Reproducing Web Application Vulnerabilities with Patch-Guided Routing Inference and Sink Exploration

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

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

Feng Xue
Fudan University
Shanghai, China
23212010038@m.fudan.edu.cn

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

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

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

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

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

Abstract

Vulnerability reproduction is essential for verifying vulnerability existence, assessing impact, and validating patch effectiveness. Till now, patch-based directed fuzzing has been embraced as the mainstream approach to achieve this goal, which takes a known security patch as input to attempt to generate the corresponding Proof-of-Concept (PoC). However, its effectiveness in the PHP web ecosystem is severely limited by two critical gaps: (1) the routing challenge, where flexible and fragmented routing semantics in modern applications make it difficult to construct valid entry URLs; and (2) the sink challenge, where security patches typically implement upstream checks rather than modifying the sinks, which are often shielded by deep encapsulation.

In this paper, we propose P2PGen, a novel patch-guided PoC generation framework dedicated to PHP web applications. To navigate complex routing, we design an LLM-powered agent augmented with a reachability beacon-feedback mechanism, allowing the agent to autonomously self-correct routing hypotheses based on execution feedback. To bridge the gap between patches and sinks, we introduce forced path exploration coupled with an interpreter-level sink monitor based on vulnerability indicator functions. This approach enables deterministic traversal of downstream execution paths and precise detection of encapsulated sinks.

We evaluated P2PGen on 140 known vulnerabilities across 15 popular PHP applications. It successfully generated PoCs for 124 instances (88.57%), achieving a 95.95% reproduction success rate on 74 ground-truth cases. Notably, P2PGen significantly outperformed the state-of-the-art directed fuzzer, Predator, by 407.14%. Beyond reproduction, P2PGen identified 8 incomplete patches and 10 potential new vulnerabilities, 7 of which remain exploitable in the latest version (5 XSS, 2 SQLi), yielding 2 CVEs to date.

CCS Concepts

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

Keywords

Vulnerability Reproduction, Web Security

ACM Reference Format:
Youkun Shi, Yuan Zhang, Feng Xue, Jiarun Dai, Lei Zhang, Fengyu Liu, Bocheng Xiang, and Xiapu Luo. 2026. Reproducing Web Application Vulnerabilities with Patch-Guided Routing Inference and Sink Exploration. 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/10.1145/3830454.3832678



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/10.1145/3830454.3832678

1 Introduction

PHP web applications serve as the backbone of the modern World Wide Web, powering 72.9% of all websites [14]. However, this widespread deployment makes them a primary target for cyber threats. Critical vulnerabilities such as Cross-Site Scripting (XSS), SQL Injection (SQLi), and Command Injection (CMDi) persistently threaten data security and user privacy.

To manage these threats, vulnerability reproduction is indispensable. Central to this process is the vulnerability Proof-of-Concept (PoC) [49], which serves as a crafted input (e.g., an HTTP request) designed to navigate routing logic and deterministically trigger the vulnerable function (i.e., the sink). These PoCs empower practitioners to verify vulnerability existence, assess impact scope, and validate patch effectiveness. Nevertheless, prior research [49, 57]

indicates that vulnerability PoCs are often unavailable. Our evaluation dataset also corroborates this: among 140 CVEs across 15 popular PHP web applications, 47.14% lacked functional PoCs.

To mitigate this scarcity, directed greybox fuzzing has emerged as the mainstream approach for automated PoC generation. Unlike coverage-guided fuzzing [29, 31, 50, 51, 58, 59], which blindly maximizes code coverage, directed fuzzers steer execution toward designated target sites [41, 60, 70], making them significantly more efficient for vulnerability reproduction. However, their success hinges on two critical elements: the vulnerability entry point and the sink.

For a target vulnerability, an intuitive idea to obtain these elements is to analyze its descriptions from public vulnerability databases. However, these descriptions are often ambiguous (e.g., failing to specify vulnerable files or containing inaccuracies) [27, 53]. To address this, we turn to a more reliable vulnerability information: security patches. Specifically, the code diffs introduced by patches offer critical insight. Since the patched code invariably lies on the vulnerability's execution trace, it serves as a pivot point for locating the entry upstream and the sink downstream, naturally offering a precise, developer-validated roadmap [55–57]. Nevertheless, security patches-guided directed fuzzing still requires addressing two main challenges to succeed in vulnerability reproduction:

❶ *How to construct the correct Entry URL to initiate directed fuzzing?* Modern web applications feature a multitude of potential entrances (URLs). Constructing a valid entry URL is a prerequisite for PoC generation. However, this process is severely hindered by the inherent complexity of the application. First, pinpointing the vulnerability entry based on patch analysis requires meticulous handling. Unlike legacy direct-script execution, modern MVC frameworks utilize complex controller-based dispatching, where a single controller file encapsulates multiple distinct action methods. Coarse-grained approaches [60, 70] that merely identify a file as the entry will introduce significant noise. Second, synthesizing a valid entry URL to initiate directed fuzzing is complicated. This is because the routing semantics of modern web applications are rarely centralized; instead, they are scattered across varying layers—including configuration files, class or method annotations, and code logic. Reconstructing a valid URL often requires synthesizing these intertwined layers. These complexities force existing approaches to either rely on manually provided entry URLs [41] or fail to construct correct ones automatically [60, 70], rendering all subsequent fuzzing efforts futile from the outset.

❷ *How to efficiently locate the vulnerability sink to guide the fuzzer towards the target?* Even with a correct entry URL, vulnerability reproduction also necessitates a precise destination: the vulnerability sink. However, locating the sink via patch analysis is non-trivial, as two critical problems remain. First, the patch is rarely the sink itself. Patches typically introduce upstream validation rather than modifying the sink, which often resides far downstream behind complex control flows. This gap forces patch-targeted fuzzers to fall back on blind exploration to locate the downstream sink upon reaching the patched code, resulting in significant inefficiency. Second, sinks are deeply encapsulated. Modern applications frequently wrap built-in sinks (e.g., `query()`) within multiple layers of abstraction, such as data access objects (DAOs) and database abstraction layers (DBALs), making identification via static analysis [47, 57] expensive and error-prone.

In light of this, we propose a novel framework, named P2PGen (i.e., Patch to PoC Generation). Leveraging security patches as input, P2PGen addresses the two aforementioned challenges, and empowers existing directed fuzzing techniques (e.g., Predator [62]) to generate PoCs for vulnerability reproduction. Specifically, to tackle the challenge of entry URL generation, P2PGen employs an LLM-powered approach augmented with a novel reachability beacon-feedback mechanism. Instead of relying on ambiguous HTTP status codes, we instrument a deterministic beacon at the code entry point. This allows the LLM to utilize execution feedback as a precise oracle, autonomously inferring complex routing semantics and iteratively self-correcting invalid URLs. To tackle the challenge of efficiently locating the vulnerable sink downstream from the patched code, P2PGen utilizes a novel Forced Path Exploration (FPE) strategy, which dynamically modifies branch conditions (e.g., forcing `True` / `False`) to deterministically traverse all execution paths. To overcome the inherent path explosion problem, we introduce Sequential Unit Analysis. By decomposing logic into sequential execution units, this approach effectively linearizes the search space. Furthermore, to identify sinks deeply encapsulated within function wrappers, we implement an interpreter-level Vulnerability Indicator Function (VIF)-based sink monitor, ensuring precise sink discovery.

To evaluate P2PGen, we implemented a prototype based on Predator [60], a state-of-the-art directed fuzzer for PHP web applications. We evaluated the prototype using a representative dataset of 140 known vulnerabilities across 15 popular PHP web applications. The results demonstrate that P2PGen successfully generated Proofs-of-Concept (PoCs) for 124 instances (88.57%). Verification against a ground truth of 74 vulnerabilities with public PoCs confirmed that P2PGen accurately reproduced 71 cases, achieving a 95.95% success rate. Ablation studies indicate that our novel LLM-powered entry URL generation improved the base model's precision and recall by 138.72% and 132.28%, respectively. In baseline comparisons, P2PGen significantly outperformed Predator, increasing the reproduction count from 14 to 71 (a 407.14% improvement). Beyond reproduction, P2PGen identified 8 incomplete patches and 10 potential new vulnerabilities, 7 of which remain exploitable in the latest versions (5 XSS, 2 SQLi), yielding 2 CVEs to date.

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

- We propose P2PGen, a patch-guided framework for automated PoC generation. By integrating LLM-powered routing inference with forced path exploration, P2PGen effectively addresses critical challenges in entry URL construction and sink localization.
- We evaluated P2PGen on 140 vulnerabilities across 15 popular PHP applications, successfully generating PoCs for 124 instances (88.57%). When validated against a ground truth of 74 cases with public PoCs, P2PGen achieved a 95.95% reproduction success rate (71/74), outperforms the state-of-the-art work, Predator, by 407.14% (71 vs. 14 reproductions).
- Beyond reproduction, P2PGen identified 8 incomplete patches and 10 new vulnerabilities, 7 of which remain exploitable against the latest versions. To date, we have been assigned 2 CVEs.

2 Background

To provide the necessary background, this section explores the diverse routing mechanisms (§2.1) and discusses related work (§2.2).

2.1 PHP Web Routing Mechanisms

In web applications, a primary method for accessing the functionalities is by visiting their corresponding URLs. The mechanism by which an application translates an incoming URL into the execution of specific code is known as routing.

In legacy or simpler PHP applications, routing often follows a **direct-path mapping model**. In this model, the URL's path directly corresponds to a PHP script's location on the server's file system. For example, a request to *https://example.com/admin/delete.php?id=1* would execute the */admin/delete.php* script and pass *id=1* as a GET parameter, which serves as a user-controllable variable during program execution. However, this approach has been largely superseded by modern software engineering practices.

