Web Application Vulnerability Repair via Context-Aware Fault Localization and Directed Differential Fuzzing

Chenlin Wang
The Chinese University of Hong Kong
clwang23@cse.cuhk.edu.hk

Wei Meng
The Chinese University of Hong Kong
wei@cse.cuhk.edu.hk

***Abstract*—Web applications handle sensitive data, yet exploitation of their vulnerabilities can cause significant losses due to their widespread use. While vulnerability detection techniques have become increasingly sophisticated, timely remediation remains challenging due to substantial manual effort requirements. Automated vulnerability repair has become increasingly mature, yet research targeting web applications remains limited due to challenges posed by dynamic languages like PHP, which powers 73.1% of websites. Leveraging existing LLM-based repair work is non-trivial as these systems rely on specialized toolchains and readily available test suites that web applications rarely provide.**

We propose SLICEPATCH, a novel automated vulnerability repair framework for PHP web applications. SLICEPATCH leverages PoC-driven dynamic profiling to capture runtime program behavior and retain vulnerability-specific code paths in a context-aware manner. The isolated vulnerable code slices, distilled from the broader application codebase, guide LLMs to generate precise patches while cutting invocation cost. We pioneer directed differential fuzzing that helps validate and refine patch candidates iteratively without requiring test suites. Our extensive evaluation across 96 real-world vulnerabilities and 7 LLMs shows that SLICEPATCH attains a 94.79% repair success rate, significantly outperforming baselines by over 30%.

1. Introduction

Modern Internet infrastructure relies heavily on web applications to deliver essential services, handling vast amounts of sensitive data including personal information, financial records, and business-critical assets. Attacks on web applications can lead to significant financial losses, with recent surveys showing that the most disruptive security breaches cost organizations an average of £2,050 in staff time alone, with medium/large businesses facing even higher costs of £3,230 per incident [63]. The escalating frequency and sophistication of web-targeted attacks underscore the urgent necessity for rapid and effective vulnerability remediation strategies. While advanced vulnerability detection methods [9, 18, 31, 33] have become increasingly sophisticated at identifying security flaws, many vulnerabilities remain unfixed for extended periods due to the substantial manual investigation and remediation effort required from developers.

Automated vulnerability repair (AVR) has emerged as a promising solution to address the growing demand for timely vulnerability mitigation. AVR techniques automatically generate patches for identified software defects, reducing the manual effort required from developers and accelerating the remediation process [14]. Traditional AVR approaches have shown considerable success in addressing common programming errors in compiled languages, leveraging techniques such as genetic programming [14, 15] and template-based patch generation [21]. Recent advances in large language models (LLMs) have further revolutionized the field, enabling more sophisticated patch generation [24, 38].

Despite these advances, well-developed automated repair mechanisms encounter distinctive obstacles when applied to web application ecosystems, which are predominantly written in dynamic languages, with PHP alone commanding a substantial 73.1% market presence across global websites [64]. Existing AVR solutions cannot be applied to web applications without significant modifications due to the fundamental differences between compiled languages and dynamic interpreted languages. For example, rule-based methods can only handle predetermined vulnerability patterns and require extensive manual engineering for new vulnerability types. Dynamic languages like PHP exhibit loose typing, runtime symbol resolution, variable method invocation, and conditional file inclusion. These features make it difficult to define fixed patterns for runtime-determined behaviors, which in turn severely limits generalizability. The high dynamism also hinders LLM-based repair techniques, as they rely primarily on static analysis and cannot effectively handle runtime behaviors alone.

To generate high-quality patches, automated repair systems need to accurately identify vulnerability locations (*i.e.*, fault localization) by analyzing control flows and data flows. It is non-trivial to achieve this in extensive web application codebases. In practice, modern web applications integrate diverse components spanning multiple programming paradigms. Individual vulnerabilities often span multiple modules and demand understanding of distant code relationships, yet they can be triggered only under specific runtime conditions. Context awareness is thus essential for effective fault localization and patch generation, yet it must

be balanced against the risk of overwhelming the LLM with excessive code.

Beyond addressing the vulnerability itself, verifying that patches both fix the security flaws and do not introduce new errors is crucial for automated repair systems, yet this validation remains challenging in web application contexts. Widely used approaches rely on test suites to validate patch correctness [14, 17, 38], but comprehensive suites are not always available. Our survey of the 12 applications commonly studied by prior work (see Table 9) shows that many provide no usable test suites, making traditional test-based validation infeasible. Moreover, relying solely on test suites is insufficient [38]. Recent research [24] proposed multi-LLM voting mechanisms to assess patch quality. However, these evaluation methods can be unreliable as they lack actual patch deployment and runtime validation. This limitation becomes more pronounced for dynamic languages like PHP. There is a critical need for complementary validation methods that can effectively assess patch correctness in web application environments.

This paper presents SLICEPATCH, a novel automated vulnerability repair framework that is highly effective for web applications. The prototype targets PHP because it underpins the majority of web services, accumulates a disproportionate number of disclosed vulnerabilities, and consequently requires timely, effective mitigation. It dynamically profiles the target application by replaying the proof-of-concept (PoC) exploit to capture precise control flow and data flow information, observing exactly which dynamic code constructs are executed at runtime. This PoC-driven approach enables SLICEPATCH to provide large language models with minimal yet comprehensive vulnerability context, containing only the essential code segments relevant to the vulnerability while excluding irrelevant application logic. For patch validation, rather than relying on potentially unavailable test suites, we propose leveraging *directed differential fuzzing* (DDF) to validate patches. This complementary approach helps detect response differences between the original and patched applications, flagging patches that may introduce functional regressions even when they block the original exploit. Additionally, SLICEPATCH employs an iterative patch refinement technique that leverages validation feedback to progressively improve patch quality, striving for both security effectiveness and functional correctness.

To validate the effectiveness of SLICEPATCH, we conduct comprehensive evaluation across 96 real-world vulnerabilities and 7 state-of-the-art LLMs, including GPT-5, Claude 4 Sonnet, and Gemini 2.5 Pro. SLICEPATCH demonstrates superior vulnerability repair performance, achieving up to 94.79% repair success rate, which is over 30% higher than prior baselines. We conduct comprehensive evaluation of our system components, demonstrating the effectiveness of our two key innovations. Our context-aware fault localization approach precisely identifies vulnerability-relevant code that comprises only a small fraction of the original codebase, reducing code volume by 80.80% to 99.92% while significantly improving patch precision and cost-effectiveness. Our evaluation shows that DDF can detect when patched applications that pass PoC validation are still vulnerable to malicious input variants or introduce functional regressions. Without DDF, 35.70% of defective patches would remain unidentified, severely undermining AVR system reliability. These two key techniques enable SLICEPATCH to achieve superior performance and provide insights that advance future automated vulnerability repair research.

Contributions. In summary, this paper makes the following key contributions:

- We design a zero-shot automated vulnerability repair framework tailored for PHP web applications without requiring vulnerability-specific training data.
- We introduce a context-aware fault localization technique that leverages runtime evidence to extract vulnerability-relevant context from large codebases.
- We propose a novel directed differential fuzzing-based patch validation method that effectively detects patch-induced regressions and complements traditional test suite-based methods.
- We conduct comprehensive evaluation across 96 real-world vulnerabilities and 7 state-of-the-art LLMs, demonstrating SLICEPATCH's superior repair effectiveness with up to 94.79% success rate, significantly outperforming prior baseline tools by over 30%.
- We release SLICEPATCH to facilitate future research in automated vulnerability repair.[1]

2. Background

2.1. Automated Vulnerability Repair

Traditional AVR methods can be broadly categorized into template-based, search-based, constraint-based, and learning-driven approaches [17]. Template-based approaches like PAR [12] use predefined fix templates to guide the repair process. As a representative search-based technique, GenProg [35] mutates AST nodes through evolutionary crossover operators. Constraint-based methods extract repair constraints [5, 8] and employ constraint solvers like Z3 [6] to synthesize patches. Learning-based AVR employs neural networks to learn repair patterns from historical fixes. Early methods adopt encoder-decoder architectures like neural machine translation, where models such as SeqTrans [4] and VRepair [3] translate vulnerable code to patched versions using curated bug-fix datasets. However, they suffer from critical limitations, including context blindness (*i.e.*, models process code at statement/function level, missing inter-procedural context information) and syntactic bias (*i.e.*, generated patches compile but fail functional validation due to semantic misalignment). Fine-tuning strategies like adversarial training [10] and reinforcement learning [37] further improve robustness.

Despite significant advancements in AVR techniques, critical gaps persist in the domain of web application security. Most existing AVR tools primarily target compiled

1. https://github.com/cuhk-seclab/SlicePatch

languages like C/C++ and Java, leaving web programming languages significantly underexplored. Existing AVR approaches targeting PHP web applications either address only specific vulnerability categories or rely on rigid, template-based repair strategies. FixMeUp [30] focuses exclusively on extracting and applying access control templates to protect sensitive operations, while SkyPort [29] serves as a patch backporting tool that identifies and migrates injection vulnerability patches from newer versions to legacy codebases. Pfortifier [26] is a framework that automatically generates a minimal set of potential patches for gadget chain vulnerabilities, preserving functionality through heuristic rules, semantic simulation, and lightweight conditional checks. WAP [19, 20] is a static taint-analysis pipeline that tracks user inputs to sensitive sinks, and automatically injects sanitization code to remediate SQLi, XSS, command injection, and similar input-validation flaws. Additional efforts have targeted other web development languages, such as JavaScript-based XSS vulnerability repair [22], but comprehensive AVR solutions for the broader web ecosystem remain limited.

2.2. Program Slicing

Program slicing is a fundamental technique that extracts semantically relevant code segments (slices) based on a specific program point, variable, or criterion of interest while preserving essential data and control flow dependencies. The primary objective of program slicing is to reduce the scope of code analysis by eliminating irrelevant statements, thereby enhancing the efficiency and precision of downstream tasks such as vulnerability detection, debugging, and program comprehension. With the escalating demands for automated software security analysis, researchers have increasingly integrated program slicing with vulnerability discovery methodologies to improve both scalability and accuracy. A typical application involves enhancing static analysis techniques, particularly taint analysis [32], where slicing helps identify data flow paths that propagate potentially malicious inputs (*i.e.*, sources) to security-sensitive operations (*i.e.*, sinks). Recent research explored synergistic combinations of slicing with dynamic vulnerability detection techniques to address the limitations of purely static approaches. For instance, Sfuzz [2] integrates backward slicing to facilitate vulnerability detection in resource-constrained real-time operating systems (RTOS), decomposing complex monolithic systems into manageable submodules suitable for targeted fuzzing campaigns. Similarly, Fuzzslice [23] operationalizes the hypothesis that slices derived from static vulnerability reports that fail to trigger crashes within predetermined time bounds may indicate potential false positives.

3. Problem Statement

In this section, we motivate our work by identifying the limitations of existing approaches in §3.1, and present our research goals and the technical challenges in §3.2.

3.1. Motivation

Limited Web Application AVR. Few automated vulnerability repair tools target web applications [26, 30], and those that exist focus on specific vulnerability types and demonstrate poor generalizability. Supporting new vulnerability types requires extensive manual rule engineering and domain expertise. Consequently, purely rule-based approaches cannot easily adapt to the diverse coding patterns, framework-specific implementations, and business logic variations found across different web applications. This limitation becomes particularly problematic in the rapidly evolving landscape of web security, where new attack vectors and vulnerability patterns emerge regularly.

Limited Applicability of LLM-Based Approaches to Web Applications. While large language models have demonstrated promising capabilities in vulnerability analysis and automated repair, existing research has primarily focused on native binaries and compiled languages [24, 27, 36, 38], with limited applicability to dynamic interpreted languages, which are commonly used in web development. Although some LLM-based APR work has covered dynamic languages such as Python, PHP remains critically underserved. A recent survey [39] reports that only 1% of LLM-based APR studies target PHP, despite PHP commanding 73.1% of server-side market share [64]. Web applications exhibit characteristics that differ substantially from most programs studied in previous APR research. These applications operate in request-dependent contexts where execution behavior varies based on user input, session state, and environmental configurations. The modular architecture of web applications, with multiple entry points through HTTP endpoints, creates a more complex attack surface.

