affect 29 applications, including 232 XSS, 122 SQLi, and 3 CMDi cases; Appendix-Table 6 provides the per-application distribution. Given the associated security risks, we responsibly disclosed all confirmed vulnerabilities to the corresponding developers; to date, 70 have been acknowledged and 28 have been fixed.

Vulnerability Cause Analysis. We further analyzed the V-Units of the confirmed vulnerabilities. The results show that 230 (64.43%) confirmed vulnerabilities (154 XSS, 73 SQLi, and 3 CMDi) contain vulnerable protections inside their V-Units. Specifically, these protections comprise 25 unique control-flow constraints and 41 unique data-flow sanitizers, 29 of which are developer-defined (e.g., regular expression and keyword matching). This finding indicates that the practical challenge lies not only in the absence of protections but also in the presence of flawed protection logic.

False Positive Analysis. All four false positives reside in dormant code, e.g., code guarded by an always-false but sink-input-space-irrelevant branch (shown in Figure 5). Under strict full-application confirmation criterion, we count these reports as false positives because they cannot currently be triggered in the deployed application. Nevertheless, they still indicate unsafe sink-level code that should be debloated before future maintenance reactivates it.

```
1 $promoting = false;
2 if(isset($config["isPromoting"]))
3     $promoting = $config["isPromoting"]; // Default by False
4 ...
5 if($promoting) echo $_POST["toRow"];
```

Figure 5: A dormant XSS sink detected in WebChess.

5.3 RQ3: Vulnerability Detection Comparison

In this experiment, we compare VUnitFuzz against representative state-of-the-art approaches for taint-style vulnerability detection.

Baselines. We compare against static (ZIPPER [49], XSSky [41]), dynamic (Witcher [46]), and hybrid (Predator [48]) baselines. Atropos is excluded because its repository notes that the current configuration is hard to setup [2]. Because XSSky originally targets only XSS, we extend it to SQLi and CMDi using the same source/sink models and fuzzing engines as VUnitFuzz, yielding XSSky+. Similarly, we follow [48] to add XSS support to Witcher, yielding Witcher+. Each tool runs with its original artifact and default configuration under a 24-hour budget per application. Since Witcher+ and Predator require full deployments, we evaluate them only on the 29 applications where at least one static baseline confirms a vulnerability, following the recall-oriented assumption that static analysis provides broader code coverage.

Ground Truth. Enumerating all vulnerabilities in real-world PHP applications is impractical. Following established practice [41, 46, 48], we construct an aggregated ground truth from the union of manually confirmed reports produced by all evaluated tools. In total, the ground truth comprises 375 vulnerabilities, including 250 XSS, 122 SQLi, and 3 CMDi. A breakdown of the detection results across all tools is provided in the Appendix-Table 6.

Result Overview. Table 3 shows that VUnitFuzz confirms 357 vulnerabilities, achieving the highest recall (95.20%) while maintaining 98.89% precision. Compared to the baselines, VUnitFuzz improves precision by up to 370.68% and recall by up to 2129.51%. Notably, VUnitFuzz is also efficient: its 100.06-hour execution time is higher only than purely static ZIPPER and remains substantially lower than the other fuzzing-based baselines.

Table 3: Comparison of VUnitFuzz against state-of-the-art work in vulnerability detection effectiveness and efficiency.

Baseline	TP	FP	FN	Prec. (%)	Recall (%)	Time (h)
ZIPPER	313	1,177	62	21.01%	83.47%	11.77
XSSky+	233	153	142	60.36%	62.13%	232.86
Witcher+[1]	16	0	359	100.00%	4.27%	476.91
Predator[1]	34	0	341	100.00%	9.07%	262.10
VUnitFuzz	357	4	18	98.89%	95.20%	100.06

[1] Due to the high effort required to deploy target applications, tools are evaluated only on the 29 applications for which at least one static-based tool reports a vulnerability.