Modern PHP applications feature far more flexible routing mechanisms, which are either dictated by frameworks or customized by developers. Analysis of our evaluation dataset, comprising 15 popular PHP applications, confirms this trend: only two retain the simple direct-path mapping. The remaining applications (13/15, 86.67%) adopt diverse advanced routing mechanisms, which we classify into four primary patterns, as illustrated in Figure 1.

```
// Parameter-Based Routing (e.g., PrestaShop)
https://example.com/[handler]?controller=class&method=method&param=value

// Clean URL-Based Routing (e.g., MISP)
https://example.com/[handler]/class/method/param_value1/param_value2/...

// Annotation-Based Routing (e.g., Pimcore)
@Route("/index/statistics", name="pimcore_admin_index_statistics"
, methods={"GET"})

// Custom-Defined Routing (e.g., osTicket)
url('^/queue/', patterns('ajax.search.php:SearchAjaxAPI',
                         url('^(?P<id>\d+/)?preview$', 'previewQueue')))
```

Figure 1: Pattern examples of modern routing mechanisms.

(1) **Parameter-Based Routing** is a common implementation of the Front Controller pattern, where all incoming requests are routed through a centralized handler script (e.g., *index.php*). This script is then responsible for dispatching the request to the appropriate controller and function based on query parameters.
(2) **Clean URL-Based Routing** has been adopted by modern frameworks and utilizes so-called "Clean URLs" [16]. Instead of relying on query strings, the application's routing engine parses the URL path itself. Each segment of the path is mapped to a specific component, such as a class, a method, and parameters.
(3) **Annotation-Based Routing** allows developers to define routes within the annotations of a controller class or action method, which are subsequently parsed into the routing table.
(4) **Custom-Defined Routing** allows developers to implement highly customized routing logic, e.g., using regular expressions to match and parse incoming URLs.

Notably, routing analysis is further complicated by rules dispersed across diverse layers, including configuration files, source code, and annotations. Moreover, distinct routing patterns could coexist within a single application; for instance, MISP [10] simultaneously employs clean URL-based routing and custom-defined routing, thereby necessitating careful, fine-grained analysis.

2.2 PHP Web Application Fuzzing

Fuzzing is a dynamic software testing technique that involves providing automatically generated, invalid, or unexpected inputs to a program to discover software defects and bugs. In the context of web applications, fuzzing approaches can be broadly classified into two main types: coverage-guided and directed fuzzing.

Coverage-Guided Fuzzing. This line of work [31, 50, 58, 59] leverages code coverage information collected during fuzzing to guide the fuzzer's input mutations. The goal of these approaches is to maximize code exploration throughout the fuzzing process, aiming to uncover potential security bugs. However, while effective in validating more code locations, this untargeted approach suffers from inherent inefficiency. This limitation stems from the typically uneven distribution of vulnerabilities within an application; a coverage-guided fuzzer may inadvertently expend significant resources on non-critical areas, making it difficult to reach deep vulnerabilities hidden within allocated time.

Directed Fuzzing. In contrast to the broad exploration of coverage-guided fuzzing, this line of work [60, 70] aims to efficiently steer the fuzzer toward specific, predefined code locations. By maintaining a clear target, these techniques avoid exhaustive exploration of vast search spaces, focusing resources on specific targets. Nevertheless, they face the fundamental challenge of handling contemporary flexible routing mechanisms. Unlike coverage-guided fuzzing, which relies on crawlers to collect URLs over extended periods to initiate testing, directed fuzzing must autonomously construct reachable entry URLs based on target locations. This requires a semantic understanding of route resolution. Unfortunately, existing directed approaches [60, 70] are confined to rudimentary direct-path mapping model, which are now obsolete.

3 Problem Statement

This section first presents the rationale for choosing patches for PoC generation (§3.1), then motivates our work with a real-world example, and outlines the key challenges (§3.2).

3.1 Rationale for Choosing Security Patches

Deconstructing the PoC Generation Task. One can conceptualize a modern web application as a massive, convoluted labyrinth. It features a multitude of potential entrances (URLs) and an exponential number of execution branching paths. To efficiently generate a PoC for vulnerability reproduction within this complex structure, we must fundamentally answer three pivotal questions:

(1) *Which entrance serves as the correct starting point?*
(2) *Where is the vulnerability sink destination located?*
(3) *How to efficiently traverse from the entrance to the destination?*

Regarding question (3), prior research has extensively explored this challenge within the context of directed fuzzing (e.g., distance-guide) [21, 48, 60]. Therefore, the primary bottleneck shifts to acquiring precise answers for questions (1) and (2).

Acquiring Clues for Entrance and Destination. To address this, we should determine which publicly available vulnerability information provides reliable clues to pinpoint its entrance and destination. Intuitively, one might look to vulnerability descriptions (e.g., CVE entries). However, as noted in previous studies [27, 53], these descriptions are often ambiguous. They typically describe

the impact of a flaw rather than its causes. For instance, as shown in Figure 2 (a), the description for CVE-2023-1578 specifies only the vulnerability type and affected versions, providing no clues to facilitate analysis of the vulnerability causes.

Strong Clues in Security Patches. In contrast to ambiguous descriptions, security patches are a superior choice for two primary reasons. First, as an essential product of the vulnerability fixing process, security patches are always readily accessible from public repositories, ensuring a reliable data source. Second, and more critically, the code changes (diffs) introduced by a patch offer a precise, developer-validated roadmap to the vulnerable code. Specifically, since these modifications are intended to fix the vulnerability, they are inherently situated within the vulnerability exploit's execution flow. Consequently, the patch acts as a natural pivot point that provides the missing semantic clues:

- By performing *backward analysis* from the patch modification, we can trace the vulnerability entrance.
- By conducting *forward exploration* from the patch modification, we can locate the vulnerability sink destination.

3.2 Motivating Example and Challenges

Here, we analyze a real-world vulnerability and its security patches to illustrate the inherent challenges of patch-based PoC generation, demonstrating why existing state-of-the-art techniques fail.

Real-World Example. Figure 2 illustrates a disclosed SQL injection vulnerability (CVE-2023-1578) identified in the Pimcore (3.7k stars in Github). This flaw resides in the search functionality, where the failure to validate the user-controlled `$types` parameter permits SQL injection. As shown in Figure 2 (c), developers fixed this issue by implementing `preg_replace()` to sanitize the data flow (line 10). Notably, during vulnerability reproduction, we found that this patch is incomplete and a detailed case study is provided in §6.4.1.

Patch-Based PoC Generation Task. Accomplishing this task entails two primary steps: (1) analyzing patch clues and routing mechanisms to identify the entry point and construct a reachable entry URL; and (2) pinpointing the specific vulnerability sink. Subsequently, the entry URL serves as the initial input for the directed fuzzer, which targets the designated sink to generate the PoC. However, executing these steps presents the following challenges.

Challenge I: How to construct the correct Entry URL to initiate directed fuzzing? Constructing a valid entry URL is a prerequisite for PoC generation. However, this process is hindered by the complexity of modern web applications, specifically in two aspects.

❶ *Fine-grained code-level entry identification.* Modern web applications, heavily reliant on MVC frameworks (e.g., Symfony, Laravel), have shifted from direct script execution to complex controller-based dispatching. Consequently, identifying the entry point requires a fine-grained understanding of the code structure. Coarse-grained approaches [60] that simply treat the file identified via patch backtracking as the entry script are often ineffective. For example, a single controller file (e.g., `SearchController.php` in Figure 2) often contains multiple action methods, each handling distinct functionalities with different routing contexts. Merely designating the file as the entry point introduces significant analysis noise and may fail to capture the specific routing logic associated with the actual entry method (e.g., `findAction()`, line 5).

CVE-2023-1578

SQL Injection in GitHub repository pimcore/pimcore prior to 10.5.19.

a) CVE Description

```
_pimcore_search:
  resource: "@PimcoreAdminBundle/Controller/Searchadmin/"
  type:     annotation
  prefix:   /admin/search
  options:
    expose: true
```

b) Routing File: Resources/config/pimcore/routing.yml

```
1  /*@Route("/search")*/
2  class SearchController extends AdminController {
3     ...
4     /*@Route("/find", name="search_find",
                                  methods={"GET", "POST"})*/
5     public function findAction(Request $request,... ){
6        echo "reach_e238151d42ca1ccf8659e3deae61b014";
                                              // The Inserted Beacon
7        ...
8        $allParams = $request->request->all();
9        ...
10 +     $types = explode(',', preg_replace("/[^a-z,]/i", ''
                                          ,$allParams['type'] ?? ''));
11       $conditionParts = [];
12       $searcherList = new Data\Listing();
13       ...
14       if (is_array($types) and !empty($types[0])) {
15          $conditionTypeParts = [];
16          foreach ($types as $type)
               $conditionTypeParts[] = $db->quote($type);
17          $conditionParts[] = '( maintype IN (' . implode(',',
                                          $conditionTypeParts) . ') )';
18       }
19       ...
20       if (count($conditionParts) > 0)
            $condition = implode(' AND ', $conditionParts);
21       ...
22       $hits = $searcherList->load($condition);  // Vulnerability Sink
23       ...
24    }
25 }
```

c) Security Patch: Controller/Searchadmin/SearchController.php

Figure 2: Motivation example: CVE-2023-1578, a SQLi vulnerability in Pimcore (3.7k stars). Note that this patch is incomplete and identified by P2PGen during reproduction.

❷ *Understanding fragmented routing semantics.* Even after pinpointing the code-level entry point, reconstructing the corresponding entry URL remains non-trivial because routing rules are rarely centralized. Instead, they are fragmented across various application layers and defined using diverse mechanisms. As discussed in §2.1, routing rules may be implemented in PHP code, annotations, or configuration files, and these sources can be combined to construct the final routing semantics. For example, as illustrated in Figure 2, the routing logic for the entry method `findAction()` is distributed across three locations: one configuration file and two scopes of the annotation layer: (1) *Configuration Layer:* A YAML configuration file defines a global prefix `/admin/search` for the controller resource (Figure 2 (b)). (2) *Class Layer:* The controller class is annotated with `@Route("/search")`, adding a secondary path segment (line 1). (3) *Method Layer:* The entry method itself bears the annotation `@Route("/find")` (line 4). This hybrid routing definition renders traditional rule-based analysis ineffective, as capturing such diverse and scattered semantics is difficult to achieve through modeling.

Existing work failed to construct entry URL. When providing this patch modification (line 10) as the target to the state-of-the-art tool, Predator [60], it coarsely identifies `SearchController.php` as the entry script. Relying on a simplistic direct-path mapping model, it constructs the entry URL *http://localhost/Controller/Searchadmin/SearchController.php*. This deviates significantly from the correct entry URL, *http://localhost/admin/search/search/find*, thereby rendering all subsequent fuzzing efforts futile from the outset.