Existing LLM-based repair frameworks [24, 38] designed for C/C++ and similar compiled languages benefit from well-established debugging and analysis tools, such as AddressSanitizer [40], which provide consistent runtime error detection across different codebases. These frameworks can leverage uniform vulnerability detection mechanisms that enable systematic patch validation and testing approaches. However, web applications lack such unified bug detection toolchains. Developers often implement custom input validation and sanitization logic tailored to their specific application requirements and frameworks to detect and mitigate vulnerabilities. This diversity in security implementation approaches makes direct migration of existing LLM-based repair techniques challenging, as these methods cannot rely on consistent patterns or standardized security mechanisms.

3.2. Research Goals and Challenges

Our research goal is to design an automated vulnerability repair framework that moves beyond vulnerability-specific and hard-to-extend static analysis-based solutions, and overcomes the inability of existing LLM-based methods to handle web application vulnerabilities. This work also aims to explore novel techniques for patch generation and

validation beyond existing methods, advancing AVR research a step further.

Challenges. Achieving these research objectives faces several key technical challenges: ❶ Context extraction creates a dilemma. On the one hand, insufficient context (*i.e.*, only the target function) misses inter-procedural dependencies crucial for correct patches. This limitation is acute for web applications where vulnerabilities often span multiple modules. On the other hand, full codebase inclusion overwhelms computational resources and LLM context windows, introducing noise that degrades patch quality because web application codebases are typically large (see Table 9) while only a small fraction of the code is relevant to any specific vulnerability. ❷ Dynamic language features complicate LLM-assisted repair. As the most widely used web application language, PHP exhibits high levels of dynamism and flexibility. Runtime-determined variables, function calls, class instantiation, and include paths create ambiguous control and data flow graphs, leading to over-approximation with phantom edges and reduced precision in vulnerability repair. ❸ Existing patch validation methods are inadequate for web applications. Previous research [38] relies on standardized bug detection toolchains and developer-provided test suites. On the one hand, input sanitization and bug detection logic in web applications is ad-hoc and diverse. On the other hand, test suites are not always available. Therefore, we need a patch pipeline that remains effective when unified bug detection toolchains and test suites are unavailable.

4. Design of SLICEPATCH

4.1. Architecture Overview

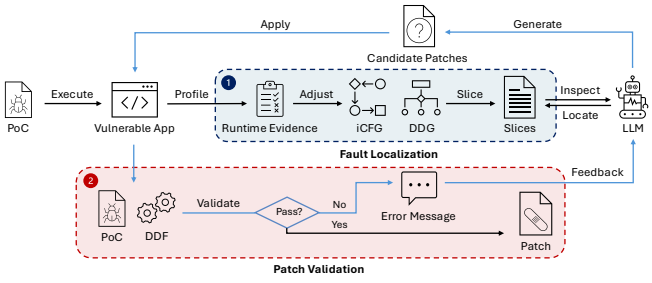


Figure 1: Overall architecture of SLICEPATCH.

As illustrated in Figure 1, SLICEPATCH presents a novel automated vulnerability repair framework tailored for web applications developed in dynamic programming languages like PHP. SLICEPATCH addresses the three key challenges identified in §3.2 through targeted solutions: For the context extraction dilemma (*C1*), we propose *context-aware fault localization* to construct a minimal vulnerability-specific codebase that includes only code relevant to the vulnerability while preserving essential inter-procedural dependencies. For dynamic language feature complications (*C2*), SLICEPATCH leverages runtime evidence observed during profiling to staticize dynamic constructs, transforming ambiguous runtime-determined behaviors into concrete, analyzable code structures. For patch validation challenges (*C3*), we design *directed differential fuzzing* (DDF) to validate patches without relying on comprehensive test suites or standardized toolchains. The framework operates through an iterative *patching loop*, represented by the blue arrows. By continuously using validation feedback to refine generated patches, this loop allows SLICEPATCH to progressively enhance patch quality until a validated patch emerges.

4.2. PoC-Driven Codebase Reconstruction

Although the codebase of web applications is typically large, it often exhibits high modularity. Typically, a PHP web application consists of libraries and different pages, where each page usually corresponds to one or more PHP scripts, and each script typically handles specific business logic. This modularity facilitates the isolation of vulnerability-related code. SLICEPATCH replays the PoC to trace the code actually executed to narrow down the codebase that needs investigation. Meanwhile, it captures runtime evidence for dynamic constructs, thereby improving the precision of subsequent analysis and facilitating efforts to locate the vulnerability. One might argue that static program slicing alone could suffice. However, for dynamic languages such as PHP, pure static analysis must conservatively over-approximate runtime-determined behaviors [1, 18], which yields large slices with high false-positive rates. By replaying the PoC, SLICEPATCH observes concrete runtime values and call paths, enabling precise reconstruction of the vulnerability-relevant codebase.

4.2.1. Irrelevant Code Removal. Given a PoC input triggering the target vulnerability, SLICEPATCH traces the execution path exercised by the PoC across the web application. The profiling process captures all invoked PHP scripts, class definitions, function/method calls, and control flow decisions made while replaying the PoC. We define *profiled codebase* as the codebase that includes only the code that was actually executed during PoC replay. During profiling, files that are not recorded are naturally excluded from the profiled codebase. For files that are captured, SLICEPATCH performs function-level profiling according to the following strategy: 1) Functions that are directly invoked during PoC replay are preserved in their entirety. 2) When methods are called, SLICEPATCH retains all actually executed methods within the containing class, along with magic methods (*e.g.*, `__construct`, `__destruct`) and properties. This selective strategy maximally preserves code that lies on the execution path leading to the vulnerability while excluding unexecuted code segments that would otherwise introduce noise.

4.2.2. Dynamic Construct Staticization. To address the inherent challenges posed by PHP's dynamic nature, SLICEPATCH employs systematic runtime evidence analysis to transform dynamic constructs into statically analyzable equivalents. This staticization process is essential because purely static analysis cannot accurately model PHP state-

ments whose behavior is determined at runtime, such as variable file paths, class names, and method identifiers.

SLICEPATCH targets three primary categories of dynamic features for staticization, as summarized in Table 1.

TABLE 1: Dynamic features addressed by SLICEPATCH's staticization process.

Dynamic Feature	AST Node Type	Runtime Evidence
Dynamic File Inclusion	`AST_INCLUDE`	Concrete file paths
Dynamic Class Instantiation	`AST_NEW`	Concrete class names
Variable Method/Function Calls	`AST_*_CALL`	Concrete callee names

Dynamic File Inclusion Staticization. SLICEPATCH first identifies `AST_INCLUDE` nodes where the file path is specified by a variable rather than a static string literal (*e.g.*, `require($filePath);`). It handles dynamic file inclusion by capturing concrete file paths executed during profiling and replacing variable include statements with static equivalents. As dynamic file inclusion statements may be executed multiple times with different runtime values, SLICEPATCH collects all concrete file paths encountered during execution. Additionally, when magic constants like `__DIR__` are encountered, SLICEPATCH substitutes them with the actual directory paths resolved during execution.

Dynamic Class Instantiation Staticization. Dynamic class instantiation occurs when classes are created using variable class names, which static analysis alone cannot accurately resolve. SLICEPATCH addresses this by parsing profiling results, specifically analyzing `__construct` method calls to determine the actual classes being instantiated. This staticization process targets `AST_NEW` nodes where the class name is specified by a variable rather than a static identifier. For each dynamic instantiation, SLICEPATCH replaces the variable class name with the concrete class name observed during profiling, transforming statements like `new $class();` into `new ConcreteClass();`.

Variable Method and Function Call Staticization. SLICEPATCH handles three types of variable calls: static method calls (`AST_STATIC_CALL`), instance method calls (`AST_METHOD_CALL`), and function calls (`AST_CALL`). For each variable call, the staticization process applies when the method or function name is specified by a variable (*e.g.*, `$class::$method();` → `ConcreteClass::concreteMethod();`). Moreover, indirect calls dispatched through PHP built-in function-handling functions such as `call_user_func()` are transparently resolved during profiling as the execution trace records the concrete callee. This helps correctly capture hook-based dispatch patterns commonly used in large web frameworks (*e.g.*, WordPress and Joomla).

Throughout the staticization process, SLICEPATCH maintains strict provenance tracking by marking all dynamically resolved statements as *artificial* to distinguish them from original code. This process lets subsequent static analyses operate on concrete program representations while preserving traceability for all modifications. Upon completion of this step, SLICEPATCH establishes a codebase consisting solely of code that was actually invoked during PoC replay, with all dynamic features staticized. While this codebase is significantly smaller than the original application, it still requires further processing because it contains code that participates in business logic but is unrelated to the vulnerability's root cause. Such code may introduce noise during vulnerability analysis and patch generation, necessitating precise fault localization to identify the exact vulnerable logic.

4.3. Fault Localization

Fault localization is a critical aspect of vulnerability analysis, aiming to pinpoint the exact vulnerable logic of a defect within the codebase. While LLMs are conventionally regarded as static code analysis tools capable of providing vulnerability repair suggestions, they often struggle with context comprehension when confronted with large codebases. Static analysis techniques can compensate for this deficiency.

Drawing inspiration from the systematic approach employed by human security experts during vulnerability review and repair processes [7], we propose a hybrid fault localization methodology. Human security experts typically begin by analyzing the PoC request's entry point (commonly URLs in web applications) and parameters to roughly identify potentially vulnerable modules, subsequently examining relevant functions and methods within the source code files. They then observe the application's actual execution behavior during PoC replay, tracing the execution path to determine the vulnerability's propagation from source to sink, thereby localizing weaknesses such as inadequate input sanitization or missing validation mechanisms. SLICEPATCH mimics this workflow to automate fault localization and does not rely solely on static analysis due to poor scalability across vulnerability types and high false-positive rates for dynamic PHP. Specifically, SLICEPATCH requires the vulnerability PoC and profiled codebase as input. Each PoC script encodes structured vulnerability metadata, including the vulnerability type (*e.g.*, SQL injection), the HTTP method, the target URL, and the exploit parameters, which collectively guide the LLM during subsequent localization steps. SLICEPATCH initially prompts the LLM to perform preliminary fault localization by analyzing the PoC metadata and content along with a directory structure distilled from the profiled codebase with standard `find`/`sed` tooling. The prompt employed for this initial localization step is shown below, which guides the LLM to identify the most likely vulnerable file.

> **TASK:** Analyze the directory structure and PoC script to identify which file contains the vulnerable logic.
> **INPUT:**
> - Directory structure of the profiled codebase
> - PoC script content and metadata
>
> **OUTPUT:** The file path most likely to contain vulnerable logic

Once the LLM provides an initially suggested file path, SLICEPATCH proceeds to supply the LLM with the file content together with the PoC metadata, requesting it to identify which lines might serve as vulnerability sinks. The vulnerability type specified in the PoC metadata enables

the LLM to disambiguate the correct sink when multiple sink candidates coexist. If the LLM determines that no sinks exist within the current file content, it is allowed to select another file path for examination. This iterative file exploration continues until potential vulnerability sinks are identified. The prompt is structured as follows:

> **TASK:** Analyze file content and PoC script to identify vulnerable lines or select another file for examination.
> **INPUT:**
> - The LLM-suggested file path for examination
> - The complete file content
> - The PoC script content and metadata
>
> **OUTPUT OPTIONS:**
> 1) The identified vulnerable lines, if found
> 2) An alternative file path to examine, if no vulnerabilities are detected in the current file

Note that at this stage, the LLM only possesses rudimentary statement-level understanding of the vulnerability and cannot yet precisely analyze control flow and data flow dependencies within the extensive codebase. Moreover, even code that was actually executed or residing within the same file may not necessarily maintain any dependencies with the vulnerability. Therefore, to leverage the execution context while minimizing interference from irrelevant code, SLICEPATCH conducts further static analysis on the profiled codebase to identify vulnerability-specific code fragments.