False Negative Analysis of VUnitFuzz. Despite detecting the most vulnerabilities, VUnitFuzz still misses 18 ground-truth cases. All misses stem from fuzzing limitations: VUnitFuzz extracts executable V-Units for them, but the fuzzer fails to trigger the vulnerabilities within the time budget. ZIPPER identifies these cases only by treating their vulnerable protections as symbols, yielding accidental rather than faithful detections.

Compared with ZIPPER & XSSky+. ZIPPER represents a purely static design: it achieves broad code coverage and the lowest running time, confirming 313 vulnerabilities in 11.77 hours. However, this coverage comes at the cost of imprecise sink-input-space verification: ZIPPER reports 1,177 false positives, yielding only 21.01% precision. Our manual analysis shows that these false positives arise from four causes. First, 226 cases contain control-flow constraints that restrict the values reaching the sink, but ZIPPER ignores these protections. Second, 4 cases occur in dormant code, similar to VUnitFuzz. Third, 52 cases stem from incorrect call edges, where same-name functions or methods are resolved to the wrong callees. Finally, 895 cases are caused by sanitizer-related protections outside ZIPPER's analysis capability: ZIPPER abstracts these protections as symbols and fails to account for their security effects, so sanitized flows are still reported as vulnerable. Besides, ZIPPER also misses 62 ground-truth vulnerabilities. Among them, 36 cases contain vulnerable sanitizers. Because ZIPPER cannot faithfully analyze the implementation of these sanitizers, it relies on coarse modeling and incorrectly regards the corresponding flows as secure. The remaining 26 cases are due to its statement-level abstraction: ZIPPER treats the source as protected when a sanitizer is applied to another variable in the same statement, even though the actual source reaching the sink remains unprotected.

Although XSSky+ improves upon purely static analysis by converting identified paths into local fuzzable paths, these paths still rely on statically recovered contexts and ignore control-flow constraints. Consequently, it confirms 233 vulnerabilities but misses 142 ground-truth cases. Further analysis reveals two primary causes for its 153 false positives: 4 occur in dormant code (similar to VUnitFuzz), and 149 stem from unpreserved control-flow constraints. These latter cases predominantly affect SQLi and CMDi, where unprotected data flows remain restricted by branch conditions at the sink input space. While its false negatives arise from four issues. First, 63 cases result from timeouts; excessive irrelevant code in the recovered execution context prevents XSSky+ from generating fuzzable paths within the time budget. Second, 41 cases fail during conversion because the presence of PHP dynamic features makes it difficult to statically locate their definitions. Third, 15 cases fail because of the lack of control-flow preservation, interleaving statements from different branches, overwriting assignments, and rendering the sink parameter uncontrollable. Finally, 23 cases are missed due to fuzzing limitations.

In contrast, VUnitFuzz preserves the sink-defining control-flow structure, avoiding many false positives associated with missing constraints. Furthermore, our on-demand execution context recovery not only effectively handles PHP's dynamic features but also avoids the context explosion caused by cascading dependencies, significantly reducing time overhead (100.06h compared to 232.86h for XSSky+) and enabling the extraction of V-Units across more source-sink pairs within the vulnerability detection time budget.

Compared with Witcher+ & Predator. Witcher+ detects vulnerabilities through concrete execution, but must first crawl and drive the full application to the target sink. Consequently, it confirms only 16 vulnerabilities. Its false negatives arise across four stages. First, 19 cases are missed because Witcher's implementation is incompatible with two vulnerable applications, resulting in startup failures. Second, 153 cases are missed because front-end exploration fails to collect URLs for the target functionality. Third, 147 cases with collected URLs still cannot reach the sink due to server-side functional checks. Finally, 40 cases reach the sink but remain unconfirmed; the full request exposes numerous parameters, and the AFL-style mutation strategy cannot effectively identify and mutate the sink-related parameters.