Challenge II: How to efficiently locate the vulnerability sink to guide the fuzzer towards the target? Even if the correct entry URL is identified, generating a PoC requires guiding the execution flow to the actual vulnerability sink (e.g., line 22). However, this process is obstructed by the semantic gap between the patch and the sink. The details are as follows.

❶ *The patch is rarely the sink.* In many vulnerabilities, the security patch does not directly modify the sink but instead introduces upstream validation logic. For instance, in Figure 2 (c), the patch adds `preg_replace()` to sanitize the `$types` parameter (line 10), whereas the vulnerability sink is `$SearcherList->load()` (line 22). In the original file, the patched code and sink are separated by over 200 lines and involve 30 control-flow constraints. Without a designated vulnerability sink, the fuzzer is forced into blind exploration, wasting substantial resources on paths irrelevant to the vulnerability. Furthermore, this lack of sink guidance necessitates testing every request parameter, resulting in excessive invalid attempts and significantly impeding PoC generation.

❷ *Sinks can be deeply encapsulated.* Identifying the vulnerability sink is further complicated by the extensive encapsulation found in modern applications. As illustrated in Figure 2 (c), the actual SQL execution is triggered by the custom sink wrappers `$SearcherList->load()` (line 22). The common sink modeled by security experts (e.g., `query()`) resides five layers deep within this wrapper (as shown in Figure 5 (a)). Relying on traditional static and model-based approaches [55–57] to identify such sinks necessitates extensive inter-procedural analysis. Moreover, given the multitude of function calls residing downstream of the patched code, the computational overhead of analyzing each one individually becomes prohibitively expensive. Thus, how to efficiently pinpoint such deeply wrapped sinks remains a challenge.

Existing work failed to support sink-targeted fuzzing. These challenges prevent current state-of-the-art fuzzing techniques from generating PoCs for this vulnerability. For instance, even when provided with the correct entry URL and the target patch modification, Predator [60] remained trapped in blind exploration due to sink-unawareness, failing to generate PoCs within the time.

4 Methodology

This section first presents our key solutions for the aforementioned challenges (in §4.1) and then introduces how they operate together to construct P2PGen (in §4.2).

4.1 Key Solutions

Solution for C1: LLM-Powered Generation with Reachability Beacon Feedback. To overcome the semantic gap in fragmented routing mechanisms, we move beyond rigid rule-based analysis. Instead, we leverage the semantic inference capabilities of Large Language Models (LLMs) [43, 44] to reconstruct routing logic from patch-guided contexts. However, a critical challenge in employing LLMs lies in handling their inherent hallucination problem, which can severely hinder the effectiveness of subsequent vulnerability reproduction. We address this with a novel *Reachability Beacon-Feedback* mechanism. By instrumenting a deterministic beacon at the identified code-level entry point, P2PGen utilizes the beacon's presence in the URL request response—rather than ambiguous HTTP status codes—as a precise oracle. This feedback allows the LLM to autonomously self-correct invalid hypotheses, effectively bridging the gap between routing semantics and URLs.

Solution for C2: Forced Path Exploration with VIF-Based Sink Monitor. To efficiently pinpoint sinks downstream of the patched code, we propose *Forced Path Exploration* (FPE). Unlike traditional fuzzing that attempts to satisfy each path constraint during exploration, FPE dynamically modifies branch conditions (e.g., forcing `True` / `False`) to deterministically traverse execution paths. To tackle the inherent path explosion problem, we introduce a *Sequential Unit Analysis* strategy, which linearizes the search space by decomposing logic into sequential execution units rather than exploring path combinatorial states. Furthermore, to detect sinks wrapped within deep encapsulation, we forego high-level PHP hooks in favor of a *Vulnerability Indicator Function (VIF)-Based Sink Monitor* at the interpreter level. This ensures sink identification regardless of encapsulation depth, enabling precise guidance for PoC generation.

4.2 P2PGen Overview

We present an overview of P2PGen, illustrating how its techniques collaborate to achieve patch-guided PoC generation. Figure 3 depicts its architecture. P2PGen accepts security patches, the application source code, and the runtime environment as inputs. By leveraging three primary modules, P2PGen empowers existing directed fuzzing tools (e.g., Predator in our prototype) to effectively generate PoCs for vulnerability reproduction: (1) The *LLM-powered Entry URL Generation* module (§5.1) analyzes the security patch to synthesize entry URLs; (2) P2PGen feeds these URLs into an existing directed fuzzer, targeting patch modification sites to produce URLs that reach the patched code; (3) The *Forced Path Exploration* module (§5.2) uses these URLs to exhaustively explore downstream branch paths, identifying target vulnerability sinks; and (4) P2PGen provides the patch-reached URLs to the directed fuzzer—targeting the identified sinks—and facilitates PoC generation via the *Vulnerability-Oriented Parameter Fuzzing* module (§5.3).

5 P2PGen Details

5.1 LLM-Powered Entry URL Generation

In this part, we describe how P2PGen generates entry URLs for given security patches. As outlined in §4.1, we utilize an LLM-based approach, which we detail in two sub-processes: (1) gathering and providing sufficient context for the LLM (in §5.1.1), and (2) verifying the LLM's output via a self-correction loop (in §5.1.2).

5.1.1 Patch-Guided Entry Context Collection. Applying LLMs to understand routing semantics is non-trivial, as it requires a carefully scoped context. Directly providing the entire application's

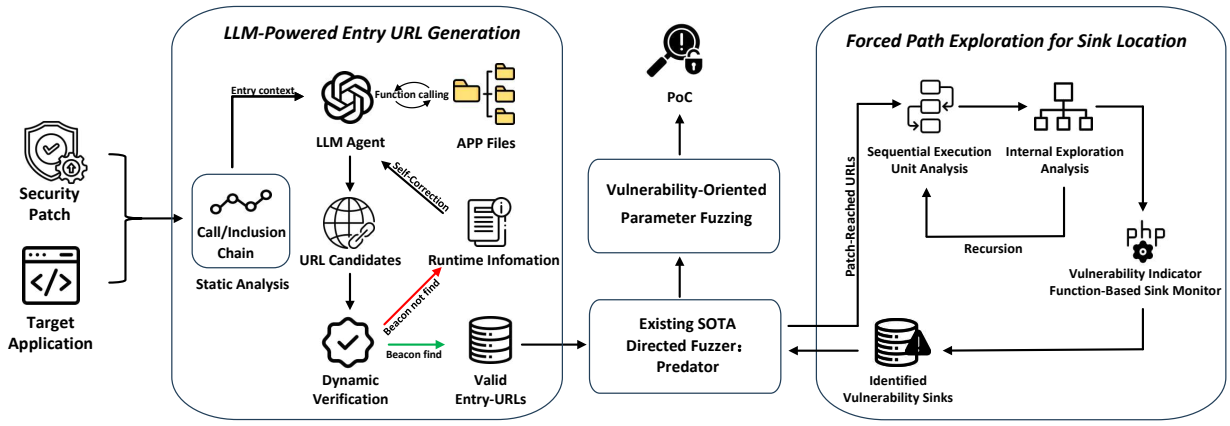

Figure 3: The architecture of P2PGen.

source code may cause the LLM to get "lost-in-the-middle", degrading output quality [45]. Conversely, inputting only the patch file is inadequate, as it often represents an intermediate point in the source-sink flow, and lacks application-wide routing information.

To address this, we first leverage static analysis to trace backward from the patch modification. By performing backward call chain and file inclusion chain analysis, we identify entry points, including action methods or top-level scripts that lead to the patched code. Specifically, our analysis proceeds as follows:

- *The patch modification is in the global scope of a file:* P2PGen employs file inclusion analysis on that file, which recursively traces backward through the file dependency graph, identifying which scripts include the current file. This process continues until a top-level script is found (not included by any other script). This top-level script is then flagged as an entry script.
- *The patch modification is within a function:* P2PGen initiates a backward call chain analysis starting from this function by querying the call graph. Then, two results are possible at each step: (1) If the function's call sites are located within other functions, the call chain analysis continues iteratively. This process repeats until a function is found that has no further call sites. This function is then flagged as an entry method. (2) If the function's call sites are located in the global scope of a file, the call chain analysis halts, and P2PGen invokes file inclusion analysis on that file.

Through this analysis, P2PGen identifies code-level entry points and records the execution chain (i.e., file and function backtracking sequences) leading to the patch. We then construct an LLM prompt combining this specific context with the application's directory tree (via `tree -h`). Empowered by Function Calling [6], the LLM autonomously retrieves and analyzes relevant files to infer routing semantics, ultimately generating high-probability entry URLs.

5.1.2 Beacon-Feedback Self-Correction Strategy. Another primary obstacle in using LLMs is mitigating "hallucinations"—plausible but incorrect outputs. An incorrect URL, even if syntactically valid, will halt the entire fuzzing process before it can begin.

Design Rationale. To address this, establishing a robust "verification then correction" loop is essential. Intuitively, one might rely on standard HTTP status codes (e.g., distinguishing `200 OK` from `404 Not Found`) as the verification oracle. However, this approach is unreliable in modern web applications due to soft 404 behaviors [13] and global exception handling [8], as routing failures are often centrally handled to render user-friendly error pages, frequently returning a `200 OK` status code despite the failure. Consequently, a status-based oracle would produce high false positives, mistakenly marking invalid URLs as correct. Therefore, a deterministic, ground-truth signal is required to precisely confirm that the specific target entry point has been triggered.