4.3.1. Inheritance Chain Resolution. In object-oriented PHP applications, methods are frequently defined in parent classes within the inheritance hierarchy rather than in the immediate class where they are invoked, and inheritance relationships may cause methods to be overridden or invoked across different classes, leading to method resolution ambiguities. Without comprehensive inheritance analysis, the reconstructed codebase would suffer from critical deficiencies including incomplete class definitions and missing method definitions. These deficiencies would render the reconstructed codebase syntactically invalid and semantically incomplete, preventing effective patch generation.

To address this issue, SLICEPATCH recursively traverses both inheritance relationships (`extends`) and interface implementation relationships (`implements`). Each inheritance chain begins from the instantiated class and extends to all inherited classes and implemented interfaces, continuing until reaching built-in classes or interfaces.

4.3.2. Vulnerability-Specific Code Slicing. To further prune vulnerability-irrelevant execution paths, SLICEPATCH examines the profiled codebase to analyze control and data dependencies, leveraging inheritance chain analysis to perform precise program slicing. The streamlined workflow is illustrated in Algorithm 1. The algorithm operates as follows. ① SLICEPATCH constructs the interprocedural control flow graph (iCFG) and data dependence graph (DDG) from the profiled codebase, and initializes the reachable nodes set and final slices collection (lines 1-3). ② SLICEPATCH performs backward traversal from each vulnerability target

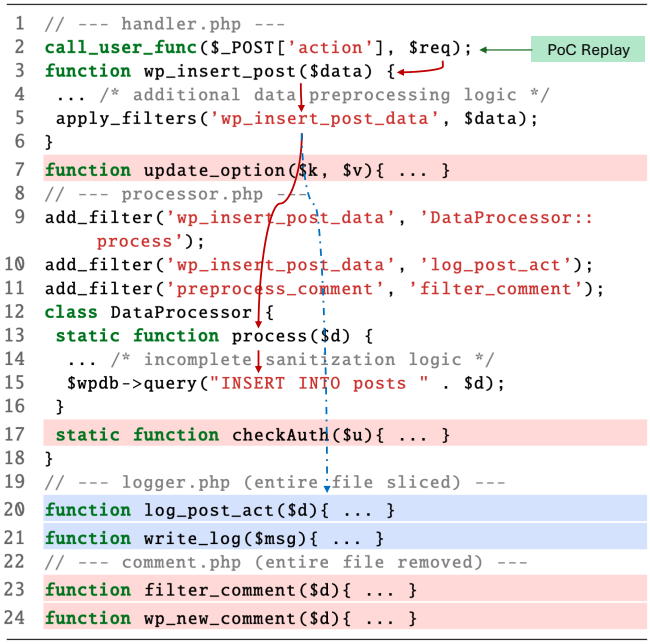
Algorithm 1 Vulnerable Logic Slicing

Require: Profiled codebase $Code$, Inheritance chains $InhChains$, Vulnerability targets $Targets$
Ensure: Vulnerability-specific slices $Slices$
1: $iCFG \leftarrow$ CONSTRUCTICFG($Code$)
2: $DDG \leftarrow$ CONSTRUCTDDG($Code$)
3: $reachables \leftarrow \emptyset$, $Slices \leftarrow \emptyset$
4: **for** each $target \in Targets$ **do**
5: $\quad reachables \leftarrow reachables \cup$ BFS($iCFG^{-1}, target$)
6: $\quad reachables \leftarrow reachables \cup$ BFS($DDG^{-1}, target$)
7: **end for**
8: **for** each $node \in reachables$ **do**
9: $\quad func \leftarrow$ GETFUNC($node$)
10: $\quad$ **if** $func$ is a method **then**
11: $\quad\quad class \leftarrow$ GETCLASS($func$)
12: $\quad\quad inh\_chain \leftarrow InhChains[class]$
13: $\quad\quad$ **for** each $inh\_class \in inh\_chain$ **do**
14: $\quad\quad\quad method\_impl \leftarrow$ GETIMPL($inh\_class, func$)
15: $\quad\quad\quad$ **if** $method\_impl \neq \emptyset$ **then**
16: $\quad\quad\quad\quad Slices \leftarrow Slices \cup \{method\_impl\}$
17: $\quad\quad\quad$ **end if**
18: $\quad\quad$ **end for**
19: $\quad$ **else**
20: $\quad\quad Slices \leftarrow Slices \cup \{func\}$
21: $\quad$ **end if**
22: **end for**
23: $Slices \leftarrow$ REMOVEARTIFICIALS($Slices$)
24: **return** $Slices$

(*i.e.*, vulnerable line nominated by the LLM) using breadth-first search on both the inverted iCFG and DDG to identify all nodes that can reach the targets through control flow or data flow paths (lines 4-7). ③ SLICEPATCH processes each reachable node to extract the corresponding code slice (lines 8-22). For regular functions, the entire function is included. For object methods, the algorithm locates method implementations across the class hierarchy to achieve complete method resolution in object-oriented code where methods may be defined in parent classes rather than the immediate class where they are invoked. Magic methods and properties of the method's class are also included so that all relevant logic is captured. ④ SLICEPATCH removes all artificial statements that are introduced during the staticization process to avoid potential interference in subsequent patch generation (line 23). Without this step, the LLM could incorrectly interpret artificial statements as part of the original code, leading to misguided patch suggestions. ⑤ SLICEPATCH returns the complete set of vulnerability-specific code slices that maintain all necessary control flow and data flow dependencies to the identified vulnerability targets (line 24). To avoid naming conflicts and syntax errors, the generated slices are organized across their original file boundaries. Hence, the class definitions, namespace declarations and include dependencies remain intact by preserving the file structure. The slicing process operates at the function and method level. Plain scripts that contain no declared functions or classes but participate in control-flow or data-flow dependencies to the vulnerability targets are included in their entirety. The resulting vulnerability-specific codebase contains only

the essential logic required to understand and patch the identified vulnerability, while maintaining syntactic validity and semantic completeness.

The design choice to perform slicing on profiled code rather than on the original application code is driven by two primary considerations. First, static analysis methods continue to face computational efficiency bottlenecks, with analysis time increasing significantly as code volume grows. The profiled code has already eliminated unexecuted code paths, reducing analytical complexity and improving efficiency. Second, the profiled code incorporates runtime evidence obtained by profiling during PoC replay to staticize dynamic features, thereby enhancing precision.




```
1  // --- handler.php ---
2  call_user_func($_POST['action'], $req);
3  function wp_insert_post($data) {
4   ... /* additional data preprocessing logic */
5   apply_filters('wp_insert_post_data', $data);
6  }
7  function update_option($k, $v){ ... }
8  // --- processor.php ---
9  add_filter('wp_insert_post_data', 'DataProcessor::
       process');
10 add_filter('wp_insert_post_data', 'log_post_act');
11 add_filter('preprocess_comment', 'filter_comment');
12 class DataProcessor {
13  static function process($d) {
14   ... /* incomplete sanitization logic */
15   $wpdb->query("INSERT INTO posts " . $d);
16  }
17  static function checkAuth($u){ ... }
18 }
19 // --- logger.php (entire file sliced) ---
20 function log_post_act($d){ ... }
21 function write_log($msg){ ... }
22 // --- comment.php (entire file removed) ---
23 function filter_comment($d){ ... }
24 function wp_new_comment($d){ ... }
```

Figure 2: Simplified WordPress-style running example. Red solid paths are preserved after profiling and slicing; blue dashed paths are sibling callbacks not on the backward slice and thus removed.

Running Example. Figure 2 illustrates the full pipeline on a simplified WordPress-style application with a SQL injection vulnerability. `call_user_func()` (line 2) is staticized to its concrete callee `wp_insert_post()`. `apply_filters()` (line 5) is expanded into direct calls to its callbacks `DataProcessor::process()` and `log_post_act()` registered by `add_filter()` at lines 9-10. The vulnerability resides in `DataProcessor::process()` (lines 13-16), which applies insufficient protection against SQL injection. Red-shaded functions were never invoked during the PoC replay and are removed by profiling, *e.g.*, `update_option()` (line 7) and `checkAuth()` (line 17). In practice, only a small fraction of functions are exercised; the rest are removed even within retained files, and entirely unexecuted files such as `comment.php` (lines 22-24) are removed wholesale. Blue-shaded functions are removed by backward slicing: although `log_post_act()` (line 20) is also registered for the same `wp_insert_post_data` hook and was executed during profiling, it does not lie on the backward slice of `process()`, so it and its enclosing file `logger.php` are removed entirely. After profiling and slicing, only the vulnerable path through `apply_filters()` (line 5) to `process()` (line 13) remains.

4.4. Iterative Patching Loop

SLICEPATCH leverages the code slices to generate patches and validate both vulnerability remediation and functional integrity through iterative feedback-driven optimization.

4.4.1. Zero-Shot Patch Generation and Application. SLICEPATCH employs LLM-based methods for patch generation due to their potential to generate high-quality patches compared to traditional solutions. Crucially, SLICEPATCH operates in a zero-shot manner, requiring no prior training on vulnerability-specific datasets or historical patch examples, which enhances the framework's generalizability.

While LLM-based AVR frameworks typically prompt LLMs to directly generate and apply patches in a multi-hunk format [11], LLMs often struggle with numerical tasks and frequently generate incorrect numbers or line positions in code differentials [38]. Direct application of LLM-generated differentials as patches may lead to incorrect repairs due to inaccurate line numbering or context misalignment. Consequently, LLM-generated patches may not be directly applied to the original codebase. Moreover, the code slices provided to the LLM are incomplete representations of the original application codebase, having removed many components. For instance, classes retain only vulnerability-related methods that were actually executed, creating substantial structural differences that make generated differentials inapplicable directly to the original application. We instead implement a two-stage patch application process. First, the LLM identifies the exact vulnerable code segments within the sliced codebase and generates secure replacements. Second, SLICEPATCH locates these code segments within the original application files through AST structure analysis and semantic content matching, then applies the corresponding replacements.

The zero-shot patch generation process operates through a structured prompt that includes four critical components: vulnerability context, the original PoC input, the vulnerability-specific code slices, and validation feedback from previous attempts, if any. The prompt is structured as follows:

TASK: Generate a secure patch for the identified vulnerability by providing exact code replacements.
INPUT:
- Vulnerability type and description
- PoC script that triggers the vulnerability
- Code slices containing vulnerable logic
- Feedback from previous patch attempts (if applicable)

OUTPUT:
1) Original vulnerable code snippet and corresponding secure replacement code
2) Brief explanation of the security improvement

As automated patch application is non-trivial [13], we design two distinct heuristic replacement strategies in a best-effort manner.

Semantic-Aware Replacement. The replacement strategy operates through AST analysis to preserve structural integrity. When both the slice and the target are plain scripts with no declared functions or classes, SLICEPATCH substitutes the entire file with the patched script. Otherwise, SLICEPATCH builds a signature-based mapping over functions, methods, and classes to pinpoint corresponding definitions while leaving untouched code intact. Before any equality checks, all annotations are stripped from the AST so that logically identical implementations with different comments are still recognized as matches. If a counterpart exists but the comment-free bodies diverge, SLICEPATCH deep-copies the patched definition into place; if no counterpart is found, the new definition is appended to the file. Within class scopes, member mappings are further organized by category (methods, properties, constants, *etc.*) to enable fine-grained replacements without disturbing unrelated members. In addition, SLICEPATCH re-injects the original comments into the patched code wherever possible so that developer documentation survives the patching process.