Predator employs static guidance [33] to direct vulnerability validation, yet it still evaluates each target via monolithic application execution. Its false negatives similarly arise before or during sink validation. First, 19 cases are missed due to similar implementation incompatibilities. Second, 239 cases fail because Predator cannot construct valid entry URLs under flexible routing patterns. Third, 36 cases acquire entry URLs but fail server-side functional checks required to reach the sink. Finally, 47 cases reach the sink but remain unconfirmed; the fuzzer also fails to focus on the sink-controlling parameter, compounded by an ineffective mutation strategy.

In contrast, VUnitFuzz decouples the sink verification from monolithic execution, so V-Unit fuzzing does not require client-side UI exploration or server-side route/state setup. Moreover, V-Units naturally reduce fuzzing noise by removing irrelevant code and focusing on sink-related parameters, allowing VUnitFuzz to detect 21.31× and 9.50× more vulnerabilities than Witcher+ and Predator, respectively.

5.4 RQ4: Ablation Study

In this experiment, we evaluate the contribution of three major designs of VUnitFuzz by disabling or replacing them individually. Each variant is evaluated against the aforementioned ground truth.

Variants. We define the following variants:

- **VUnitFuzz** $_{w/o\,PB}$: VUnitFuzz without the path-insensitive and branch-selective forward traversal strategy, wherein all forward paths are expanded without optimization.
- **VUnitFuzz** $_{w/o\,OD}$: VUnitFuzz with the on-demand context builder replaced by static context extraction, where all required symbols are resolved before fuzzing.

Table 4: Ablation study results of VUnitFuzz with individual components disabled or replaced across our dataset.

Variant	V-Units		# Detected Vulns.	Precision (%)	Recall (%)	Total Time (h)			
	# Extracted	# Executable				Extracting	Converting	Fuzzing	All
VUnitFuzz $_{w/o\,PB}$	5,369	5,352	181	97.84%	48.27%	267.33	–	29.58	296.91
VUnitFuzz $_{w/o\,OD}$	7,158	7,125	253	98.44%	67.47%	38.85	170.12	20.82	229.79
VUnitFuzz $_{w/o\,CF}$	12,306	12,262	346	60.07%	92.27%	23.88	–	44.96	68.84
VUnitFuzz	12,306	12,262	357	98.89%	95.20%	38.85	–	61.21	100.06

- **VUnitFuzz** $_{w/o\,CF}$: VUnitFuzz without control-flow constraint extraction. The Skeleton retains only data-flow dependencies and omits conditional guards that protect the sink.
- **VUnitFuzz**: the full system with all components enabled.

Table 4 shows that all three designs are necessary for making V-Unit fuzzing both scalable and accurate. Disabling any component either substantially reduces recall by limiting the number of V-Units analyzed within the time budget, or substantially reduces precision by weakening the fidelity of the represented sink input space.

Contribution of Traversal Strategy. Without the path-insensitive and branch-selective traversal strategy, VUnitFuzz $_{w/o,PB}$ spends 267.33 hours on Skeleton extraction alone, which far exceeds VUnitFuzz (38.85 hours). Consequently, it extracts only 5,369 Skeletons within the time budget. This limited coverage yields only 181 confirmed vulnerabilities and a 48.27% recall, demonstrating the necessity of the strategy for scalable extraction and effective detection.

Contribution of On-Demand Strategy. VUnitFuzz $_{w/o,OD}$ statically recovers all definitions before fuzzing. This upfront approach introduces massive overhead, wasting 170.12 hours resolving execution contexts that include unneeded references. Consequently, it times out frequently, completing only 7,125 V-Units and leaving just 20.82 hours for fuzzing (compared to 61.21 hours in VUnitFuzz). By recovering missing definitions dynamically during execution, the full VUnitFuzz avoids this overhead, confirming 104 more vulnerabilities and improving recall from 67.47% to 95.20%.