Reachability Beacon-Feedback Mechanism. Therefore, we propose a novel reachability beacon-feedback mechanism that serves as a lightweight yet precise verification oracle, operating as follows:

(1) **Instrumentation:** P2PGen first instruments a unique "reachability beacon" at the beginning of each identified entry point. This beacon is programmed to output a unique token string (e.g., in Figure 2 (c), line 6) into the HTTP response upon successful access.
(2) **Verification:** We then empower the LLM with the ability to access URLs via Function Calling [6], so that the LLM will sequentially visit its generated URLs for correctness verification (i.e., by making an HTTP request).
(3) **Feedback:** Next, P2PGen observes the HTTP response from each request. If the unique beacon (the token string) is detected in the response, this proves the URL can reach the specified entry point. The URL is therefore marked as valid and passed to the subsequent analysis steps.
(4) **Correction Loop:** If the beacon is not detected, this failure signal is fed back to the LLM. This feedback prompts the LLM to initiate a reflection and correction loop. Notably, P2PGen places this verification loop under PHP's debug mode (e.g., `error_reporting(E_ALL)`). This allows the LLM to capture the runtime errors or warning messages from the HTTP response. Then, the LLM uses this information to re-evaluate its hypothesis and generate a refined URL, which is then re-verified.

After that, P2PGen provides verified entry URLs to the existing directed fuzzing tool as input and designates the patch modification

as the target, to further evolve the entry URLs into ones capable of reaching the patch modification.

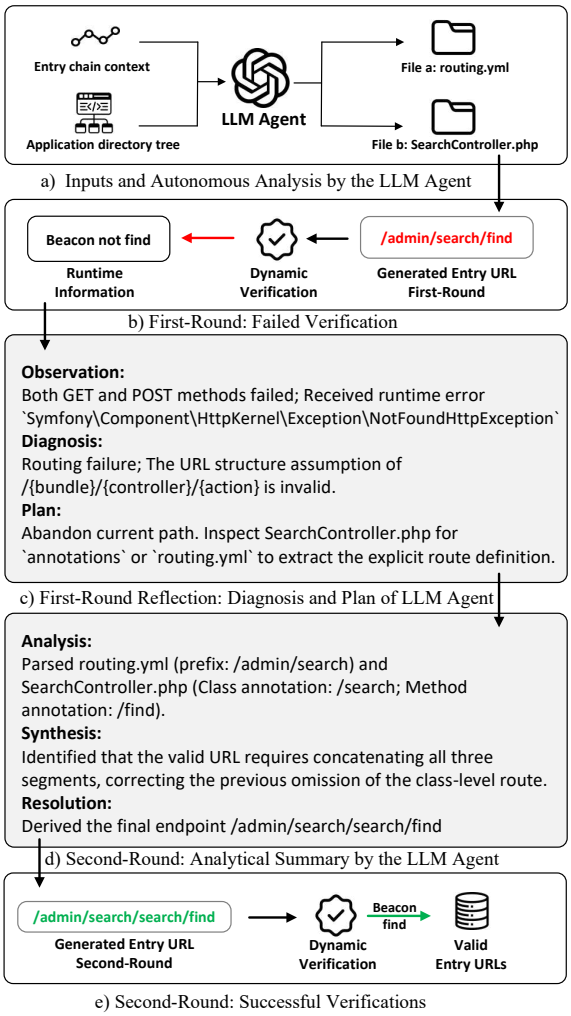

Figure 4: Example of entry URL generation for CVE-2023-1578 using the beacon-feedback self-correction strategy.

Running Example. Figure 4 illustrates the process using CVE-2023-1578. Initialized with the directory tree and static entry context (a), the agent autonomously inspects files to infer routing. It initially constructs an incomplete URL (b), which fails verification (missing beacon). This failure triggers self-reflection (c), allowing the agent to diagnose the issue using runtime feedback. Consequently, it identifies the overlooked class-level `@Route("/search")` annotation (d) and refines the URL. Finally, the beacon is detected in the response (e), confirming the entry URL's validity.

5.2 Forced Path Exploration for Sink Location

In this part, we detail how P2PGen explores code path downstream from patched code to uncover vulnerability sinks. This process encompasses two critical components: (1) monitoring signals triggered by target sinks for identification (§5.2.1), and (2) conducting efficient full-path exploration to prevent omissions (§5.2.2 and §5.2.3).

5.2.1 Runtime Sink Monitor. First, we should establish a monitor to detect signals triggered by target sinks, enabling their efficient identification during the path exploration. However, constructing a robust sink monitor is a non-trivial task.

Design Rationale. Modern PHP applications extensively encapsulate native sink functions. Consequently, monitoring by directly hooking built-in PHP sinks (e.g., `query()`) typically identifies the sink within a generic library layer (e.g., `/PDOConnection.php` in Figure 5 (a)). This misdirects the fuzzer toward library internals rather than the relevant application business logic. To address this, the monitor must backtrack the call stack to identify the upstream encapsulation layer (e.g., `/SearchController.php:load()`). However, relying on `debug_backtrace()` at the PHP layer incurs substantial runtime overhead [15], severely degrading the efficiency.

Vulnerability Indicator Function-Based Monitor. To overcome these obstacles, we shift our monitoring strategy from the PHP layer to the interpreter level. Drawing inspiration from recent work [35], we utilize Vulnerability Indicator Functions (VIFs). VIFs are the low-level C functions within the PHP interpreter that underpin high-level PHP sinks. For instance, the C function `php_output_write()` serves as the underlying mechanism for PHP XSS sinks like `echo` and `print`. By hooking these VIFs directly, P2PGen achieves two key advantages: (1) *Completeness:* It directly captures all sink executions regardless of how many layers of PHP wrappers are used. (2) *Efficient Stack Traversal:* Crucially, operating at the interpreter level allows us to directly access the Zend Engine's execution context via the Executor Globals (e.g., `EG(current_execute_data)`). This allows for lightweight stack traversal without the heavy overhead associated with userland backtracing. As shown in Figure 5 (b), this allows P2PGen to pinpoint the precise upstream sink encapsulation (e.g., `SearchController.php:load()`) for directed fuzzing, while maintaining the high performance required for path exploration.

Request-Sink Parameter Correlation Analysis. Nevertheless, the VIFs-based monitor alone may yield false positives, especially for functions like `echo` or `print` that are used frequently throughout the application. To address this, upon detecting a sink execution, P2PGen identifies the upstream encapsulation function and conducts backward data flow analysis to correlate its parameters with the request parameter, thereby filtering out irrelevant noise. Notably, establishing this request-sink parameter correlation also facilitates the subsequent vulnerability-oriented parameter fuzzing (detailed in §5.3), further enhancing the poc generation efficiency.

5.2.2 Sequential Execution Unit Analysis. Next, we describe how P2PGen performs Forced Path Exploration (FPE) to pinpoint target vulnerability sinks utilizing our VIF-based sink monitor. Overall, it systematically explores downstream control-flow paths from the patch location by dynamically modifying conditional branches. Specifically, instead of relying on specific inputs to satisfy path constraints, P2PGen forcibly redirects control flow (i.e., by injecting `|| True` or `&& False`) at runtime. This mechanism enables deterministic exploration of every potential branch, effectively bypassing the significant overhead of satisfying each complex constraint to uncover target vulnerability sinks.

```
Controller/Searchadmin/SearchController.php:load()
  → models/Search/Backend/Data/Listing/Dao.php:load()
    → lib/Doctrine/DBAL/Connection.php:fetchAll()
      → lib/Db/PimcoreExtensionsTrait.php:executeQuery()
        → lib/Doctrine/DBAL/Connection.php:executeQuery()
          → lib/Doctrine/DBAL/Driver/PDOConnection.php:query()
```

a) PHP Layer: Call Stack of the Vulnerability Sink of CVE-2023-1578

```
static PHP_METHOD(PDO, query){
    // Parse SQL string
    zend_parse_parameters(1, "s", &statement, &statement_len);
    // Prepare SQL (driver-specific)
    dbh->methods->preparer(dbh, statement, statement_len, stmt, NULL);
    ...
    trace_id = get_http_header("HTTP_X_TRACE_ID");
    zend_execute_data *f = EG(current_execute_data);
    char trace[2048], *p = trace;
    for (; f; f = f->prev_execute_data) {
        p += sprintf(p, "%s:%d",
            ZSTR_VAL(f->func->op_array.filename), f->opline->lineno);
    }
    snprintf(path, sizeof(path), "/tmp/%s.cov", trace_id);
    FILE *fp = fopen(path, "a");
    if (fp) { fprintf(fp, "Call Stack:%s", trace); }    // Record Call Stack
    ...
    // Execute SQL Statement
    stmt->methods->executer(stmt);
    ...
}
```

b) C Layer: Vulnerability Indictor Function Monitor for PHP SQLi Sinks

Figure 5: Example of vulnerability indicator function hooking to identify the sink of CVE-2023-1578. The inserted hook code is highlighted with a light yellow background.

The Challenge: Combinatorial Explosion. However, a naive application of FPE would treat all N conditional branches downstream of the patch modification as a single, monolithic state space. This approach necessitates exploring $O(2^N)$ combinations of branch outcomes (`True` / `False`) in the worst case, leading to combinatorial path explosion that renders the analysis computationally infeasible.

Definition of Sequential Execution Units. To mitigate this, P2PGen employs a structural decomposition strategy that linearizes the search space. Specifically, P2PGen parses the downstream execution branches into a sequence of M top-level execution units, denoted as $Unit_1, ..., Unit_M$. Crucially, we define a "top-level execution unit" to encompass two primary types of logic structures that affect execution flow:

- *Conditional Branches:* Explicit control-flow conditional statements (e.g., `if-else`) residing in the main execution scope and not nested within another conditional branch.
- *Function Call Sites:* Invocations of user-defined functions or methods in the main execution scope and not nested within conditional branches. These functions may internally contain conditional branches. If not fully explored, vulnerability sinks embedded within them may be missed.

For instance, the patch-downstream code depicted in Figure 2 (c) comprises four top-level execution units: line 12 and line 22 (function calls), lines 14-18 and line 20 (conditional branches).

Sequential Unit Analysis with State Collapsing. P2PGen analyzes these execution units sequentially. Specifically, for a conditional branch unit (its internal analysis is detailed in §5.2.3), if $Unit_i$ is fully explored without triggering the sink monitor or interrupts (e.g., `die()`), its internal branching becomes *condition-insensitive* downstream. This enables state space collapse: when analyzing $Unit_{i+1}$, we do not need to permute the internal path combinations of $Unit_i$. Instead, we force the execution flow to bypass $Unit_i$ by dynamically modifying its entry condition. P2PGen selects the modification that ensures execution avoids interruption; if neither branch contains an interrupt, it defaults to `&& False`. This ensures that the combinatorial complexity of $Unit_i$ remains isolated from that of $Unit_{i+1}$, thereby linearizing the overall analysis.