Statement-Level Replacement. When semantic-aware replacement cannot handle fragmented code modifications, SLICEPATCH falls back to string-based matching for individual statements or code fragments. This fallback strategy uses Levenshtein distance to align the original snippet specified by the LLM with its best match in the codebase. After the match is anchored, SLICEPATCH replaces that region with the generated patch. As LLMs frequently generate imprecise code replacements due to their inherent limitations in understanding numbers and exact code contexts, SLICEPATCH implements an adaptive sliding window algorithm that refines match boundaries bidirectionally. Starting with an initial match at line i, the window expands to lines $[i - k_b, i + k_f]$ where optimal backward (k_b) and forward (k_f) extensions are determined by:

$$(k_b^*, k_f^*) = \arg \max_{k_b, k_f \geq 0} S(L_{i-k_b:i+k_f}) \tag{1}$$

subject to $S(L_{i-k_b:i+k_f}) > S(L_{i-(k_b+1):i+k_f})$ and $S(L_{i-k_b:i+k_f}) > S(L_{i-k_b:i+(k_f+1)})$, where $L_{i-k_b:i+k_f}$ represents the code segment and $S(\cdot)$ computes the Levenshtein similarity score. Expansion stops when additional lines would decrease similarity.

4.4.2. Patch Validation and Iterative Refinement.

Applied candidate patches undergo two-tier validation to assess functional correctness and security compliance.

PoC Replay Testing. The preliminary validation replays the original PoC against the patched application to verify that the vulnerability has been effectively mitigated. Before applying any candidate patch, SLICEPATCH first replays the PoC on the original application to confirm the exploit reliably triggers. After applying the patch, SLICEPATCH replays the same PoC and accepts the candidate only if the attack with the original payload is blocked.

Directed Differential Fuzzing. Even after a candidate patch survives PoC replay, residual weaknesses may remain, so additional validation is required. Prior work commonly relies on developer-provided test suites to check that patches do not introduce regressions. This approach breaks down when applications do not provide test suites. Moreover, it cannot detect incomplete fixes that block the original PoC but remain vulnerable to slight variations of the exploit.

We observe that human experts tend to concentrate on the modified code regions and manually design variant inputs that probe boundary conditions and alternative bypass paths. Inspired by this workflow, SLICEPATCH employs directed differential fuzzing to conduct targeted testing of the identified vulnerability and monitor behavioral differences between the original vulnerable application and the patched version. SLICEPATCH sets up two identical instances of the web application: one running the original vulnerable code and the other running the patched code. The fuzzing campaign generates variants of the request parameters and sends them to both applications. SLICEPATCH simultaneously compares their execution behaviors and response patterns.

A successful patch must satisfy three critical criteria: 1) the vulnerability exploitation should be blocked during fuzzing, preventing any attack variants from succeeding on the patched application; 2) the PHP interpreter must not report any syntax errors or fatal execution errors that could indicate improper code modification; and 3) the responses from the original and patched applications must be consistent when processing benign inputs, indicating that normal functionality remains intact. Specifically, the HTTP status code and body content returned by both applications must match after stripping dynamic noise (*e.g.*, timestamps and random tokens). We note that although PHP natively reports syntax errors, SLICEPATCH detects them through DDF, as it compares pre- and post-patch behavior to determine whether errors are introduced by the patch or are pre-existing. To avoid misclassifying transient request or response glitches due to environmental factors, SLICEPATCH retries any inconsistent request again and treats it as a functional regression only if the discrepancy persists. When test cases reveal continued vulnerability indicators, PHP interpreter errors, or behavioral inconsistencies, SLICEPATCH triggers patch refinement cycles to fix the identified issues.

Feedback-Driven Patch Refinement. SLICEPATCH provides structured feedback through three distinct failure categories to guide precise patch refinement:

Incomplete Fixes

Type: Security Failures
Content: Vulnerability reports indicating continued exploitability
Purpose: Guide refinement for insufficient patches

Syntax Errors

Type: Parse/Runtime Failures
Content: PHP interpreter error logs
Purpose: Identify malformed code modifications

Response Inconsistencies

Type: Functional Regressions
Content: The original and patched application responses
Purpose: Detect functionality regressions or side effects

Upon receiving validation feedback, SLICEPATCH leverages this structured diagnostic information to guide the LLM through targeted patch refinement using the structured prompt described in §4.4.1. This feedback-driven approach helps each refinement iteration build upon previous attempts, progressively addressing specific failure modes rather than generating entirely new patches from scratch. As a practical safeguard, if consecutive patch refinement attempts fail, SLICEPATCH re-invokes the fault localization stage to re-identify the vulnerable code segment. The rationale is that if the LLM initially localizes to the wrong code segment, all subsequent patches derived from the incorrect slice would likely fail regardless of refinement effort. Re-running fault localization allows the system to correct course and locate the right vulnerability target before resuming patch generation.

5. Implementation

We implement SLICEPATCH, comprising approximately 5,200 lines of Python code and 1,100 lines of PHP code. We detail key implementation aspects below.

Fault Localization. We employ Xdebug [68], a comprehensive PHP debugging and profiling extension, to capture detailed execution traces during PoC replay. It records function execution sequences and code coverage information, collecting all executed code segments at function granularity into the profiled codebase. SLICEPATCH leverages PHPJoern [1] to construct interprocedural control flow graphs and data dependence graphs. SLICEPATCH integrates php-parser [60] to perform fine-grained AST manipulation while maintaining syntactic correctness and semantic integrity.

Directed Differential Fuzzing. Our DDF implementation extends Predator [33], a directed fuzzing tool for detecting taint-style vulnerabilities in web applications. We parse each PoC to identify the exercised request URL and parameter seeds. Test cases are extracted from the PoC and the modified code regions by analyzing the AST to identify user-controllable superglobals and right-hand values from statements like assignment, comparison, and loop. We inject them into a dictionary-based mutation pool [31] to enable effective fuzzing. We modify Predator to simultaneously dispatch identical test inputs to both the original vulnerable application and the patched version, enabling real-time behavioral comparison to detect incomplete patches and functional regressions. We monitor Apache logs to detect syntax errors and runtime failures, and adapt Witcher's fault escalator [31] to support PoC replay validation within the patch validation pipeline.

Bug Oracles. The current prototype targets three of the most common and impactful web vulnerabilities: SQL injection, cross-site scripting, and command injection. In principle, SLICEPATCH's workflow can support any vulnerability that a PoC can trigger, provided the PoC remains replayable before and after patching. However, Predator only offers directed detection capability for these three vulnerability types, so the implementation presently covers the same scope. *We emphasize that the main contribution of this work lies in proposing a general automated repair framework for web applications and validating two novel AVR techniques. Supporting additional vulnerability types falls outside our current research scope, which we leave for future work.*

6. Evaluation

We evaluate SLICEPATCH through comprehensive assessment on real-world web applications. We aim to answer the following research questions:

- **RQ1:** How effective is SLICEPATCH in repairing real-world vulnerabilities?
- **RQ2:** How efficient is SLICEPATCH in patching vulnerabilities?
- **RQ3:** How does each component contribute to SLICEPATCH's overall performance?
- **RQ4:** How well does SLICEPATCH generalize to unseen vulnerabilities beyond LLMs' knowledge cut-off dates?

6.1. Experimental Setup

Hardware Configuration. All experiments are conducted in Docker containers running on a host machine equipped with 4x Intel Xeon Platinum 8160 processors, 512GB RAM, and Debian 12 operating system.

Dataset. SLICEPATCH's PoC-driven design requires functional PoC exploits for each target vulnerability. We collect PoCs from publicly available vulnerability databases [43, 54], security advisories [46, 47, 50, 52, 57], and research repositories [41]. Our evaluation dataset comprises 12 real-world PHP web applications covering 96 vulnerabilities across diverse domains and complexity levels. The selected applications range from large-scale enterprise systems to lightweight management systems, ensuring comprehensive coverage of web application characteristics. These applications have either been extensively studied in prior research or demonstrated significant adoption in production environments. While some applications may not be popular, we include them because they have been extensively studied in previous research [16, 18, 25, 28, 31, 33, 34], thus they have continued research value. §A.1 summarizes the evaluated applications, including version, codebase size, vulnerability counts, GitHub stars, test suite availability, and references to prior studies (up to 2). The dataset encompasses 67 SQL Injection vulnerabilities (SQLi), 22 Cross-Site Scripting vulnerabilities (XSS), and 7 Command Injection vulnerabilities (CMDi). Detailed CVE information is provided in §A.2.

Large Language Models. We evaluate SLICEPATCH using multiple LLM models from Anthropic, OpenAI, Google, and DeepSeek to assess its generality and robustness and provide insights into model performance variations. We use the following concrete model releases:

- Anthropic: *claude-sonnet-4-20250514*, *claude-3-7-sonnet-20250219*
- OpenAI: *gpt-4o-2024-05-13*, *gpt-5-2025-08-07*
- Google: *gemini-2.5-pro-06-17*, *gemini-2.5-flash-06-17*
- DeepSeek: *deepseek-v3*

Baselines. We compare SLICEPATCH against the strongest baselines for PHP web application vulnerability repair: the traditional static AVR tool WAP [19, 20] and the LLM-based zero-shot repair system proposed by Pearce et al. [27] (ZeroShot for short). As both baselines require the precise location of each vulnerability, we manually prepared and provided the vulnerable files exactly as specified in their documentation for a fair comparison. For ZeroShot, both their paper and Appatch [24] recommend the "Simple 2" template as the best fit for real-world repairs, which removes the vulnerable code and asks the LLM to fill in the blank. We also consider their "Commented Code" template representative because it emphasizes contextual information by commenting out the vulnerable code instead of deleting it. Accordingly, we evaluate both templates as baselines.

Adapting tools designed for other languages or settings would be impractical and unfair as a baseline. For example, one author of PatchAgent [38] confirmed that adapting their system to PHP is non-trivial, while Appatch [24] would require a large PHP-specific vulnerability training dataset and modifications to the code analysis components. We also do not compare SLICEPATCH with Pfortifier [26] and FixMeUp [30] due to vulnerability scope mismatch.

Evaluation Criteria. For each vulnerability, we configure two isolated application instances running the original vulnerable code and the patched version respectively, which enables directed differential fuzzing to detect behavioral differences, if any. The validation process launches 24 fuzzing instances per candidate patch for one hour to verify exploit mitigation while preventing regressions. All successful patches undergo manual review to further confirm correctness.

6.2. RQ1: Effectiveness of SLICEPATCH

Table 2 demonstrates SLICEPATCH's effectiveness in fixing real-world PHP vulnerabilities. SLICEPATCH successfully fixes most vulnerabilities across all three types. Compared with the baselines, SLICEPATCH delivers substantial improvements. The best-performing ZeroShot$_{s2}$ and ZeroShot$_{cn}$ configurations fix 37.50% and 46.88%, respectively, while WAP fixes 35.42%. In contrast, every instantiation of SLICEPATCH with a single LLM exceeds 68% success rate, and running SLICEPATCH with GPT-5 or Claude 3.7 Sonnet fixes 91 of 96 vulnerabilities (94.79%). The SLICEPATCH union fixes 64 SQLi, 22 XSS, and all 7 CMDi vulnerabilities with a 96.88% success rate, while the ZeroShot$_{s2}$ and ZeroShot$_{cn}$ unions plateau at 60.42% and 62.50% respectively. When comparing best-performing individual models head-to-head, SLICEPATCH's top configuration exceeds ZeroShot$_{s2}$ by 57.29% and ZeroShot$_{cn}$ by 47.91%. At the union level, SLICEPATCH still leads, outpacing ZeroShot$_{s2}$ by 36.46% and ZeroShot$_{cn}$ by 34.38%. Even when paired with its weakest LLM configuration (Gemini 2.5 Flash at 68.75%), SLICEPATCH still outperforms ZeroShot unions and WAP.

SLICEPATCH's performance varies across LLM models. When powered by state-of-the-art LLMs such as GPT-5 and Claude 3.7 Sonnet, SLICEPATCH successfully fixes 91 out of 96 vulnerabilities. Anthropic's models achieve >90% success rates, OpenAI's models show variation (94.79% for GPT-5 vs 76.04% for GPT-4o), and Google's models fix 80.21% (Gemini 2.5 Pro) and 68.75% (Gemini 2.5 Flash).