Contribution of Constraint-Aware Strategy. VUnitFuzz $_{w/o\,CF}$ removes control-flow constraint extraction and therefore retains only data-flow dependencies. This makes the analysis faster, reducing the total time to 68.84 hours, but the speedup comes from omitting logic that is necessary for faithfully representing Ω_{sink}, causing false positives when branch conditions act as security protections. Moreover, the lack of control-flow preservation can interleave statements from different branches, overwriting assignments and rendering the sink parameter uncontrollable. Consequently, although VUnitFuzz $_{w/o\,CF}$ still confirms 346 vulnerabilities, its precision drops sharply to 60.07%. In comparison, the full system preserves sink-defining control-flow constraints and executes them during fuzzing, achieving much higher precision (i.e., 98.89%).

6 Discussion

Dynamic Verification of Locally Executable Program Artifacts. VUnitFuzz and XSSky share the high-level concept of dynamically verifying locally executable program artifacts instead of relying on monolithic application execution. However, they construct and complete these artifacts differently. XSSky materializes a path-level PUT for each statically identified source-to-sink path and recovers its execution context statically. In contrast, VUnitFuzz constructs one compact V-Unit for each source-sink pair. Its path-insensitive, branch-selective extraction preserves the sink-defining control-flow constraints without enumerating individual paths, while its interpreter-guided, on-demand recovery loads only the definitions required by the exercised execution. Thus, although VUnitFuzz reuses XSSky's fuzzing engine for XSS, our contribution lies in V-Unit construction and completion. As detailed in §5.3 and summarized in Table 3, the V-Unit design enables VUnitFuzz to outperform XSSky+ in both detection effectiveness and efficiency, achieving higher precision (98.89% vs. 60.36%) and recall (95.20% vs. 62.13%) while reducing the total analysis time from 232.86 to 100.06 hours.

Defensive Use of V-Unit Reports. VUnitFuzz reports sink-level unsafe behavior rather than proving end-to-end exploitability. A V-Unit report should therefore be interpreted as concrete evidence that the preserved sink-defining logic is unsafe: the sink input space Ω_{sink} still admits malicious values. From a defender's perspective, this evidence is sufficient to prioritize the reported location for inspection and remediation, even when surrounding application logic currently prevents immediate exploitation (e.g., code debloat).

Extensibility. Although our prototype targets XSS, SQLi, and CMDi, the V-Unit abstraction is not tied to these three vulnerability classes. It can be extended to other PHP taint-style vulnerabilities, such as Server-Side Request Forgery (SSRF) and arbitrary file-operation vulnerabilities, by adding the corresponding sink models and vulnerability-oriented fuzzers/oracles.

Limitations. VUnitFuzz combines the advantages of static analysis and dynamic execution through directed sink input space verification. However, like most taint-style detectors, VUnitFuzz requires predefined source/sink models and an underlying static analysis framework. While limitations in these foundations may impact overall performance, VUnitFuzz significantly improves the precision of static detectors and effectively mitigates notorious PHP features through its novel execution-driven design. Furthermore, although its vulnerability confirmation capability employs existing mutation strategies, the V-Unit concept concentrates all fuzzing efforts entirely on sink input space verification, significantly improving detection rates compared to prior work. Ultimately, VUnitFuzz remains highly adaptable; future advancements in these areas can be directly incorporated to further enhance its capabilities.

7 Related Work

Automated Unit Testing. In software engineering, automated unit testing has been widely studied for functional validation and high code coverage [19]. A key step is isolating the target code from

external dependencies through mocks, stubs, or other test doubles [17, 18]. Prior work has explored automatic mock generation and environment isolation for complex components (e.g., network services) [7, 8]. Recent advances further automate developer testing practices, including characterizing mocking decisions [55], refactoring improper test-production inheritance into proper mocking frameworks [50], and synthesizing test-passing stubs from runtime behavior [54]. However, these techniques are not directly applicable to taint-style vulnerability detection because the target of analysis is fundamentally different. Traditional unit testing assumes a developer-defined functional unit and seeks to approximate its missing environment for correctness testing. In contrast, our goal is to isolate and execute only the logic that constrains sink-bound data. Moreover, VUnitFuzz does not attempt to reconstruct the full environment through mocks; it resolves missing dependencies only when they are required during the fuzzing process. This design avoids the dependency explosion and brittleness of mock-heavy tests while enabling direct vulnerability verification.