For a function call unit, P2PGen adopts a "verify-once" policy to prune redundant states. If a callee is encountered for the first time, P2PGen steps in and recursively applies sequential unit analysis to its internal logic to detect target sinks. If this full exploration yields no sinks, P2PGen marks the callee as "bypass". In all subsequent re-executions of the top-level sequence, P2PGen executes these functions naturally but without re-analyzing their internals. Note that modern PHP applications may rely heavily on massive third-party code (e.g., `vendor/` or `lib/` directories). Forcing exploration paths within these dependencies is computationally expensive and yields minimal utility, as our target vulnerabilities reside in the application's business logic rather than external libraries. Consequently, P2PGen treats third-party callees as black boxes, restricting forced path exploration exclusively to the application code.

Lazy Dependency Recovery Mechanism. While the sequential unit analysis and state collapsing strategy significantly reduce the search complexity, they introduce a challenge regarding execution continuity: data dependency violations.

This issue arises when developers conditionally initialize variables and subsequently access them downstream—a recognized "code smell" [2–5]. Given our decoupled exploration strategy, P2PGen may force upstream units to the default `False` path, leaving downstream variables uninitialized. While PHP treats such variables as `NULL` rather than immediately halting [12], the outcome depends on context. Accessing the variable as a scalar yields only a non-fatal `Warning`, whereas invoking a method on it triggers a `Fatal Error` (i.e., the uninitialized variable is expected to be an object). Unlike warnings, fatal errors terminate execution, causing path exploration to abort prematurely.

To resolve this without reverting to computationally expensive analysis, we implement a Lazy Dependency Recovery mechanism. This mechanism prioritizes control-flow reachability over strict data precision during the path exploration phase, as the primary goal is to discover sinks rather than to verify them. Guided by this principle, our mechanism ignores non-fatal scalar warnings to maintain efficiency but actively traps execution-blocking fatal errors. Specifically, the recovery process operates in three steps: (1) During forward exploration, P2PGen hooks the `ZEND_NEW` opcode at the interpreter level to record all PHP object instantiations. It constructs an *Object Definition Index* that maps each initialization to its enclosing conditional branch unit and captures the condition predicates required to reach it. (2) Upon encountering a fatal error, P2PGen intercepts the runtime exception context. It parses the error message to identify the uninitialized object responsible for the

crash. (3) P2PGen queries the *Object Definition Index* to locate the conditional branch unit containing the missing instantiation. It then repairs the execution path by modifying the predicate sequence associated with the target unit, thereby ensuring the dependency is satisfied in subsequent executions.

5.2.3 Internal Exploration Analysis of Execution Units. In essence, the logic for exploring each top-level execution unit aligns with §5.2.2, merely shifting from the top-level scope (e.g., the scope of `findAction()`, lines 5-24 in Figure 2 (c)) to the local scope within a specific execution unit (e.g., lines 14-18). Therefore, P2PGen recursively applies the sequential analysis strategy to these local scopes. This allows P2PGen to decompose complex internal logic into manageable sub-units, further mitigating the risk of path explosion.

Atomic Conditional Processing. Ultimately, when the recursive decomposition reaches an atomic conditional statement, P2PGen performs the following procedure:

- **Probe:** P2PGen dynamically forces the condition to `True` and then `False`. If the runtime monitor detects the execution of a potential sink during this process, P2PGen reports its location.
- **Prune:** If a forced branch triggers an execution interruption (e.g., `exit`), P2PGen permanently inverts the current branch condition. This ensures the subsequent exploration.
- **Bypass:** If a branch yields neither sinks nor interruptions after full exploration (both `True` and `False`), P2PGen permanently sets the branch condition to `False` by default for path pruning.

5.3 Vulnerability-Oriented Parameter Fuzzing

Once the Forced Path Exploration phase (§5.2) successfully pinpoints the potential sink downstream of the patch modification, P2PGen transitions to the final phase: PoC generation. Specifically, P2PGen feeds the *patch-reached URL* into the existing directed fuzzer and designates the identified sink as the target. However, existing directed fuzzers prioritize code reachability (e.g., path constraints) over exploitability, may fail to confirm vulnerabilities. P2PGen addresses this with two strategies.

Vulnerability Parameter-Aware Fuzzing. A primary inefficiency in current web fuzzers is their "blind mutation" strategy, which applies random mutations uniformly across all request parameters. However, in vulnerability reproduction, data flow to sensitive sinks is typically governed by a small subset of parameters. Consequently, mutating irrelevant parameters wastes fuzzing resources on inputs that cannot influence the sink. P2PGen addresses this by leveraging the Request-Sink Parameter Correlation established by our runtime monitor (§5.2.1). Specifically, we impose a *parameter locking constraint*: upon reaching the sink, the fuzzer freezes all irrelevant parameters and concentrates mutation efforts solely on those influencing the vulnerability. This optimization eliminates the noise of invalid attempts, enhancing PoC generation efficiency.

Pluggable Advanced Technique Integration. Besides, existing directed fuzzers are typically designed to validate static analysis reports [60] involving unprotected code, leaving them with inadequate mutation capabilities for actual exploitation. To bridge this gap, P2PGen integrates advanced, vulnerability-oriented input generation strategies [9, 54].

6 Evaluation

Implementation. We implemented P2PGen atop Predator [60] with 9,089 lines of code. Specifically, the LLM-powered agent is developed in Python, utilizing Gemini 2.5 Pro [7] with a temperature setting of 0 to ensure determinism. Following [46], the self-correction limit defaults to 5 attempts and remains user-configurable. The VIF-based Sink Monitor and FPE module are implemented in C (for instrumentation) and Node.js (for asynchronous code modification), leveraging Predator's native capabilities for backtracking request-sink parameter data. The Vulnerability-Oriented Parameter Fuzzing module is written in Python, incorporating advanced mutation strategies into Predator. For evaluation, we enforce a 30-minute timeout per entry URL and sink.

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

- RQ1: How effective is P2PGen in PoC generation for vulnerability reproduction? (in §6.1)
- RQ2: How efficient is P2PGen in performing the end-to-end analysis? (in §6.2)
- RQ3: How do the different components of P2PGen contribute to its success? (in §6.3)

Dataset. Evaluating P2PGen requires a high-quality dataset of reproducible, in-scope vulnerabilities (i.e., PHP and injection flaws) accompanied by complete runtime environments, security patches, and sufficient scale (exceeding 100 vulnerabilities). As existing datasets [17, 32, 62] fail to meet these specific criteria, we constructed a representative dataset through the following three steps:

1) Known Vulnerability Collection (376 CVEs). We collected CVEs (2015–2025) from NVD [11] and MITRE [1] for popular GitHub PHP applications (>1k stars) with accessible patches. Following [60], we restricted the scope to Cross-Site Scripting (XSS), SQL Injection (SQLi), and Command Injection (CMDi), yielding 376 candidates.

2) Vulnerability-Density-Prioritized Version Selection (209 CVEs). To balance diversity with the high labor cost of environment setup, we prioritized the top 15 application versions exhibiting the highest vulnerability density. This filtering retained 209 CVEs.

3) Exclusion of Ineligible Vulnerabilities (140 CVEs). We further excluded 69 ineligible cases based on manual verification and subsequent evaluation results: inaccessible patches (10), non-PHP modifications (10), over-claimed affected version (21), and out-of-scope types (e.g., second-order injection) (28). The final dataset comprises 140 CVEs, with 74 having public PoCs, detailed in Table 1.

Environment Setup. All the experiments in this section are run on a Ubuntu 22.04 machine with an Intel Xeon Gold 6242 processor and 256 GB memory. Each vulnerable version was deployed as a Docker container running Apache 2.4.29 and PHP 7.4.33.

6.1 RQ1: Effectiveness of P2PGen

In this experiment, we evaluate the effectiveness of P2PGen in generating PoCs for vulnerability reproduction against our dataset.

Result Overview. A PoC is valid only if it triggers the intended vulnerability and manual verification confirms it against a public PoC or the patch scope. Under this criterion, P2PGen generated valid PoCs for 124/140 vulnerabilities (88.57%) across 14/15 applications. Manual verification confirmed that all generated PoCs were valid, demonstrating 100.00% precision. Furthermore, among the 74

Table 1: Breakdown of our evaluation dataset.

Applications	Stars	Version	LoC	CVEs	Avail. PoCs
OpenEMR	4.6k	5.0.1	1,139,760	23	12
LibreNMS	4.5k	22.10.0	195,139	12	11
Microweber	3.4k	1.2.9	214,252	12	10
Pimcore	3.7k	6.9.6	218,392	11	8
phpMyAdmin	7.7k	4.6.0	208,528	10	1
Cacti	1.8k	1.2.22	157,376	10	10
GLPI	5.4k	9.1	165,265	10	1
EcommerceCB	1.4k	7c3c32d	70,909	10	2
Dolibarr	6.7k	8.0.2	719,716	10	1
Piwigo	3.6k	2.9.2	153,004	8	5
osTicket	3.6k	2.9.2	124,674	6	4
MantisBT	1.7k	2.1.0	65,753	5	4
phpIPAM	2.6k	1.3.2	71,709	5	5
PrestaShop	8.9k	1.7.8.1	382,559	4	0
MISP	6.0k	2.4.30	27,231	4	0
Total	-	-	3,914,267	140	74 (52.86%)

vulnerabilities with public PoCs, verification revealed that P2PGen accurately reproduced 71 by generating consistent PoCs, yielding a reproduction success rate of 95.95%. Additionally, we identified 8 incomplete patches and 10 potential new vulnerabilities, 7 of which remain unpatched and exploitable in the latest versions (5 XSS, 2 SQLi).