TABLE 2: Effectiveness of SLICEPATCH across LLM models by vulnerability type. ZeroShot$_{s2}$ and ZeroShot$_{cn}$ reuse the "Simple 2" and "Commented Code (alternative), no token" templates from [27]. *Union* reports the distinct vulnerabilities fixed by combining all models within each setting.

Setting	Model	SQLi	XSS	CMDi	Failed	Success%
SLICEPATCH	GPT-5	62	22	7	5	94.79%
	Claude 3.7 S	63	21	7	5	94.79%
	Claude 4 S	61	21	5	9	90.63%
	DeepSeek V3	56	18	7	15	84.38%
	Gemini 2.5 P	53	17	7	19	80.21%
	GPT-4o	51	15	7	23	76.04%
	Gemini 2.5 F	40	20	6	30	68.75%
	Union	**64**	**22**	**7**	**3**	**96.88%**
ZeroShot$_{s2}$	GPT-5	18	11	7	60	37.50%
	Claude 3.7 S	15	10	6	65	32.29%
	Claude 4 S	16	11	6	63	34.38%
	DeepSeek V3	10	11	5	70	27.08%
	Gemini 2.5 P	11	6	3	76	20.83%
	GPT-4o	17	9	6	64	33.33%
	Gemini 2.5 F	12	11	3	70	27.08%
	Union	**35**	**16**	**7**	**38**	**60.42%**
ZeroShot$_{cn}$	GPT-5	23	12	7	54	43.75%
	Claude 3.7 S	16	15	6	59	38.54%
	Claude 4 S	15	16	6	59	38.54%
	DeepSeek V3	12	17	7	60	37.50%
	Gemini 2.5 P	14	7	4	71	26.04%
	GPT-4o	14	11	7	64	33.33%
	Gemini 2.5 F	23	15	7	51	46.88%
	Union	**35**	**18**	**7**	**36**	**62.50%**
WAP	/	21	9	4	62	**35.42%**

Unfixed Cases. Despite the overall high repair effectiveness, several vulnerabilities consistently resist repair attempts across all LLM models. Specifically, *CVE-2020-29142* in OpenEMR and *CVE-2020-22452* in phpMyAdmin persistently fail directed differential fuzzing validation. Our investigation reveals that while LLM-generated patches can be corrected via feedback-driven patch refinement, this approach cannot completely eliminate all edge cases in complex code structures. Additionally, *CVE-2020-5504* in phpMyAdmin fails due to errors in the static analysis phase, where insufficient vulnerability context information is preserved, preventing accurate fault localization and subsequent repair. These cases highlight the challenges in automated vulnerability repair for complex web applications, where intricate dependencies and nuanced code semantics can confound even sophisticated repair mechanisms. Nevertheless, SLICEPATCH shows evidence of generalizing across code-intensive and logically complex applications such as Open-

EMR (9/10), WordPress (1/1), and Joomla (2/2), suggesting it can patch most vulnerabilities while handling diverse application architectures.

False Negatives. We define false negatives as patches that genuinely remove the vulnerability without introducing regressions yet are still flagged as failures. During our evaluation, we found two false negatives in OpenEMR (*CVE-2020-29142*) and phpMyAdmin (*CVE-2020-22452*). In both cases the patches had already fixed the underlying vulnerabilities, but directed differential fuzzing observed response mismatches and marked them as failures. The root cause is that these vulnerabilities sit in execution paths that also contain developer-provided functions adjusting permissions or database settings. For example, a database table modification might succeed on one instance but fail on the other during fuzzing. When a new correct patch is tested later, the validator sees different outputs and raises a false regression alarm.

These observations indicate that patching vulnerabilities in an iterative manner can produce false alerts when execution paths in the vulnerable scripts may also modify permissions or database settings, particularly in applications such as phpMyAdmin. Potential solutions include implementing database backup and restoration mechanisms or configuration state management during the fuzzing process. However, SLICEPATCH does not adopt these measures as they would significantly reduce validation efficiency and complicate the automated repair pipeline. We also do not expect a significant improvement in overall effectiveness since such edge cases are rare in practice and they require specific conditions (*e.g.*, database setting changes during fuzzing) to manifest.

False Positives. We define false positives as patches that remain exploitable yet pass the validation pipeline. We observed a false positive that traced back to a limitation in Predator rather than SLICEPATCH. When repairing *CVE-2023-34626* in Piwigo, a generated patch containing syntax errors was incorrectly classified as successful. Investigation revealed that the vulnerability existed in a script that also contains the authentication logic. After applying the syntactically incorrect patch, the authentication mechanism became non-functional. Consequently, the underlying directed fuzzing tool Predator kept retrying login indefinitely for the entire one-hour fuzzing duration. When the one-hour time budget was exhausted with no vulnerabilities triggered or errors detected, the monitoring system interpreted this as successful validation, leading to incorrect classification. We patched Predator to treat persistent login failures as validation failures and immediately move to the next patch attempt. After deploying this fix, the issue has not recurred any further. Any syntax errors in generated patches are now reliably detected once forwarded to the directed differential fuzzing phase.

Incorrect Patches. To analyze the failure modes of incorrect patches and investigate whether DDF really improves patch quality, Table 3 presents a comprehensive breakdown of all incorrect patches produced by different LLM models during the evaluation. The table categorizes unsuccessful patch attempts into four distinct failure types: *PoC failures* where patches fail to pass the PoC replay, *Incomplete repairs* where patches remain vulnerable to variant malicious inputs during directed differential fuzzing validation, *Syntax errors* in generated code causing runtime failures, and *Functional regressions* indicating functional damage where patched applications respond incorrectly to benign inputs compared to the original application.

TABLE 3: Detailed failure analysis across LLM models showing breakdown of unsuccessful patch attempts by failure type.

Model	PoC	Directed Differential Fuzzing			Total
		Incomplete	Syntax	Regression	
GPT-5	204	25	26	1	256
Claude 3.7 S	378	52	141	30	601
Claude 4 S	393	110	96	4	603
DeepSeek V3	1,541	271	275	646	2,733
Gemini 2.5 P	355	71	56	1	483
GPT-4o	1,206	50	160	337	1,753
Gemini 2.5 F	425	13	108	27	573
Ratio	64.30%	8.45%	12.31%	14.94%	100.00%

PoC replay alone is not sufficient, even though it captures the majority of patch failures (64.30%). Our results demonstrate that incomplete repairs, syntax errors, and functional regressions collectively account for 35.70% of patch failures and are detected during the directed differential fuzzing stage. Without this additional validation step, over one-third of seemingly correct patches would be incorrectly accepted as valid fixes. This finding highlights how directed differential fuzzing addresses a critical gap in web application vulnerability repair.

Failure pattern analysis reveals substantial performance disparities. GPT-4o and DeepSeek V3 generate significantly more incorrect patches than other models, with 1,753 and 2,733 total failures, respectively, and these models account for the majority of functional regressions, indicating their patches frequently introduce unintended behavioral changes. For syntax errors, GPT-4o, Claude 3.7 Sonnet, and DeepSeek V3 exhibit higher rates of generating syntactically invalid code, while other models demonstrate superior code generation stability. For incomplete repairs, Claude 4 Sonnet and DeepSeek V3 show particular susceptibility to generating patches that remain exploitable under variant attack inputs, indicating limited vulnerability comprehension. GPT-5 emerges as the most reliable model with only 256 total failures across all categories, while DeepSeek V3 exhibits the poorest performance with the highest failure rate across all failure modes.

6.3. RQ2: Efficiency of SLICEPATCH

Table 4 presents SLICEPATCH's efficiency across different LLM models, demonstrating reasonable resource requirements per vulnerability with an average cost of $0.716 and 81.05 minutes processing time. We compute averages over all 96 vulnerabilities to reflect the true operating cost. Cases

TABLE 4: Efficiency of SLICEPATCH in terms of average token usage, cost, and time (minutes) per vulnerability across LLM models. Averages are computed over all vulnerabilities, including both successful and failed cases.

Model	Input	Output	Cost	Time
GPT-5	118,636	30,687	$0.455	80.14
Claude 3.7 S	196,253	8,794	$0.721	80.57
Claude 4 S	212,978	10,163	$0.791	75.88
DeepSeek V3	552,366	25,158	$0.138	94.49
Gemini 2.5 P	205,119	97,182	$1.228	77.78
GPT-4o	260,929	12,476	$1.554	86.39
Gemini 2.5 F	73,696	40,147	**$0.122**	**72.13**
Average	**231,425**	**32,087**	**$0.716**	**81.05**

where the input exceeds the LLM's context length limit cannot be processed and consume no tokens, so they are excluded from the averages. Among the LLM models, Gemini 2.5 Flash achieves the lowest cost ($0.122) with the fastest repair time (72.13 minutes); DeepSeek V3 also maintains low cost ($0.138) despite high input token consumption; and Claude 3.7 Sonnet demonstrates superior output efficiency with only 8,794 output tokens while maintaining high effectiveness. GPT-5 provides optimal balance between effectiveness and efficiency, achieving 94.79% success rate at moderate cost ($0.455) and processing time (80.14 minutes). These results confirm that SLICEPATCH delivers effective vulnerability repair without excessive computational or financial overhead, establishing its viability for real-world deployment in security-critical environments.

TABLE 5: Average resource consumption by LLM model across different stages of SLICEPATCH.

Model	Fault Localization		Patch Generation	
	Input / Output	Cost	Input / Output	Cost
GPT-5	3,077 / 1,020	$0.014	26,707 / 6,747	$0.101
Claude 3.7 S	5,711 / 105	$0.019	26,116 / 1,225	$0.097
Claude 4 S	3,669 / 188	$0.014	25,562 / 1,181	$0.094
DeepSeek V3	2,642 / 40	$0.001	17,785 / 826	$0.004
Gemini 2.5 P	3,224 / 1,174	$0.016	21,058 / 9,959	$0.126
GPT-4o	2,164 / 36	$0.012	14,342 / 680	$0.085
Gemini 2.5 F	3,037 / 885	$0.003	17,966 / 9,139	$0.028
Average	**3,361 / 493**	**$0.011**	**21,362 / 4,251**	**$0.076**

Cost Breakdown. Table 5 decomposes SLICEPATCH's resource consumption across two stages. SLICEPATCH demonstrates cost-effective operation across both fault localization and patch generation phases, with average costs of $0.011 and $0.076, respectively. Model-wise analysis shows consistent cost patterns, with most models following similar resource allocation between the two stages. Comparing the per-invocation subtotal ($0.087) with the per-vulnerability total in Table 4 ($0.716) reveals that each vulnerability requires on average roughly 8 rounds of iterative refinement.

6.4. RQ3: Ablation Study

We conduct a comprehensive ablation study using GPT-5, which demonstrated the highest overall performance in our evaluation. We removed key components of SLICEPATCH and measured their impact on repair effectiveness and efficiency. To evaluate the importance of our vulnerability-specific code slicing approach, we designed three variants to assess the effectiveness of our slicing-based context extraction approach: SLICEPATCH$_{Tar}$ operates with minimal context by extracting only the vulnerable function, SLICEPATCH$_{Prof}$ uses the complete profiled codebase without further slicing, and SLICEPATCH$_{Orig}$ attempts repair using the original application codebase. To evaluate the impact of directed differential fuzzing validation, we further included SLICEPATCH$_{NoDDF}$, which disables directed differential fuzzing while keeping all other components intact.

TABLE 6: Ablation study on the effectiveness of different components in SLICEPATCH using GPT-5. SLICEPATCH$_{Tar}$ provides only the vulnerable function; SLICEPATCH$_{Prof}$ provides the entire profiled code without slicing; SLICEPATCH$_{Orig}$ provides the complete original application code; SLICEPATCH$_{NoDDF}$ denotes SLICEPATCH without directed differential fuzzing validation. Token usage, cost, and time (minutes) are averaged per vulnerability, including both successful and failed cases.