Automated Fuzzing Harness Generation. In library vulnerability discovery, security-sensitive Application Programming Interfaces (APIs) are rarely executable in isolation and therefore require fuzzing harnesses to initialize context and invoke target APIs. Early automated approaches [22, 24, 26, 51] reconstruct API call sequences from consumer programs or unit tests using static or dynamic analysis. Since these methods synthesize contexts artificially, they often violate implicit semantic constraints and produce spurious crashes caused by API misuse rather than real vulnerabilities. Recent work [12, 30, 31] improves this process by dynamically learning or semantically inferring more valid API contexts. However, these techniques differ fundamentally from our Vulnerability Unit in both context construction and testing goals. Traditional harness generation follows a bottom-up synthesis paradigm: it fabricates calling contexts for target functions and must then distinguish real vulnerabilities from invalid API usage. In contrast, V-Unit adopts a top-down extraction paradigm for monolithic web applications. It extracts the naturally occurring program slices that constrain the sink input space. This avoids guessing inter-API dependencies and largely eliminates the API-misuse problem, so violations observed in a V-Unit map directly to high-confidence vulnerability reports.

8 Conclusion

Taint-style vulnerability detection fundamentally hinges on verifying whether malicious values can survive the protections around a sink and still reach that sink. Existing workflows either approximate this question statically or answer it only through monolithic application execution. In this paper, we propose V-Unit fuzzing, a new paradigm that shifts detection to focused verification of sink input spaces. Based on this idea, we designed VUnitFuzz, which combines sink-input-space-oriented Skeleton extraction, interpreter-guided on-demand Execution-Context construction, and vulnerability-oriented fuzzing on independently executable V-Units. Our evaluation on 50 real-world PHP applications shows that this design can extract 12,262 executable V-Units and detect 357 confirmed taint-style vulnerabilities while substantially improving effectiveness and efficiency over state-of-the-art baselines. More broadly, our results suggest that decoupling sink verification from monolithic execution is a practical and promising direction for future taint-style vulnerability detection workflows.

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 and Xiapu Luo 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] 2026. Facebook. https://www.facebook.com.
[2] 2026. Github - Source code of Atropos. https://github.com/CISPA-SysSec/atropos-legacy.
[3] 2026. Usage statistics of PHP for websites. https://w3techs.com/technologies/details/pl-php.
[4] 2026. Wikipedia. https://www.wikipedia.org.
[5] Babak Amin Azad and Nick Nikiforakis. 2023. Role models: Role-based debloating for web applications. In *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy*. 251–262.
[6] Paul Ammann and Jeff Offutt. 2017. *Introduction to software testing*. Cambridge University Press.
[7] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2014. Automated unit test generation for classes with environment dependencies. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 79–90.
[8] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2015. Generating TCP/UDP network data for automated unit test generation. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 155–165.
[9] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is more: Quantifying the security benefits of debloating web applications. In *28th USENIX Security Symposium (USENIX Security 19)*. 1697–1714.
[10] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. Fudge: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 975–985.
[11] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE european symposium on security and privacy (EuroS&P)*. IEEE, 334–349.
[12] Peng Chen, Yuxuan Xie, Yunlong Lyu, Yuxiao Wang, and Hao Chen. 2023. Hopper: Interpretative fuzzing for libraries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1600–1614.
[13] Johannes Dahse and J Schwenk. 2012. Rips-a static source code analyser for vulnerabilities in PHP scripts (2010).
[14] 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.
[15] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. 2023. Rescan: A middleware framework for realistic and robust black-box web application scanning. In *Network and Distributed System Security (NDSS) Symposium*.
[16] 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.
[17] Mattia Fazzini, Chase Choi, Juan Manuel Copia, Gabriel Lee, Yoshiki Kakehi, Alessandra Gorla, and Alessandro Orso. 2022. Use of test doubles in android testing: An in-depth investigation. In *Proceedings of the 44th international conference on software engineering*. 2266–2278.
[18] Mattia Fazzini, Alessandra Gorla, and Alessandro Orso. 2020. A framework for automated test mocking of mobile apps. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1204–1208.