Result Breakdown. The process breakdown is as follows:

- *Entry URL Generation:* The LLM-powered module first successfully generated valid entry URLs for 129/140 (92.14%) vulnerabilities, effectively handling routing mechanisms to provide a viable starting point for subsequent fuzzing.
- *Patch Reachability:* Then, the directed fuzzer enabled 127/129 (98.45%) entry URLs to successfully reach the patched code locations during dynamic execution.
- *Sink Localization:* After that, for all 127 cases reaching the patch, the Forced Path Exploration module successfully located the sinks, achieving a 100% success rate, highlighting the robustness of our sequential unit analysis and VIF-based sink monitoring.
- *Sink-Targeted PoC Generation:* Finally, equipped with vulnerability-oriented parameter fuzzing, P2PGen successfully generated valid PoCs for 124/127 cases (97.64%).

Detailed Analysis of Failure Cases. We manually analyzed the 16 vulnerabilities where P2PGen failed to generate PoCs. These failures are attributed to two primary causes:

1) Patch-Modified Common Utility Functions (11 vulnerabilities). The majority of failures (11/16) occurred during the entry URL generation phase. We observed that these failures involve patches applied to global utility functions, such as generic sanitizers. Given that these functions are invoked by multiple call sites (averaging 362) across the application, identifying which sites flow to the target vulnerability is quite challenging. We argue that such scenarios are also extremely difficult even for human security experts to analyze.

2) Hard-to-Satisfy Path Constraints (5 vulnerabilities). The remaining 5 failures were due to the inability to satisfy specific path constraints. Specifically, 2 cases failed to reach the patch, while 3 reached the patch but failed to reach the sink. Manual analysis identified two main causes:

- *Server-Side State Dependencies:* The path constraint relies on specific server-side states (e.g., complex session states or global configuration flags). Since these values are not directly mapped to HTTP request parameters, the fuzzer's mutation strategy fails to manipulate them to satisfy the branch conditions.
- *Database Value Dependencies:* The path constraint relies on retrieving specific data from the database. The fuzzer lacks the semantic knowledge to infer the precise input values required to make the query return an expected result, causing the failure.

Verification of Generated PoCs. To verify the reproduction validity of the 124 addressed CVEs (i.e., confirming generated PoCs target intended vulnerabilities), we conducted two validations: (1) For the 71 vulnerabilities with public PoCs, we assessed consistency between the generated and public PoCs. (2) For the 53 vulnerabilities lacking public PoCs, we examined whether the generated PoCs aligned with the scope of the corresponding security patches.

1) PoC Verification against public PoCs (71 vulnerabilities). We manually confirmed that P2PGen's generated PoCs fully covered the public ones, demonstrating 100.00% consistency between the generated PoCs and vulnerability reproduction.

Additionally, we discovered extra PoCs during the reproduction of 16 vulnerabilities. Specifically, for 10 vulnerabilities, we found new PoCs targeting the same sinks as the public ones but utilizing different parameters (indicating potential incomplete patches). For 6 vulnerabilities, we identified 11 new exploitable sinks downstream of the patched code, distinct from the public targets (indicating potential new vulnerabilities). To investigate, we deployed the post-patch versions of these 16 vulnerabilities for verification. The results confirmed 6 incomplete patches and 9 new vulnerabilities.

2) PoC Verification against security patches (53 vulnerabilities). Given the absence of public PoCs, we manually inspected the corresponding security patches to verify that the parameters exploited by the generated PoCs were the intended targets of the fix (i.e., addressed via data flow sanitization or control flow constraints). Our analysis confirmed that all 53 vulnerabilities have generated PoC corresponding to the intended patch fixes, further indicating 100.00% reproduction consistency.

Moreover, for 3 vulnerabilities, we identified extra PoCs where the exploited parameters lacked a clear connection to the patched code. Verification against the post-patch versions confirmed 2 incomplete patches and 1 potential new vulnerability.

Vulnerability Disclosure. Given the security risks posed by the 8 incomplete patches and 10 potential new vulnerabilities, we verified them against the latest versions. We manually confirmed that 7 remain unpatched, comprising 5 XSS and 2 SQLi vulnerabilities. We have responsibly disclosed these active vulnerabilities to the respective vendors. To date, we have been assigned 2 CVE identifiers.

6.2 RQ2: Efficiency of P2PGen

In this experiment, we evaluate the efficiency of P2PGen in generating PoCs for vulnerability reproduction. Overall, P2PGen required a total of 47 hours and 48 minutes, consuming $169.4 USD in LLM tokens to reproduce the vulnerabilities in our dataset. On average, each vulnerability reproduction required 23.13 minutes and cost $1.21 USD. Notably, the FPE module is highly efficient, taking an average of only 7.58 minutes to locate downstream sinks per security patch. Furthermore, the final Time-To-Exploit (TTE) averaged 9.90 minutes per generated PoC.

6.3 RQ3: Ablation Study

In this experiment, we evaluate the contribution of each component of P2PGen. Specifically, we divide the ablation study into two parts: (1) an evaluation of the LLM-powered entry URL generation module (§6.3.1), and (2) an evaluation of each module within P2PGen (§6.3.2). We conduct this experiment on the ground truth dataset (i.e., 74 vulnerabilities with public PoCs).

6.3.1 Component Analysis of LLM-Powered Agent. We first investigate the contribution of our entry-context collection and self-correcting strategies on the effectiveness of entry URL generation. We designed the following variants for comparison and evaluated them against the entry URLs confirmed by 74 public PoCs:

- **Agent$_{Base}$:** The baseline variant that queries the LLM using the raw security patch.
- **Agent$_{Context}$:** A variant augmented with our patch-guided entry context collection strategy.
- **Agent$_{Full}$:** The complete module with both entry context collection and beacon-feedback self-correction mechanism.

Table 2: Ablation study of the Entry URL Generation module against the ground truth (74 URLs confirmed by Public PoCs).

Variants	TP	FP	FN	Prec. (%)	Recall (%)
Agent$_{Base}$	31	43	43	41.89%	41.89%
Agent$_{Context}$	58	14	16	80.56%	78.38%
Agent$_{Full}$	72	0	2	100.00%	97.30%

Overall, the results are presented in Table 2. Our patch-guided entry context collection and beacon-feedback self-correction strategy successfully improved recall in entry URL generation (41.89% → 97.30%, improving 132.28%) and effectively eliminated hallucination-induced false positives (41.89% → 100.00%, improving 138.72%). Specifically, compared to the baseline model (Agent$_{Base}$) directly analyzing patches, providing fine-grained context clues significantly enhances accuracy. Among the 72 vulnerabilities with extracted context, Agent$_{Context}$ generated entry URLs for an additional 27 cases while reducing hallucination-induced false positives (43 → 14). With the introduction of the beacon-feedback self-correction strategy, these hallucinations were further rectified: 10 entry URLs were corrected after the second reflection round, and 4 after the third. Ultimately, Agent$_{Full}$ generated correct entry URLs for 72 vulnerabilities against the ground truth.

6.3.2 Component Analysis of P2PGen. Next, we evaluate the contribution of each module of P2PGen with the following variants:

- **Predator:** The original state-of-the-art work [60] for PHP web applications, which directly targets the patched code.
- **Predator$_{Entry}$:** Predator augmented with our LLM-Powered Entry URL Generation module.
- **Predator$_{Entry+FPE}$:** This variant further incorporates our Forced Path Exploration (FPE) module.
- **P2PGen:** The complete framework, enabling Vulnerability-Oriented Parameter Fuzzing module.

Table 3 presents the detailed results. Compared to Predator (14 reproduced vulnerabilities), P2PGen achieved a 407.14% improvement,

Table 3: Ablation study of P2PGen against the ground truth.

Variants	Reproduced Vulns	Recall (%)	Imprv.(%)
Predator	14 / 74	18.92%	-
Predator$_{Entry}$	43 / 74	58.11%	+207.14%
Predator$_{Entry+FPE}$	66 / 74	89.19%	+53.49%
P2PGen	71 / 74	95.95%	+7.58%

successfully reproducing 71 vulnerabilities. This can be attributed to the contributions of the following three key modules:

Entry URL Generation. Predator's simplistic direct-path mapping found only 21 valid entries, limiting it to 14 reproductions. In contrast, P2PGen generated 72 valid URLs. When provided with these correct URLs, Predator's performance improved by 207.14% (14 → 43 reproductions). This result highlights that accurately resolving routing mechanisms is a prerequisite for successful reproduction.

Forced Path Exploration. Regarding the 29 failures where Predator failed despite possessing entry URLs, our analysis identified two primary causes: 25 cases failed to reach sinks primarily due to blind exploration, and 4 cases failed despite reaching and testing sinks, as the preliminary blind exploration consumed excessive time, preventing successful reproduction within the time limit. P2PGen resolved these issues by deterministically locating target sinks. When provided with these sinks, Predator successfully reproduced 66 vulnerabilities (43 → 66), achieving a 53.49% improvement.

Vulnerability-Oriented Fuzzing. Finally, for the 6 remaining failures (with both entry URLs and target sinks), we observed that Predator often wasted resources on irrelevant parameters or failed to bypass flawed sanitizers (in 5 cases). P2PGen's vulnerability-oriented strategies resolved these bottlenecks, securing the final successes and achieving a total of 71 reproductions (66 → 71, +7.58%).

6.4 Case Study

To further illustrate P2PGen's additional discoveries during the vulnerability reproduction process, we present two case studies[1]: an incomplete patch and a new sink-caused vulnerability.

6.4.1 Incomplete Patch of CVE-2023-1578 (Pimcore, 3.7k stars). During the reproduction of CVE-2023-1578, P2PGen explored downstream of the patched code (line 3) via FPE and identified a sink at line 16. Leveraging the request-sink parameter correlation analysis within our sink monitor, P2PGen pinpointed two sources for this sink parameter: `$types` and `$fields` (lines 3 and 7). Subsequently, P2PGen generated exploitable PoCs for both sources. Interestingly, verification revealed that the public PoC targeted only `$types`. Consequently, developers only fix this source variable (line 3), leaving the other source exploitable even in the post-patch version (line 7). More critically, this vulnerability remained exploitable in the latest version; following our active communication with the developers, a follow-up patch was finally released 34 months later (lines 9-10).

[1]For ethical consideration, we selected two discovered cases that have been fixed. Moreover, we simplified the code snippets due to space constraints.