Setting	Success%	Input	Output	Cost	Time
SLICEPATCH	**94.79%**	118,636	30,687	$0.455	80.14
SLICEPATCH$_{NoDDF}$	83.33%	42,767	15,734	$0.211	75.12
SLICEPATCH$_{Tar}$	78.13%	197,751	83,888	$1.086	77.54
SLICEPATCH$_{Prof}$	72.92%	337,080	46,958	$0.891	72.90
SLICEPATCH$_{Orig}$	27.08%	489,146	45,699	$1.068	107.03

Table 6 provides compelling evidence for the importance of the two key components. SLICEPATCH achieves 94.79% success rate across all vulnerability types, significantly outperforming all variant configurations. When provided with only the vulnerable target function, success rates decline to 78.13%. Using the entire profiled codebase without further slicing results in further decline to 72.92% success rate, despite having more contextual information, highlighting how excessive irrelevant code can impede repair performance. Most dramatically, relying on the original codebase results in severe performance degradation to just 27.08%, with virtually no success on XSS (1/22) and complete failure on command injection vulnerabilities (0/7). These findings demonstrate that SLICEPATCH's slicing mechanism effectively improves repair performance by focusing on relevant code segments and underscore the importance of context awareness in LLM-based vulnerability repair for complex applications. This optimal context enables more accurate, efficient, and cost-effective repairs compared to both minimalist approaches that provide insufficient context and exhaustive approaches that overwhelm the model with irrelevant code.

Removing directed differential fuzzing validation drops SLICEPATCH$_{NoDDF}$'s repair success rate from 94.79% to 83.33%. The lower cost per vulnerability reflects the lack of DDF-guided iterative refinement, which leaves complex

vulnerabilities unsolved rather than delivering efficiency gains. This observation aligns with the results in Table 3, showing that DDF is an essential validation layer that filters incorrect patches before they are reported as fixes.

Resource pattern analysis across configurations reveals interesting trade-offs. Among the context variants, the SLICEPATCH$_{Tar}$ variant requires the most output tokens (83,888), which also drives its cost up to $1.086 per vulnerability, indicating that the model struggles to efficiently generate effective patches with limited context. The SLICEPATCH$_{Prof}$ variant exhibits high consumption, requiring 337,080 input tokens and costing $0.891 per vulnerability, which is nearly twice the cost of SLICEPATCH while achieving lower success rates. The SLICEPATCH$_{Orig}$ variant demonstrates the highest input token consumption (489,146) per vulnerability, where excessive code noise prevents the model from understanding vulnerable code and fixing the vulnerability.

To provide a transparent breakdown, Table 7 categorizes all outcomes by failure type across ablation settings. SLICEPATCH$_{Orig}$'s failures are dominated by context-length exceeded errors (50 out of 70). Profiling and slicing progressively reduce this issue. SLICEPATCH$_{Prof}$ reduces context failures to 9, while SLICEPATCH eliminates them entirely. Disabling directed differential fuzzing (SLICEPATCH$_{NoDDF}$) primarily increases patch failures from 2 to 13, demonstrating that DDF is essential for filtering incorrect patches. As noted in §6.3, the averages in Table 6 exclude context-length exceeded cases.

TABLE 7: Failure type breakdown. *Context* refers to context-length exceeded failures, *Locate* refers to failures in fault localization, *Patch* refers to failures in patch generation.

Setting	Context	Locate	Patch	Success
SLICEPATCH	0	3	2	91
SLICEPATCH$_{NoDDF}$	0	3	13	80
SLICEPATCH$_{Tar}$	4	3	14	75
SLICEPATCH$_{Prof}$	9	4	13	70
SLICEPATCH$_{Orig}$	50	1	19	26

Code Reduction. To further understand the effectiveness of our fault localization mechanism, we analyze two critical metrics: 1) localization accuracy assessed by determining whether the process successfully identifies vulnerable code segments, and 2) code reduction effectiveness measured by comparing original codebase size against profiled and sliced code volumes. We conducted exhaustive manual verification of each fault localization attempt. Given the substantial effort required to manually review across all models, we focused on GPT-5, which achieved the highest overall performance.

Table 8 demonstrates SLICEPATCH's fault localization accuracy and code reduction effectiveness across different applications. We examined all 114 fault localization attempts, confirming that SLICEPATCH successfully identified vulnerable code segments in 93 cases (81.58%), with remaining cases primarily involving complex code structures in OpenEMR and phpMyAdmin. Our approach consistently achieves high code reduction rates, with large applications (>70K LoC) achieving reductions exceeding 98%. For example, OpenEMR is reduced from 1,839.5K LoC to 16,100 LoC after profiling, and further to 1,486 LoC after slicing (99.92% overall reduction). Smaller applications also demonstrate substantial reductions between 80.80-99.04%. Localization accuracy is notable, with 10 out of 12 applications achieving 100% accuracy. The two exceptions, OpenEMR (64.29%) and phpMyAdmin (5.88%), represent the most complex applications in our dataset with intricate dependency structures. Notably, none of the generated slices exceeded the LLM context windows during our evaluation, further indicating that SLICEPATCH's fault localization strategy keeps prompts practical even for complex applications.

TABLE 8: Fault localization accuracy and code reduction effectiveness across applications. *Succ./All* denotes the number of successful localizations over all attempts, and *Time* denotes minutes from start to successful generation of vulnerability-specific slices.

Application	Succ./All	Success%	Original	Profiled	Sliced	Reduction%	Time
OpenEMR	9/14	64.29%	1,839.5K	16,100	1,486	99.92%	107.26
Joomla	2/2	100.00%	529.6K	9,860	408	99.92%	11.65
WordPress	1/1	100.00%	474.5K	16,271	9,328	98.03%	16.74
phpMyAdmin	1/17	5.88%	325.7K	20,220	1,172	99.64%	9.15
Piwigo	5/5	100.00%	241.8K	5,180	1,398	99.42%	2.74
phpIPAM	16/16	100.00%	143.4K	1,787	355	99.75%	2.12
rConfig	18/18	100.00%	79.5K	634	123	99.85%	0.65
WeBid	11/11	100.00%	53.2K	3,614	2,140	95.98%	1.09
Hospital Mgmt.	10/10	100.00%	14.5K	586	161	98.89%	0.46
Webchess	2/2	100.00%	5.0K	2,248	385	92.30%	0.59
Doctor Appt.	8/8	100.00%	4.6K	180	44	99.04%	0.42
Login Mgmt.	10/10	100.00%	1.0K	359	192	80.80%	0.30

Efficiency varies by application size, with most applications completing within 3 minutes. Larger applications like OpenEMR, WordPress, and Joomla require more time (107.26, 16.74, and 11.65 minutes respectively) due to their size and complexity, but remain practical for real-world deployment.

6.5. RQ4: Unseen Vulnerability Repair

To verify that SLICEPATCH's repair capability is unaffected by potential LLM training data leakage, we measure its ability to fix *unseen vulnerabilities*. We define unseen vulnerabilities as those disclosed after a model's knowledge cut-off or released within the past five years but still lacking any patch references. For these vulnerabilities, the models either have no prior knowledge of their existence or cannot learn any concrete fixes from existing patches.

Table 12 reports how each model performs on these unseen vulnerabilities. The evaluation covers 28 unseen vulnerabilities drawn from six applications and repair outcomes from seven models. The precise knowledge cut-off for each LLM is summarized in §B.1. As some vulnerabilities in phpIPAM were fixed before the knowledge cut-offs for certain models, we exclude them from those models' unseen evaluation scope. SLICEPATCH fixes every unseen vulnerability when paired with GPT-5, Claude 3.7 Sonnet, or

Claude 4 Sonnet, and it repairs over 80% with the remaining models, underscoring SLICEPATCH's strong generalization capability to unseen vulnerabilities. Detailed analysis can be found in §B.2.

7. Discussion

Why SlicePatch Works. SLICEPATCH uses context-aware fault localization, which pinpoints executed, vulnerability-related code more accurately compared to pure static slicing. Runtime traces from real exploits give the LLM focused context, shrink the code volume, reduce compute cost, and expose dynamic constructs in concrete forms amenable to static analysis. As PatchAgent [38] suggests, patch validation remains an open challenge in automated vulnerability repair as functional tests alone are insufficient, and correctness spans more than security. We identify directed differential fuzzing as a promising approach that complements existing validation techniques, effectively detecting incomplete patches and functional regressions. Moreover, validating patches through actual execution is more reliable than approaches such as multi-LLM voting, as runtime checks expose semantic errors (*e.g.*, invoking unavailable APIs that static methods cannot detect).

Limitations. As a novel context-aware automated vulnerability repair system for PHP web applications, SLICEPATCH presents both opportunities and limitations. SLICEPATCH requires functional PoC exploits for profiling and fault localization, which restricts it to vulnerabilities with available exploits. Nevertheless, this requirement aligns with standard practice, as maintainers routinely use PoCs to confirm vulnerabilities, and crafting a PoC from a vulnerability report demands less effort than manually localizing the flaw, generating a fix, and validating the patch. Another practical limitation is the potential for false alerts when the vulnerable scripts can modify permission settings or database configuration. Mitigation strategies include database backup and restoration or configuration state management during fuzzing, though these techniques introduce additional overhead. Additionally, in rare cases where multiple sinks of the same vulnerability type targeting the same parameters reside in different functions, SLICEPATCH may produce a larger code slice or the LLM may localize to an incorrect sink even across multiple localization rounds. Our manual inspection found that over 30% of cases contain multiple same-type sinks within the same file, but instances where same-type sinks additionally operate on the same PoC parameter are rarer (around 9%) and typically reside within the same function body, where the slicing result is unaffected due to function-level granularity. Finally, the current SLICEPATCH prototype focuses on three specific vulnerability types (SQL Injection, Cross-Site Scripting, and Command Injection) due to the limited bug oracles of the underlying directed fuzzing tool Predator [33]. However, these three types represent the most critical and prevalent web application vulnerabilities, providing sufficient evidence to demonstrate SLICEPATCH's effectiveness.

Future Work. While focusing on three prevalent vulnerability types in PHP web applications, SLICEPATCH's design is inherently language- and vulnerability type-agnostic. SLICEPATCH's context-aware fault localization and directed differential fuzzing-based patch validation can be adapted to other languages with appropriate tooling. Specifically, to support more programming languages, one would need to integrate compatible slicing tools and directed fuzzers. Expanding the vulnerability type support in directed fuzzing tools like Predator would enable SLICEPATCH to address a broader spectrum of security weaknesses. Future research could also explore hybrid validation techniques complementing directed differential fuzzing.

8. Related Work

LLM-Based Vulnerability Repair. Recent advances in large language models have revolutionized automated vulnerability repair through sophisticated code understanding capabilities. Pearce et al. [27] demonstrated that LLMs face significant challenges in real-world security bug fixes due to the complexity of generating functionally correct repair code. To address these limitations, Appatch [24] fine-tunes LLMs using vulnerable code samples and correct patch examples, relying on multi-LLM voting to determine patch correctness, but this approach requires extensive training data and relies on potentially unreliable voting mechanisms instead of actual execution. In contrast, SLICEPATCH operates zero-shot without additional training data and validates patches through PoC replay and directed differential fuzzing. PatchAgent [38] introduces autonomous code retrieval and patch generation for C/C++ programs. SLICEPATCH instead adopts a context-aware fault localization strategy aimed at improving reliability and explicitly tackles dynamic language challenges that prior tools only partially address.

Web Application Vulnerability Repair. Limited research has addressed automated repair specifically for web applications. PAR [12] addresses the limitation of genetic programming approaches like GenProg by manually analyzing human-written patches to extract common fix patterns. FixMeUp [30] extracts and applies access control templates for sensitive operations, while SkyPort [29] serves as a backporting tool for injection vulnerabilities. Pfortifier [26] automatically generates minimal patch sets for gadget chain vulnerabilities using heuristic rules and semantic simulation. WAP [19, 20] employs static taint analysis to track user inputs to sensitive sinks and automatically injects predefined sanitization code to remediate related input-validation flaws. These approaches demonstrate narrow scope and limited extensibility to diverse vulnerability types found in modern web applications, whereas SLICEPATCH provides a generalized framework extendable to new vulnerabilities without significant manual rule engineering.