[19] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.
[20] 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*.
[21] Xiangyu Guo, Akshay Kawlay, Eric Liu, and David Lie. [n. d.]. EvoCrawl: Exploring Web Application Code and State using Evolutionary Search. In *the Network and Distributed System Security Symposium (NDSS)*.
[22] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. {FuzzGen}: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*. 2271–2287.
[23] 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)*. 5557–5573.
[24] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. 2023. Utopia: Automatic generation of fuzz driver using unit tests. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2676–2692.
[25] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 6–pp.
[26] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. 2021. Winnie: Fuzzing windows applications with harness synthesis and fast cloning. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*.
[27] David Klein, Thomas Barber, Souphiane Bensalim, Ben Stock, and Martin Johns. 2022. Hand sanitizers in the wild: A large-scale study of custom javascript sanitizer functions. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 236–250.
[28] Sophie Lathouwers, Maarten Everts, and Marieke Huisman. 2020. Verifying sanitizer correctness through black-box learning: A symbolic finite transducer approach. In *Proceedings of the 6th International Conference on Information Systems Security and Privacy: Volume 1: ForSE*. SciTePress, 784–795.
[29] Triet Huynh Minh Le, Xiaoning Du, and M Ali Babar. 2024. Are latent vulnerabilities hidden gems for software vulnerability prediction? an empirical study. In *Proceedings of the 21st International Conference on Mining Software Repositories*. 716–727.
[30] Jiayi Lin, Qingyu Zhang, Junzhe Li, Chenxin Sun, Hao Zhou, Changhua Luo, and Chenxiong Qian. 2025. Automatic Library Fuzzing through API Relation Evolvement.. In *NDSS*.
[31] Yuwei Liu, Junquan Deng, Xiangkun Jia, Yanhao Wang, Minghua Wang, Lin Huang, Tao Wei, and Purui Su. 2025. PromeFuzz: A Knowledge-Driven Approach to Fuzzing Harness Generation with Large Language Models. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. 1559–1573.
[32] Yuwei Liu, Yanhao Wang, Xiangkun Jia, Zheng Zhang, and Purui Su. 2024. AFGen: Whole-Function Fuzzing for Applications and Libraries. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1901–1919.
[33] Changhua Luo, Penghui Li, and Wei Meng. 2022. Tchecker: Precise static interprocedural analysis for detecting taint-style vulnerabilities in php applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2175–2188.
[34] Gary McGraw. 2004. Software security. *IEEE Security & Privacy* 2, 2 (2004), 80–83.
[35] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code*. Pearson Education.
[36] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. 2004. *The art of software testing*. Vol. 2. Wiley Online Library.
[37] 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.
[38] Eric Olsson, Benjamin Eriksson, Adam Doupé, and Andrei Sabelfeld. 2024. SpiderScents: Grey-box database-aware web scanning for stored XSS. In *33rd USENIX Security Symposium (USENIX Security 24)*. 6741–6758.
[39] 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, 295–316.
[40] Tadeusz Pietraszek and Chris Vanden Berghe. 2005. Defending against injection attacks through context-sensitive string evaluation. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 124–145.
[41] 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.
[42] Youkun Shi, Yuan Zhang, Tianhan Luo, Guangliang Yang, Shengke Ye, Chengyu Yang, Fengyu Liu, Xiapu Luo, and Min Yang. 2025. PHPJoy: A Novel Extended Graph-based PHP Code Analysis Framework. *IEEE Transactions on Software Engineering* (2025).
[43] Murugiah Souppaya, Karen Scarfone, and Donna Dodson. 2022. Secure software development framework (ssdf) version 1.1. *NIST Special Publication* 800, 218 (2022), 800–218.
[44] Aleksei Stafeev, Tim Recktenwald, Gianluca De Stefano, Soheil Khodayari, and Giancarlo Pellegrino. 2024. YURASCANNER: Leveraging LLMs for Task-driven Web App Scanning. (2024).
[45] He Su, Lili Xu, Huina Chao, Feng Li, Zimu Yuan, Jianhua Zhou, and Wei Huo. 2022. A sanitizer-centric analysis to detect cross-site scripting in PHP programs. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 355–365.
[46] 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 (SP)*. IEEE, 2658–2675.
[47] 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.
[48] 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.
[49] Xinyi Wang, Yeting Li, Jie Lu, Shizhe Cui, Qin Mai, Yunpei Zhang, Yang Xiao, Feng Li, and Wei Huo. 2025. ZIPPER: Static Taint Analysis for PHP Applications with Precision and Efficiency. In *34th USENIX Security Symposium (USENIX Security 25)*. 8273–8291.
[50] Xiao Wang, Lu Xiao, Tingting Yu, Anne Woepse, and Sunny Wong. 2021. An automatic refactoring framework for replacing test-production inheritance by mocking mechanism. In *Proceedings of the 29th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 540–552.
[51] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. 2021. APICraft: Fuzz driver generation for closed-source SDK libraries. In *30th USENIX Security Symposium (USENIX Security 21)*. 2811–2828.
[52] 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.
[53] Hengcheng Zhu, Valerio Terragni, Lili Wei, Shing-Chi Cheung, Jiarong Wu, and Yepang Liu. 2025. Understanding and characterizing mock assertions in unit tests. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 554–575.
[54] Hengcheng Zhu, Lili Wei, Valerio Terragni, Yepang Liu, Shing-Chi Cheung, Jiarong Wu, Qin Sheng, Bing Zhang, and Lihong Song. 2023. Stubcoder: Automated generation and repair of stub code for mock objects. *ACM Transactions on Software Engineering and Methodology* 33, 1 (2023), 1–31.
[55] Hengcheng Zhu, Lili Wei, Ming Wen, Yepang Liu, Shing-Chi Cheung, Qin Sheng, and Cui Zhou. 2020. Mocksniffer: Characterizing and recommending mocking decisions for unit tests. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*. 436–447.