Lessons learned I: Developers should examine all sources of sink parameters when fixing a reported vulnerability. Focusing solely on the source exploited by the reported PoC may result in incomplete patches. During evaluation, 8/8 (100.00%) of the incomplete patches identified by P2PGen were caused by this.

6.4.2 New Sink-Caused Vulnerability from CVE-2020-25088 (EcommerceCB, 1.4k stars). During the reproduction of CVE-2020-25088, P2PGen executed the FPE downstream from the patched location (line 3) and successfully identified two potential sinks at lines 8 and 9. In the subsequent PoC generation phase, P2PGen automatically constructed exploitable PoCs for both sinks. Manual verification confirmed that while the sink at line 8 was the target of the public PoC (and fixed in line 3), the sink at line 9 remained exposed and exploitable in the post-patch version. However, both vulnerabilities stemmed from insufficient sanitization of similar sources: `$imageValue` (line 2) and `$imageURL` (line 5). In the initial fix, the developers focused narrowly on the parameter exploited by the reported PoC, overlooking parallel data flows and leaving the application exposed. This new sink-related vulnerability was not fixed until 12 months after the initial patch (line 6).

Lesson Learned II: Developers should scrutinize code for similar patterns when fixing a reported vulnerability. Focusing solely on the specific sink identified in a bug report may result in similar vulnerabilities remaining unpatched. During evaluation, P2PGen identified that 8/10 (80.00%) of the new sink-caused vulnerabilities stemmed from this oversight.

```
1    $allParams = $request->request->all();
2    // Incomplete Patch: Commit 367b744, Released on 2023-5-14
3  + $types = explode(',', preg_replace("/[^a-z,]/i", '' ,$allParams['type'] ?? ''));
4    $conditionParts = [];
5    $searcherList = new Data\Listing();
6    …
7    $fields = $allParams['fields'];
8    // Follow-up Patch: Commit 7d951da, Released on 2026-1-8
9  + if (!Definition::getByKey($fields))
10 +     throw new InvalidArgumentException('Check you object arguments');
11   ...
12   $conditionParts[] .= 'LEFT JOIN object_query_' . $fields;
13   $conditionParts[] .= '( maintype IN (' . implode(',', $types) . ') )';
14   …
15   $condition = implode(' AND ', $conditionParts);
16   $hits = $searcherList->load($condition); // Sink
```

Figure 6: An incomplete patch (also our motivation example) of a SQLi vulnerability for CVE-2023-1578 in Pimcore.

7 Discussion

Soundness of Forced Path Exploration. A potential concern is that FPE might identify sinks within dead code by forcibly modifying branch conditions. However, we encountered no such cases during evaluation. We attribute this to the fact that dead code in web applications primarily consists of "software bloat" (i.e., unused files and functions) rather than logically unsatisfiable path constraints (e.g., developer mistakes) [19, 34]. Unlike static analysis, P2PGen inherently avoids this structural dead code by dynamic execution.

```
1    // First Patch: Commit 7c3c32d, Released on 2020-8-3
2  - $imageValue = $_POST['imageValue'];
3  + $imageValue = htmlspecialchars($_POST['imageValue']);
4    // Second Patch: Commit 56465fb, Released on 2021-8-21
5  - $imageURL = $_POST['imageURL'];
6  + $imageURL = htmlspecialchars($_POST['imageURL']);
7    ...
8    echo '<input name="image" value="' . $imageValue . '">'; // Sink
9    echo '<img src="' . base_url('/images/' . $imageURL) . '">'; // New Sink
```

Figure 7: A new sink-caused XSS vulnerability discovered in the reproduction of CVE-2020-25088 in EcommerceCB.

Shared Limitations and Extensibility. As P2PGen is implemented atop Predator, it inherits certain boundaries regarding vulnerability scope and constraint solving. Currently, same as Predator, we prioritize XSS, SQLi, and CMDi; however, the framework can be readily extended to cover other injection-based vulnerabilities like SSRF. Furthermore, while P2PGen shares the inherent challenge of satisfying complex state-dependent constraints common to web fuzzing, its modular design allows it to seamlessly integrate emerging advanced techniques to further enhance its effectiveness.

Extension to Patch Evasion Detection. While our evaluation successfully identifies incomplete patches and new vulnerabilities involving alternative sources or new sinks, we acknowledge the potential for patch evasion where the patched source remains exploitable. Detecting such cases requires testing on post-patch versions; however, manually configuring environments for each vulnerability entails prohibitive engineering effort. We anticipate that as automated environment construction technologies [22, 33, 67] mature, P2PGen can be readily adapted to facilitate this task.

8 Related Work

Web Vulnerability Detection. Program analysis is the mainstream approach for identifying vulnerabilities in PHP web applications. Static analysis [20, 23–25, 38, 47, 55, 63, 65] examines source code based on predefined source-sink models and heuristics. However, this line of work reports high false positives stemming from the absence of runtime verification. In contrast, dynamic analysis [28, 30, 31, 40, 50, 51, 58, 59, 61] provides superior precision by executing the application and observing its behavior. Unfortunately, these approaches often suffer from significant limitations in code coverage and energy scheduling strategies, potentially leading to a high rate of false negatives. A promising alternative is to combine the strengths of static and dynamic analysis, a technique known as hybrid analysis [18, 54, 60, 70]. A representative example is Predator [60], which leverages static analysis reports to guide directed fuzzing. However, this line of work still suffers from several limitations when applied to the vulnerability reproduction task.

Security Patch-Based Research. In recent years, security patches have served as a crucial data source for numerous studies [26, 36, 37, 42, 55–57, 64, 68], owing to the rich vulnerability information and features they contain. One line of work [36, 55, 64] leverages patch modifications to identify vulnerable code clones for detecting the propagation of recurring vulnerabilities. Another line of work [42, 57] analyzes patch modifications to understand the root causes of vulnerabilities and assess the range of affected versions.

Other studies [26, 37] utilize patches to conduct patch presence testing. Furthermore, some research [39, 52, 56, 66, 69] focuses on patch semantics to facilitate automated patch backporting, recommendation, and generation, thereby accelerating the vulnerability remediation process. In this paper, we also analyze and leverage vulnerability clues embedded in patches to facilitate automated PoC generation for vulnerability reproduction.

9 Conclusion

In this paper, we propose P2PGen, a novel framework that leverages security patches for automated vulnerability reproduction. By combining an LLM-powered agent with a beacon-feedback mechanism for routing inference and a Forced Path Exploration strategy for sink localization, P2PGen effectively bridges the semantic gap between patches and exploits. Our evaluation on 140 real-world vulnerabilities demonstrates a PoC generation success rate of 88.57%, significantly outperforming the state-of-the-art fuzzer, Predator, by 407.14%. Furthermore, P2PGen proved its practical value by identifying 8 incomplete patches and 10 new vulnerabilities during the reproduction process, 7 of which remain unpatched in the latest versions (5 XSS, 2 SQLi), yielding 2 CVEs to date.

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. CVE: Common Vulnerabilities and Exposures. https://www.cve.org/.
[2] 2026. CWE-457: Use of Uninitialized Variable. https://cwe.mitre.org/data/definitions/457.html.
[3] 2026. CWE-481: Assigning instead of Comparing. https://cwe.mitre.org/data/definitions/481.html.
[4] 2026. Decompose Conditional. https://refactoring.guru/decompose-conditional.
[5] 2026. Disallow Assignment Operators in Conditional Expressions. https://eslint.org/docs/latest/rules/no-cond-assign.
[6] 2026. Function Calling with LLMs. https://www.promptingguide.ai/applications/function_calling.
[7] 2026. Gemini API. https://ai.google.dev/gemini-api/docs.
[8] 2026. Global Exception Handling. https://www.php.net/manual/en/language.exceptions.php.
[9] 2026. Intruder Payloads. https://github.com/1N3/IntruderPayloads.
[10] 2026. MISP - Threat Intelligence Sharing Platform. https://www.misp-project.org/.
[11] 2026. National Vulnerability Database. https://nvd.nist.gov/.
[12] 2026. PHP Manual - Basics. https://www.php.net/manual/en/language.variables.basics.php.
[13] 2026. Soft 404 Errors. https://www.techmagnate.com/blog/soft-404-errors-guide/.
[14] 2026. Usage Statistics of PHP Websites. https://w3techs.com/technologies/details/pl-php.
[15] 2026. Why is debug_backtrace() using so much memory? https://stackoverflow.com/questions/5175969/why-is-debug-backtrace-using-so-much-memory.
[16] 2026. Wikipedia - Clean URLs. https://en.wikipedia.org/wiki/Clean_URL.
[17] Feras Al Kassar, Giulia Clerici, Luca Compagna, Davide Balzarotti, and Fabian Yamaguchi. 2022. Testability Tarpits: the Impact of Code Patterns on the Security Testing of Web Applications.. In *NDSS*.
[18] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. 2018. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In *Proceedings of the 27th USENIX Security Symposium*.
[19] 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.
[20] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P)*.
[21] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2329–2344.
[22] Islem Bouzenia and Michael Pradel. 2025. You name it, I run it: An LLM agent to execute tests of arbitrary projects. *Proceedings of the ACM on Software Engineering* (2025).
[23] Johannes Dahse and Thorsten Holz. 2014. Simulation of Built-in PHP Features for Precise Static Code Analysis.. In *NDSS*, Vol. 14. 23–26.
[24] Johannes Dahse and Thorsten Holz. 2014. Static Detection of Second-order Vulnerabilities in Web Applications. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*. 989–1003.
[25] Johannes Dahse and J Schwenk. 2012. Rips-a static source code analyser for vulnerabilities in PHP scripts (2010).
[26] Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, Junyan Chen, Xinyu Xing, Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang. 2020. BScout: Direct Whole Patch Presence Test for Java Executables. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*. 1147–1164.
[27] Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. 2019. Towards the detection of inconsistencies in public security vulnerability reports. In *28th USENIX security symposium (USENIX Security 19)*. 869–885.
[28] 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.
[29] 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.
[30] Benjamin Eriksson, Amanda Stjerna, Riccardo De Masellis, Philipp Rüemmer, and Andrei Sabelfeld. 2023. Black Ostrich: Web Application Scanning with String Solvers. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 549–563.
[31] 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*.
[32] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29.
[33] Li Hu, Guoqiang Chen, Xiuwei Shang, Shaoyin Cheng, Benlong Wu, LiGangyang LiGangyang, Xu Zhu, Weiming Zhang, and Nenghai Yu. 2025. Compileagent: Automated real-world repo-level compilation with tool-integrated llm-based agent system. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2078–2091.
[34] 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)*.
[35] Rasoul Jahanshahi and Manuel Egele. 2024. Argus: All your PHP Injection-sinks are belong to us.. In *33rd USENIX Security Symposium (USENIX Security 24)*.
[36] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding Unpatched Uode Clones in Entire OS Distributions. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*. 48–62.
[37] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. 2020. PDiff: Semantic-based Patch Presence Testing for Downstream Kernels. In *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[38] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P)*. IEEE.
[39] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th international conference on software engineering (ICSE)*. IEEE, 802–811.
[40] Taekjin Lee, Seongil Wi, Suyoung Lee, and Sooel Son. 2020. FUSE: Finding File Upload Bugs via Penetration Testing.. In *NDSS*.