9. Conclusion

This paper presents SLICEPATCH, a novel automated vulnerability repair framework designed for PHP web ap-

plications. SLICEPATCH addresses critical gaps in PHP web application AVR by introducing two key innovations: context-aware fault localization that leverages PoC replay to slice minimal vulnerability-specific context from large dynamic codebases, and directed differential fuzzing for comprehensive patch validation. The high generalization capabilities of this workflow make it suitable for securing the vast ecosystem of PHP web applications.

SLICEPATCH establishes a new paradigm for automated vulnerability repair in dynamic PHP web application environments, offering a practical solution. Our comprehensive evaluation across 96 real-world vulnerabilities and 7 state-of-the-art LLMs demonstrates SLICEPATCH's effectiveness, achieving up to 94.79% repair success rate for the best single-model configuration. The fault localization approach reduces codebase volume by 80.80% to 99.92%, enabling precise patch generation while maintaining computational efficiency. Directed differential fuzzing proves essential, identifying 35.70% of defective patches that PoC replay alone would otherwise miss.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments. The work described in this paper was partly supported by a grant from the Research Grants Council of the Hong Kong SAR, China (Project No.: CUHK 14211325).

Ethics Considerations

Our research fully complies with established ethical guidelines for cybersecurity research, focusing exclusively on *defensive* security measures rather than offensive capabilities. SLICEPATCH is designed as an automated vulnerability repair system that enhances software security by generating patches to remediate identified security flaws, thereby contributing to the protection of users and systems from potential threats. The system operates strictly within controlled environments, ensuring no unauthorized access to sensitive data or systems. All experiments were conducted on isolated test environments without impacting production systems or real-world users, eliminating any risk of unintended harm or service disruption. By automating the vulnerability remediation process, our work directly supports the cybersecurity community's efforts to reduce the window of exposure for security vulnerabilities and mitigate potential attack vectors.

LLM Usage Considerations

SLICEPATCH leverages commercial models in the fault localization and patch generation steps. While static slicing or rule-based repair tools could, in principle, be employed for these steps, LLMs offered better generalization capabilities across programming languages and vulnerability types. Every LLM-generated patch is validated via PoC replay, directed differential fuzzing, and manual review before being accepted. During our research, no proprietary data was supplied to the LLMs. Only publicly available PoCs and code were used as inputs to the LLMs. Nor did we leverage LLMs to generate ideas or text for this paper.

References

[1] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P)*, Paris, France, April 2017.

[2] Libo Chen, Quanpu Cai, Zhenbang Ma, Yanhao Wang, Hong Hu, Minghang Shen, Yue Liu, Shanqing Guo, Haixin Duan, Kaida Jiang, et al. Sfuzz: Slice-based fuzzing for real-time operating systems. In *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*, Los Angeles, CA, USA, November 2022.

[3] Zimin Chen, Steve Kommrusch, and Martin Monperrus. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering*, 2022.

[4] Jianlei Chi, Yu Qu, Ting Liu, Qinghua Zheng, and Heng Yin. Seqtrans: automatic vulnerability fix via sequence to sequence learning. *IEEE Transactions on Software Engineering*, 2022.

[5] Nariyoshi Chida and Tachio Terauchi. Repairing dos vulnerability of real-world regexes. In *Proceedings of the 43th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, USA, May 2022.

[6] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[7] Richard A DeMillo, Hsin Pan, and Eugene H Spafford. Critical slicing for software fault localization. *ACM SIGSOFT Software Engineering Notes*, 1996.

[8] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. Automating patching of vulnerable open-source software versions in application binaries. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, February 2019.

[9] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. Black widow: Blackbox data-driven web scanning. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, USA, May 2021.

[10] Jacob Harer, Onur Ozdemir, Tomo Lazovich, Christopher Reale, Rebecca Russell, Louis Kim, et al. Learning to repair software vulnerabilities with generative adversarial networks. *Advances in neural information processing systems*, 2018.

[11] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. A survey on

automated program repair techniques. *arXiv preprint arXiv:2303.18184*, 2023.

[12] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, USA, May 2013.

[13] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, Minneapolis, MN, USA, May 2007.

[14] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, Zurich, Switzerland, June 2012.

[15] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 2011.

[16] Penghui Li, Wei Meng, Mingxue Zhang, Chenlin Wang, and Changhua Luo. Holistic concolic execution for dynamic web applications via symbolic interpreter analysis. In *Proceedings of the 45th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, USA, May 2024.

[17] Ying Li, Faysal Hossain Shezan, Bomin Wei, Gang Wang, and Yuan Tian. Sok: Towards effective automated vulnerability repair. *arXiv preprint arXiv:2501.18820*, 2025.

[18] Changhua Luo, Penghui Li, and Wei Meng. TChecker: Precise static inter-procedural analysis for detecting taint-style vulnerabilities in php applications. In *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*, Los Angeles, CA, USA, November 2022.

[19] Ibéria Medeiros, Nuno Neves, and Miguel Correia. Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Transactions on Reliability*, 2015.

[20] Ibéria Medeiros, Nuno F Neves, and Miguel Correia. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In *Proceedings of the 21st International World Wide Web Conference (WWW)*, Seoul, Korea, April 2014.

[21] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, Xudong Liu, and Chunming Hu. Template-based neural program repair. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, Melbourne, Australia, May 2023.

[22] Mahmoud Mohammadi, Bill Chu, and Heather Richter Lipford. Automated repair of cross-site scripting vulnerabilities through unit testing. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2019.

[23] Aniruddhan Murali, Noble Mathews, Mahmoud Alfadel, Meiyappan Nagappan, and Meng Xu. Fuzzslice: Pruning false positives in static analysis warnings through function-level fuzzing. In *Proceedings of the 46th International Conference on Software Engineering (ICSE)*, Lisbon, Portugal, May 2024.

[24] Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. Appatch: Automated adaptive prompting large language models for real-world software vulnerability patching. In *Proceedings of the 34th USENIX Security Symposium (Security)*, Seattle, WA, USA, August 2025.

[25] Eric Olsson, Benjamin Eriksson, Adam Doupé, and Andrei Sabelfeld. {Spider-Scents}: Grey-box database-aware web scanning for stored {XSS}. In *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, USA, August 2024.

[26] Bo Pang, Yiheng Zhang, Mingzhe Gao, Junzhe Zhang, Ligeng Chen, Mingxue Zhangt, and Gang Liang. Pfortifier: Mitigating php object injection through automatic patch generation. In *Proceedings of the 46th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, USA, May 2025.

[27] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, USA, May 2023.

[28] Youkun Shi, Yuan Zhang, Tianhao Bai, Feng Xue, Jiarun Dai, Fengyu Liu, Lei Zhang, Xiapu Luo, and Min Yang. Xssky: Detecting xss vulnerabilities through local path-persistent fuzzing. In *Proceedings of the 34th USENIX Security Symposium (Security)*, Seattle, WA, USA, August 2025.

[29] Youkun Shi, Yuan Zhang, Tianhan Luo, Xiangyu Mao, Yinzhi Cao, Ziwen Wang, Yudi Zhao, Zongan Huang, and Min Yang. Backporting security patches of web applications: A prototype design and implementation on injection vulnerability patches. In *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, USA, August 2022.

[30] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. Fix me up: Repairing access-control bugs in web applications. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, February 2013.

[31] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, USA, May 2023.

[32] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. *ACM Sigplan Notices*,

2009.
[33] Chenlin Wang, Wei Meng, Changhua Luo, and Penghui Li. Predator: Directed web application fuzzing for efficient vulnerability validation. In *Proceedings of the 46th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, USA, May 2025.
[34] Enze Wang, Jianjun Chen, Wei Xie, Chuhan Wang, Yifei Gao, Zhenhua Wang, Haixin Duan, Yang Liu, and Baosheng Wang. Where urls become weapons: Automated discovery of ssrf vulnerabilities in web applications. In *Proceedings of the 45th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, USA, May 2024.
[35] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, Vancouver, Canada, May 2009.
[36] Susheng Wu, Ruisi Wang, Yiheng Cao, Bihuan Chen, Zhuotong Zhou, Yiheng Huang, JunPeng Zhao, and Xin Peng. Mystique: Automated vulnerability patch porting with semantic and syntactic-enhanced llm. *Proceedings of the ACM on Software Engineering*, 2025.
[37] Dandan Xu, Di Tang, Yi Chen, XiaoFeng Wang, Kai Chen, Haixu Tang, and Longxing Li. Racing on the negative force: Efficient vulnerability {Root-Cause} analysis through reinforcement learning on counterexamples. In *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, USA, August 2024.
[38] Zheng Yu, Ziyi Guo, Yuhang Wu, Jiahao Yu, Meng Xu, Dongliang Mu, Yan Chen, and Xinyu Xing. Patchagent: A practical program repair agent mimicking human expertise. In *Proceedings of the 34th USENIX Security Symposium (Security)*, Seattle, WA, USA, August 2025.
[39] Quanjun Zhang, Chunrong Fang, Yang Xie, YuXiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. A systematic literature review on large language models for automated program repair. *ACM Transactions on Software Engineering and Methodology*, 2024.
[40] Addresssanitizerleaksanitizer, 2017. https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer.
[41] Awesome web pocs, 2025. https://github.com/MarkLee131/awesome-web-pocs/.
[42] Claude documentation, 2025. https://docs.claude.com/en/docs/about-claude/models/overview/.
[43] Cve - common vulnerabilities and exposures, 2025. https://cve.mitre.org/.
[44] Prompthub - deepseek-v3 model card, 2025. https://www.prompthub.us/models/deepseek-v3.
[45] Doctor appointment booking system, 2020. https://projectworlds.in/free-projects/php-projects/online-doctor-appointment-booking-system-php-and-mysql/.
[46] Friendsofphp security advisories, 2025. https://github.com/FriendsOfPHP/security-advisories.
[47] Github advisory database, 2025. https://github.com/advisories.
[48] Google deepmind, 2025. https://deepmind.google/models/gemini/.
[49] Hospital management system, 2019. https://phpgurukul.com/hospital-management-system-in-php/.
[50] Huntr - vulnerability disclosure platform, 2025. https://huntr.com/.
[51] Joomla!, 2025. https://github.com/joomla/joomla-cms.
[52] Joomla security announcements, 2025. https://developer.joomla.org/security-centre.html.
[53] User login management system, 2020. https://phpgurukul.com/user-registration-login-and-user-management-system-with-admin-panel/.
[54] Nvd - national vulnerability database, 2025. https://nvd.nist.gov/.
[55] Openai documentation, 2025. https://platform.openai.com/docs/models/.
[56] Openemr, 2025. https://github.com/openemr/openemr.
[57] Openemr security announcements, 2025. https://github.com/openemr/openemr/security.
[58] phpipam, 2025. https://github.com/phpipam/phpipam.
[59] phpmyadmin, 2025. https://github.com/phpmyadmin/phpmyadmin.
[60] Php-parser, 2025. https://github.com/nikic/PHP-Parser.
[61] Piwigo, 2025. https://github.com/Piwigo/Piwigo.
[62] rconfig, 2025. https://github.com/rconfig/rconfig-v3.
[63] Uk cyber security survey, 2025. https://www.gov.uk/government/statistics/cyber-security-breaches-survey-2025/cyber-security-breaches-survey-2025.
[64] Usage statistics of server-side programming languages for websites, 2025. https://w3techs.com/technologies/overview/programming_language.
[65] Webchess, 2025. https://github.com/halojoy/PHP7-Webchess.
[66] Webid, 2025. https://github.com/renlok/WeBid.
[67] Wordpress, 2025. https://github.com/WordPress/WordPress.
[68] Xdebug, 2025. https://xdebug.org/.