A Open Science

To support the reproducibility of our evaluation results, we have made our research artifacts, including the dataset and source code of `VUnitFuzz`, publicly available on Zenodo at https://zenodo.org/records/21659116.

B Ethical Considerations

We conducted this research with careful attention to verification safety, coordinated disclosure, and artifact release.

Verification Environment. All experiments were performed on locally deployed instances of open-source applications inside isolated environments. `VUnitFuzz` interacted only with these test deployments and never with production systems or real user data.

Responsible Disclosure. Whenever we manually confirmed a vulnerability, we reported it to the corresponding developers through coordinated disclosure channels, including GitHub issues or direct email when appropriate. Our reports included technical details

sufficient for validation and remediation. To avoid increasing real-world risk, this paper omits exploit details for vulnerabilities that remained unpatched at the time of submission.

Artifact Release. Because VUnitFuzz can help identify real vulnerabilities, artifact release must balance reproducibility with dual-use risk. We therefore release the research artifacts in a form that supports scientific evaluation while avoiding unnecessary exposure of sensitive exploit details for still-unpatched cases.

C Evaluation Details

In this part, we provide supplementary evaluation details.

C.1 Dataset

Table 5 details our evaluation dataset, specifying the versions of the applications used and the prior studies from which these targets were sourced. For each application, we selected the most recent version available at the time of writing.

Table 5: Applications included in our evaluation dataset, together with the selected versions and their source references.