[41] Zihan Lin, Yuan Zhang, Jiarun Dai, Xinyou Huang, Bocheng Xiang, Guangliang Yang, Letian Yuan, Lei Zhang, Tian Chen, and Min Yang. 2025. Effective directed fuzzing with hierarchical scheduling for web vulnerability detection. In *34th USENIX Security Symposium (USENIX Security 25)*. 8349–8366.
[42] Ahmed E. Hassan Lingfeng Bao, Xin Xia and Xiaohu Yang. 2022. V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities. In *Proceedings of the 44th ACM/IEEE International Conference on Software Engineering*.
[43] Fengyu Liu, Yuan Zhang, Tian Chen, Youkun Shi, Guangliang Yang, Zihan Lin, Min Yang, Junyao He, and Qi Li. 2025. Detecting Taint-Style Vulnerabilities in Microservice-Structured Web Applications. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 972–990.
[44] Hangtian Liu, Lei Zheng, Shuitao Gan, Chao Zhang, Zicong Gao, Hongqi Zhang, Yishun Zeng, Zhiyuan Jiang, and Jiahai Yang. 2025. EAGLEYE: Exposing Hidden Web Interfaces in IoT Devices via Routing Analysis.. In *NDSS*.
[45] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172* (2023).
[46] 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*.
[47] 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.
[48] Changhua Luo, Wei Meng, and Penghui Li. 2023. Selectfuzz: Efficient directed fuzzing with selective path exploration. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2693–2707.
[49] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*.
[50] 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.
[51] 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.
[52] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 international symposium on software testing and analysis*. 24–36.
[53] Moritz Schloegel, Daniel Klischies, Simon Koch, David Klein, Lukas Gerlach, Malte Wessels, Leon Trampert, Martin Johns, Mathy Vanhoef, Michael Schwarz, et al. 2025. Confusing Value with Enumeration: Studying the Use of {CVEs} in Academia. In *34th USENIX Security Symposium (USENIX Security 25)*. 2887–2906.
[54] 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.
[55] Youkun Shi, Yuan Zhang, Tianhao Bai, Lei Zhang, Xin Tan, and Min Yang. 2024. Recurscan: Detecting recurring vulnerabilities in php web applications. In *Proceedings of the ACM Web Conference 2024*. 1746–1755.
[56] Youkun Shi, Yuan Zhang, Tianhan Luo, Xiangyu Mao, Yinzhi Cao, Ziwen Wang, Yudi Zhao, Zongan Huang, and Min Yang. 2022. Backporting security patches of web applications: A prototype design and implementation on injection vulnerability patches. In *31st USENIX Security Symposium (USENIX Security 22)*.
[57] Youkun Shi, Yuan Zhang, Tianhan Luo, Xiangyu Mao, and Min Yang. 2022. Precise (un) affected version analysis for web vulnerabilities. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*.
[58] 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.
[59] 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.
[60] Chenlin Wang, Wei Meng, Changhua Luo, and Penghui Li. 2025. Predator: Directed Web Application Fuzzing for Efficient Vulnerability Validation. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 886–902.
[61] Enze Wang, Jianjun Chen, Wei Xie, Chuhan Wang, Yifei Gao, Zhenhua Wang, Haixin Duan, Yang Liu, and Baosheng Wang. 2024. Where URLs Become Weapons: Automated Discovery of SSRF Vulnerabilities in Web Applications. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 239–257.
[62] Peiran Wang, Xiaogeng Liu, and Chaowei Xiao. 2025. CVE-Bench: Benchmarking LLM-based Software Engineering Agent's Ability to Repair Real-World CVE Vulnerabilities. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. 4207–4224.
[63] Seongil Wi, Sijae Woo, Joyce Jiyoung Whang, and Sooel Son. 2022. HiddenCPG: Large-Scale Vulnerable Clone Detection using Subgraph Isomorphism of Code Property Graphs. In *Proceedings of the ACM Web Conference 2022*.
[64] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, and Wei Zou. 2020. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*. 1165–1182.
[65] Yichen Xie and Alex Aiken. 2006. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the 15th USENIX Security Symposium (USENIX Security)*. 179–192.
[66] Su Yang, Yang Xiao, Zhengzi Xu, Chengyi Sun, Chen Ji, and Yuqing Zhang. 2023. Enhancing oss patch backporting with semantics. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*.
[67] Zhengmin Yu, Yuan Zhang, Ming Wen, Yinan Nie, Wenhui Zhang, and Min Yang. 2025. CXXCrafter: An LLM-Based Agent for Automated C/C++ Open Source Software Building. *Proceedings of the ACM on Software Engineering* 2, FSE (2025).
[68] Hang Zhang and Zhiyun Qian. 2018. Precise and Accurate Patch Presence Test for Binaries. In *Proceedings of the 27th USENIX Security Symposium*.
[69] Xindong Zhang, Chenguang Zhu, Yi Li, Jianmei Guo, Lihua Liu, and Haobo Gu. 2020. Precfix: Large-scale patch recommendation by mining defect-patch pairs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. 41–50.
[70] Jiazhen Zhao, Yuliang Lu, Kailong Zhu, Zehan Chen, and Hui Huang. 2022. Cefuzz: An directed fuzzing framework for php rce vulnerability. *Electronics* 11, 5 (2022), 758.

Ethical Considerations

Vulnerability Reproduction Environment. All evaluated applications are open source and were downloaded and analyzed in isolated local environments. We did not run P2PGen against third-party live deployments or process real user data. Candidate incomplete patches and new vulnerabilities were likewise verified only on local servers.

Vulnerability Disclosure. Because automated PoC generation is dual-use, we manually confirmed every newly identified issue before disclosure and followed the timelines of the relevant CVE Numbering Authorities (CNAs). We promptly contacted affected developers through private email or the reporting channel specified by each project and provided the root cause and exploitation details. Some detected vulnerabilities were still being fixed when the paper was submitted. We therefore omitted identifying and exploit details for those active issues and present case studies only for issues that had already been fixed.

Open Science

The artifacts of P2PGen, including its source code, dataset, and detailed usage instructions, are publicly available on Zenodo at https://zenodo.org/records/20440300.

A Prompt Templates

Figure 8 and Figure 9 present the prompt templates used in §5.1.

URL Generation Prompt

[ROLE]
You are a Route Analysis Expert. Your task is to generate reachable entry URLs for specified entry points by analyzing the related routing semantics via the provided Entry Context.

[TOOL DEFINITIONS]
You have access to the following functions to inspect the code context:
- `read_file(file_path: str)`: Returns the full content of a file at the specified path. Use this function to inspect any files deemed necessary for analyzing and understanding the required routing semantics.

[INPUT CONTEXT]
1. **Entry Point:** <ENTRY_Point> (The top-level script or action method)
2. **Entry Context:**
- **Execution Chain:** <ENTRY_CONTEXT> (The recorded file and function backtracking sequences leading to the entry point)
- **Directory Layout**: <DIRECTORY_TREE>
3. **Reflection:** <REFLECTION_MESSAGE> (Feedback from failed attempts)

[INSTRUCTIONS]
1. **Analyze Context:** Review the Entry Context to understand the routing semantics of the specified entry point.
2. **Autonomous Exploration:** YOU MUST use the `read_file` tool to inspect relevant files (e.g., `routes.php`, `index.php`, `.htaccess`) to comprehensively understand the required routing semantics.
3. **Entry URL Generation:** After understanding the required routing semantics, generate the entry URL and determine the appropriate HTTP Method for the request (GET/POST).
4. **Output Format:** Output the generated URL in a structured JSON format.

[OUTPUT EXAMPLE]

```
{
  "routes": [
    {
      "url": "/admin/search/search/find?query=admin",
      "methods": ["GET"],
    }
  ]
}
```

Figure 8: Prompt template for entry URL generation.

Self-Correction Prompt

[ROLE]
You are a Dynamic Verification Agent. Your goal is to confirm the reachability of the generated entry URL by detecting a unique "Reachability Beacon" in the HTTP response.

[TOOL DEFINITIONS]
You have access to the following function to interact with the target application:
- `request(url: str, method: str, data: dict)`: Executes an HTTP request to the specified URL. Returns the response body.

[INPUT CONTEXT]
1. **Generated Entry URL:** <GENERATED_URL>
2. **Target Beacon Token:** <UNIQUE_BEACON_STRING> (e.g., "reach_e238151d42ca1ccf8659e3deae61b014")

[INSTRUCTIONS]
1. **Active Verification:** YOU MUST use the `request` tool to visit the provided URL.
2. **Beacon Detection:** Inspect the response body returned by the tool.
- **Success:** If the `<UNIQUE_BEACON_STRING>` is present, mark the URL as VALID and terminate.
- **Failure:** If the beacon is missing, collect and analyze the runtime information (e.g., PHP stack traces, warning messages).
3. **Reflection & Correction:** If verification fails, use the runtime feedback to diagnose specific constraints (e.g., observing a `Notice: Undefined index: action` error implies a missing parameter) and generate a refined URL to re-test immediately.

[OUTPUT]
Provide a concise, text-based summary of the error cause and specific instructions for the next generation attempt.

Figure 9: Prompt template for self-correction strategy.