Appendix A. Dataset Details

A.1. Application Summary

Table 9 summarizes our evaluation corpus. It spans large, critical platforms such as OpenEMR and phpMyAdmin, widely deployed content management systems like WordPress, Joomla, and Piwigo, and lightweight administrative portals including Login Management and Doctor Appointment systems. The table reports application version, approximate codebase size, and the number of evaluated vulnerabilities, complemented with GitHub stars and test suite availability. We additionally list representative prior studies that have analyzed each application to highlight their continued relevance in vulnerability research. Many applications ship without test suites, and the suites that do exist are outdated and incomplete, so their validity cannot be guaranteed. We therefore relied on manual validation to confirm each patch, and spent approximately 70 person-hours.

TABLE 9: Dataset summary. "-" denotes applications without GitHub repositories or with negligible star counts. We denote test suite availability of the corresponding application version with ● (available), ◐ (limited), and ○ (unavailable).

Application	Version	LoC	Vulns	Stars	Tests	Prior Studies
OpenEMR [56]	5.0	1,839.5K	10	3.6K	●	[16, 28]
Joomla [51]	3.7	529.6K	2	5.0K	●	[18, 33]
WordPress [67]	5.8	474.5K	1	21.0K	●	[9, 31]
phpMyAdmin [59]	5.0	325.7K	3	7.6K	◐	[16]
Piwigo [61]	13.5	241.8K	5	3.5K	◐	[25, 33]
phpIPAM [58]	1.4	143.4K	16	2.5K	○	[28]
rConfig [62]	3.9	79.5K	18	100	○	[31, 34]
WeBid [66]	1.2	53.2K	11	120	○	[18, 34]
Hospital Mgmt. [49]	4.0	14.5K	10	-	○	[25, 31]
Webchess [65]	0.9	5.0K	2	-	○	[16, 18]
Doctor Appt. [45]	1.0	4.6K	8	-	○	[25, 31]
Login Mgmt. [53]	3.3	1.0K	10	-	○	[25, 31]

A.2. CVE Details

Table 10 presents the complete list of 96 vulnerabilities across 12 applications, organized by application and vulnerability type. SQL Injection vulnerabilities are the most prevalent (67 instances, 69.79%), followed by Cross-Site Scripting (22 instances, 22.92%) and Command Injection (7 instances, 7.29%). The dataset includes only publicly disclosed vulnerabilities for which a functional, replayable PoC exploit is available. Consequently, the number of vulnerabilities per application does not reflect the total known vulnerabilities for that application. For example, WordPress has multiple disclosed vulnerabilities, but only one has an available PoC that meets our requirements at the time of writing.

Appendix B. Unseen Vulnerability Repair

B.1. LLM Knowledge Cut-off Dates

Table 11 summarizes the knowledge cut-off dates reported for each evaluated LLM. These cut-offs bound the unseen vulnerability evaluation scope discussed in §6 and reveal how recently each provider refreshed its training data. Models with earlier cut-offs face a larger pool of unseen vulnerabilities. We collect information from official documentation [42, 48, 55] or third-party model cards [44] where available. Claude 4 Sonnet is the most up-to-date model in our evaluation (2025-03). In contrast, GPT-4o and DeepSeek-V3 have earlier cut-offs (2023-10 and 2024-07). The Gemini 2.5 releases fall between these extremes with a shared 2025-01 cut-off.

B.2. Repair Outcomes

To complement the main text, Table 12 reports the per-vulnerability repair outcomes for vulnerabilities disclosed

TABLE 10: Detailed vulnerability dataset by application and vulnerability type.

Application	Vuln. Type	CVE IDs
OpenEMR	SQLi (8)	CVE-2018-17179, CVE-2019-14529 CVE-2019-16404, CVE-2020-29139 CVE-2020-29140, CVE-2020-29142 CVE-2020-29143, CVE-2021-41843
	XSS (2)	CVE-2022-4502, CVE-2023-2949
Joomla	SQLi (2)	CVE-2017-8917, CVE-2018-8045
WordPress	SQLi (1)	CVE-2022-21661
phpMyAdmin	SQLi (3)	CVE-2020-22452, CVE-2020-26935 CVE-2020-5504
Piwigo	SQLi (5)	CVE-2023-26876, CVE-2023-27233 CVE-2023-33361, CVE-2023-33362 CVE-2023-34626
phpIPAM	SQLi (6)	CVE-2019-16692, CVE-2019-16693 CVE-2019-16694, CVE-2019-16695 CVE-2019-16696, CVE-2023-1211
	XSS (10)	CVE-2023-0676, CVE-2023-0677 CVE-2023-24657, CVE-2024-10727 CVE-2024-41353, CVE-2024-41354 CVE-2024-41355, CVE-2024-41356 CVE-2024-41357, CVE-2024-41358
rConfig	SQLi (9)	CVE-2019-19207, CVE-2020-10220 CVE-2020-10546, CVE-2020-10547 CVE-2020-10548, CVE-2020-10549 CVE-2020-15713, CVE-2021-29004 CVE-2022-45030
	XSS (2)	CVE-2020-12256, CVE-2020-12259
	CMDi (7)	CVE-2019-16662, CVE-2019-16663 CVE-2019-19509, CVE-2020-10221 CVE-2020-10879, CVE-2020-13778 CVE-2020-23151
WeBid	SQLi (5)	CVE-2018-1000867 (5)
	XSS (6)	CVE-2018-1000868, CVE-2019-11592 (5)
Hospital Mgmt.	SQLi (9)	CVE-2020-22164, CVE-2020-22165 CVE-2020-22166, CVE-2020-22168 CVE-2020-22169, CVE-2020-22171 CVE-2020-22172, CVE-2020-22173 CVE-2020-22174
	XSS (1)	CVE-2021-39411
Webchess	SQLi (2)	CVE-2019-20896, CVE-2023-22959
Doctor Appt.	SQLi (8)	CVE-2020-29283, CVE-2025-3178 CVE-2025-3179, CVE-2025-3180 CVE-2025-3181, CVE-2025-3182 CVE-2025-3183, CVE-2025-3186
Login Mgmt.	SQLi (9)	CVE-2024-11817, CVE-2024-48280 CVE-2024-48282, CVE-2024-48283 CVE-2025-2050, CVE-2025-28011 CVE-2025-4934, CVE-2025-7542 CVE-2025-7543
	XSS (1)	CVE-2024-48284

TABLE 12: Unseen vulnerability repair results. "/" denotes vulnerabilities fixed before the corresponding model cut-off. *N/A* indicates the vulnerability has not been fixed at the time of writing.

Application	CVE ID	Type	Release Date	Fix Date	GPT-4o	DeepSeek V3	GPT-5	Claude 3.7 S	Claude 4 S	Gemini 2.5 P	Gemini 2.5 F
phpIPAM	CVE-2024-41357	XSS	2024-07-26	2024-07-16	✓	✓	/	/	/	/	/
	CVE-2024-41353	XSS	2024-07-26	2024-07-16	✓	✓	/	/	/	/	/
	CVE-2024-41358	XSS	2024-08-29	2024-07-16	✓	✓	/	/	/	/	/
	CVE-2024-41355	XSS	2024-07-26	2024-07-16	✓	✓	/	/	/	/	/
	CVE-2024-10727	XSS	2025-03-20	2024-10-24	×	×	✓	/	/	/	/
	CVE-2024-41356	XSS	2024-07-26	2024-07-16	×	✓	/	/	/	/	/
	CVE-2024-41354	XSS	2024-07-26	2024-07-16	✓	✓	/	/	/	/	/
rConfig	CVE-2021-29004	SQL	2021-10-11	N/A	✓	×	✓	✓	✓	✓	×
	CVE-2022-45030	SQL	2023-04-14	N/A	✓	✓	✓	✓	✓	✓	×
Webchess	CVE-2023-22959	SQL	2023-01-10	N/A	×	✓	✓	✓	✓	×	×
Hospital Mgmt.	CVE-2021-39411	XSS	2021-11-05	N/A	✓	✓	✓	✓	✓	✓	✓
Doctor Appt.	CVE-2025-3183	SQL	2025-04-03	N/A	✓	✓	✓	✓	✓	✓	✓
	CVE-2025-3178	SQL	2025-04-03	N/A	✓	✓	✓	✓	✓	✓	✓
	CVE-2025-3186	SQL	2025-04-03	N/A	✓	✓	✓	✓	✓	✓	✓
	CVE-2025-3181	SQL	2025-04-03	N/A	✓	✓	✓	✓	✓	✓	✓
	CVE-2025-3180	SQL	2025-04-03	N/A	✓	✓	✓	✓	✓	✓	✓
	CVE-2025-3182	SQL	2025-04-03	N/A	✓	✓	✓	✓	✓	✓	✓
	CVE-2025-3179	SQL	2025-04-03	N/A	✓	✓	✓	✓	✓	✓	✓
Login Mgmt.	CVE-2025-7543	SQL	2025-07-13	N/A	✓	✓	✓	✓	✓	✓	✓
	CVE-2024-48280	SQL	2024-10-15	N/A	✓	✓	✓	✓	✓	✓	✓
	CVE-2025-2050	SQL	2025-03-06	N/A	✓	×	✓	✓	✓	✓	×
	CVE-2025-4934	SQL	2025-05-19	N/A	✓	✓	✓	✓	✓	✓	✓
	CVE-2025-7542	SQL	2025-07-13	N/A	✓	✓	✓	✓	✓	✓	✓
	CVE-2024-48282	SQL	2024-10-15	N/A	✓	✓	✓	✓	✓	✓	✓
	CVE-2024-11817	SQL	2024-11-24	N/A	✓	✓	✓	✓	✓	✓	✓
	CVE-2025-28011	SQL	2025-03-13	N/A	✓	✓	✓	✓	✓	✓	✓
	CVE-2024-48283	SQL	2024-10-15	N/A	✓	✓	✓	✓	✓	✓	✓
	CVE-2024-48284	XSS	2024-11-14	N/A	✓	✓	✓	✓	✓	✓	✓
Total					25/28	25/28	22/22	21/21	21/21	20/21	17/21
Success %					89.29%	89.29%	100.00%	100.00%	100.00%	95.24%	80.95%

TABLE 11: Knowledge cut-off dates for evaluated LLMs.

Provider	Model	Knowledge Cut-off
OpenAI	GPT-5	2024-09
OpenAI	GPT-4o	2023-10
Anthropic	Claude 3.7 Sonnet	2024-11
Anthropic	Claude 4 Sonnet	2025-03
DeepSeek	DeepSeek-V3	2024-07
Google	Gemini 2.5 Pro	2025-01
Google	Gemini 2.5 Flash	2025-01

after each LLM's knowledge cut-off. Earlier cut-offs expand the unseen window, which is why the Total row lists 28 evaluation targets for GPT-4o and DeepSeek-V3 but only 21-22 for other models.

The table exposes several notable patterns. GPT-5 and both Claude releases achieve full recall on all unseen vulnerabilities within their evaluation windows. GPT-4o and DeepSeek-V3, whose cut-offs lag the Anthropic models, each miss three vulnerabilities. Gemini 2.5 Pro nearly matches the Anthropic results with a single miss in Webchess, whereas Gemini 2.5 Flash misses four vulnerabilities. Across applications, Doctor Appointment vulnerabilities are repaired by every model once they fall within the unseen window, while Webchess, phpIPAM, and rConfig furnish the most challenging late disclosures for models with earlier cut-offs.

Appendix C.
Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

C.1. Summary

This paper focuses on an automated vulnerability repair framework for PHP applications. The authors leverage PoC-driven dynamic profiling to capture runtime behaviors, retrain code paths, and then guide LLMs for precise patches.

C.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field

C.3. Reasons for Acceptance

1) This paper outperforms chosen baselines, and strong evaluations back the performance.
2) The paper employs clever techniques for handling issues with static analysis on dynamic programming languages through slicing and staticization of dynamic constructs.