Application	Version	Ref	Application	Version	Ref
MediaWiki	1.41.5	[20, 33]	Collabtive	3.1	[33, 48, 49]
ECI-Bootstrap	11d73a6	[33, 49]	WeBid	1.2.2	[33, 48, 49]
WordPress	6.8.3	[20, 33, 41, 46, 48, 49]	CPG	1.6.12	[33, 48, 49]
WebChess	0.9	[33, 48, 49]	Joomla	5.3.2	[33, 48, 49]
stock-mgmt.	fbbbf21	[33, 49]	PHPLiteAdmin	1.9.8.2	[33, 48, 49]
Zen-Cart	1.5.8	[20, 33, 48, 49]	Codiad	2.8.4	[33, 48, 49]
monstra	3.0.4	[33, 49]	Login Mgmt.	3.3	[46, 48]
Hosp. Mgmt.	4.0	[46, 48]	Doctor Appt.	1.0	[46, 48]
Piwigo	15.7.0	[20, 48]	rConfig	3.9.2	[46, 48]
OpenEMR	7.0.3.4	[20, 41, 46, 48]	ECShop	4.1.5	[48]
WackoPicko	cabc1b3	[46]	AltoCMS	1.1.31	[20]
Nextcloud	25.0.13	[20]	Invoice Ninja	5.5.0	[20]
Iubenda	3.12.4	[20]	lodel	1.0.6	[20]
MaxSite	1.0.9	[20]	phpwcms	1.9.47	[20]
Mini-Inv.-Sales	18aa3da	[49]	razor	0.8.0	[49]
Matomo	5.5.1	[41]	Firefly III	6.1.25	[41]
Grav CMS	1.7.49	[41]	October	4.0.3	[41]
Dolibarr	22.0.0	[41]	GLPI	10.0.19	[41]
EasyImages2.0	2.8.6	[41]	phpIPAM	1.7.3	[41]
AdminLTE	5.21	[41]	Live helper chat	4.67	[41]
Vmqphp	1.12	[41]	MantisBT	2.27.0	[41]
ProjectSend	1.9.45	[41]	Jstolpe Blog	a447ae4	[41]
Chyrp	2.5.2	[41]	Typesetter CMS	5.1	[41]
Tiny Tiny RSS	a423423	[41]	phpBB	3.3.15	[20, 33, 46]
osCommerce2	2.3.4.1	[33, 49]	LiteCart	2.6.2	[41]

C.2 Detected Vulnerabilities by Evaluated Tools

Table 6 reports the per-application distribution of true-positive vulnerabilities confirmed by each evaluated tool.

Table 6: Per-application distribution of true-positive vulnerabilities detected by evaluated tools.

Application	VUnitFuzz	XSSky+	ZIPPER	Witcher+	Predator
Collabtive	1	1	1	0	0
ECI-Bootstrap	28	28	28	0	0
CPG	1	1	1	0	0
WebChess	33	29	32	0	0
Zen-Cart	34	10	23	0	0
Codiad	31	27	31	0	4
rConfig	7	4	0	3	0
OpenEMR	5	0	4	0	0
ECShop	24	5	24	0	0
WackoPicko	5	5	5	3	0
MaxSite	2	2	0	0	0
phpwcms	8	6	6	0	0
razor	1	1	1	0	0
GLPI	14	8	7	0	0
EasyImages2.0	7	3	4	0	1
AdminLTE	1	1	1	0	0
Vmqphp	2	2	2	1	1
ProjectSend	12	10	10	0	0
Jstolpe Blog	3	3	2	0	0
Chyrp	7	4	8	0	0
Typesetter CMS	14	7	7	0	0
LiteCart	6	0	10	0	0
Tiny Tiny RSS	23	9	34	0	0
Login Mgmt.	16	7	0	3	13
Hosp. Mgmt.	50	48	50	4	10
Doctor Appt.	12	12	12	2	5
osCommerce2	5	0	5	0	0
Piwigo	3	0	3	0	0
MantisBT	2	0	2	0	0
Total	**357**	**233**	**313**	**16**	**34**