ZendDiff: Differential Testing of PHP Interpreter

Yuancheng Jiang*§, Jianing Wang*†§, Qiange Liu‡, Yeqi Fu*, Jian Mao‡, Roland H. C. Yap*, Zhenkai Liang*
*School of Computing, National University of Singapore, Singapore
{yuancheng, yeqi.fu, ryap, liangzk}@comp.nus.edu.sg
†Shandong University, China
jianingwang@mail.sdu.edu.cn
‡Beihang University, China
{liuqiangebuaa, maojian}@buaa.edu.cn

***Abstract*—The PHP interpreter, powering over 70% of websites on the internet, plays a crucial role in web development. Existing approaches to finding bugs in PHP primarily focus on detecting explicit security issues through crashes or sanitizer-based oracles, but fail to identify logic bugs that can silently lead to incorrect results. We observe that the introduction of Just-In-Time (JIT) compilation mode in PHP presents an opportunity for differential testing, as it provides an alternative implementation of the same language specification. We propose, `ZendDiff`, an automatic differential testing framework that effectively detects logic bugs in the PHP interpreter by comparing JIT and non-JIT execution results. Our differential testing incorporates three techniques: program state probing for fine-grained execution state comparison, JIT-aware program mutation to sufficiently exercise JIT functionality, and dual verification to handle non-deterministic behaviors in PHP programs. Our experimental results demonstrate that `ZendDiff` outperforms the official test suite used in PHP's continuous integration, achieving higher code coverage and executing more Zend opcodes. Through ablation studies, we validate the effectiveness of these techniques. To date, `ZendDiff` has identified 51 previously unknown logic bugs in the PHP interpreter, with 37 already fixed and 3 confirmed by the PHP maintainers. `ZendDiff` has been acknowledged by the PHP community and offers a practical tool for automatically discovering logic bugs in the PHP interpreter.**

***Index Terms*—differential testing, just-in-time compilation, logic bug detection, PHP interpreter, software testing**

I. Introduction

PHP (Hypertext Preprocessor) is one of the most widely used server-side scripting languages for web development, powering over 70% of websites on the internet [1, 2]. PHP offers a wide range of functionalities for web developers, such as database access, session management, and file system operations. The official PHP interpreter features a substantial codebase with over a million lines of code, mainly implemented in C. It is well known that complex software is prone to bugs and vulnerabilities. Given its critical role in web development, detecting bugs in the PHP interpreter is crucial to ensure the security and reliability of web applications.

Research efforts [3, 4, 5, 6, 7, 8] have been devoted to uncovering various bugs in the PHP interpreter. In particular, the proposed approaches mainly rely on fuzzing techniques dedicated to optimizing input generation for broad testing coverage. Most of the above works leverage grammar-guided or semantic-guided test case generation. For example, FlowFusion [7], utilizes dataflow-guided test case fusion techniques to generate semantic test cases. We highlight that existing approaches mainly focus on finding explicit security issues, such as crashes or memory errors. They do so by leveraging crashes or sanitizers as weak *test oracles*. Despite the effectiveness of these approaches in identifying security issues, they are not designed for detecting logic bugs. Logic bugs are dangerous as they can *silently* lead to incorrect results, thereby compromising the reliability of the PHP interpreter.

Finding logic bugs in the PHP interpreter is challenging due to the lack of an effective test oracle. As logic bugs manifest as incorrect computation results rather than explicit issues, like crashes or memory errors, they require a more sophisticated test oracle to detect. Metamorphic testing [9, 10, 11, 12, 13, 14, 15, 16] and differential testing [17, 18, 19, 20, 21, 22, 23, 24, 25, 26] are two general approaches which can be leveraged to create effective test oracles to detect logic bugs. However, they face the following challenges when applied to the PHP interpreter. For metamorphic testing, defining metamorphic relations requires significant manual effort and expert knowledge, which is neither automated nor scalable. For differential testing, there is no good alternative implementation of the PHP interpreter (HHVM [27] ended support for PHP on 2019 [28]). The situation is different with JavaScript, which has multiple independent modern implementations (*e.g.,* V8, WebKit, Gecko).

We notice that starting with PHP 8.0, released in 2020, the PHP interpreter ships with a just-in-time (JIT) compilation mode designed to accelerate script execution [29]. Instead of feeding bytecode through the Zend virtual machine, the frequently called part (hot functions) of the program or even the entire script can be translated into native machine instructions and executed directly, to reduce interpretation overhead [30]. Although JIT and non-JIT modes are both required to implement the same PHP language specification, they follow distinct execution paths. This implementation diversity creates an ideal setting for *differential testing*: by running the same PHP program in both modes and comparing their observable results, any inconsistency signals a latent miscomputation (*i.e.,* a logic bug in the engine). Building on

§Co-primary authors. †Corresponding author.

prior work [31, 17, 32], we propose a new test oracle designed specifically for detecting logic bugs in the PHP engine, which flags inconsistencies between JIT-compiled and interpreter-executed runs as potential defects. While differential testing is a well-established paradigm, its application to a new domain requires a *tailored* and *practical* oracle. However, several obstacles prevent a naive implementation from being effective in practice. We outline these challenges and motivate the refinements necessary for a robust PHP-specific differential-testing framework.

- **Challenge C1: Inconsistency detection in the presence of non-determinism.** Program outputs can depend on inherently non-deterministic factors (such as memory addresses, thread scheduling, execution timing, or random number generation), so two correct runs can legitimately differ, which then affects differential testing. To suppress these benign inconsistencies, patching the engine [17, 33] or maintaining an ad-hoc "allow lists" are feasible, but ensuring their comprehensiveness is labor-intensive and inherently fragile. Therefore, a principled and general-purpose mechanism for tolerating non-deterministic behavior during differential comparison is needed.
- **Challenge C2: Exercising JIT functionality sufficiently.** The extent to which a PHP program actually triggers JIT features largely determines how effective JIT-based testing will be. Unlike V8's automatic and tiered JIT optimization strategy, the PHP engine provides fine-grained, operator-visible controls, such as switching between function/tracing modes, tuning hotness thresholds, and sizing the native-code cache buffer. Consequently, merely toggling a JIT configuration flag rarely exposes the full PHP's optimizations.
- **Challenge C3: Probing internal miscomputations.** Logic errors can lurk behind correct-looking outputs [17, 34]. For example, the PHP JIT may compute an incorrect intermediate value that is subsequently discarded, leaving the program's final result unaltered. A differential fuzzer will miss such defects unless the affected state is explicitly inspected. This requires us to capture execution state in a fine-grained way, exposing these subtle discrepancies before they surface in user-visible behavior.

To tackle the challenges mentioned above, we propose an automatic differential testing framework, called `ZendDiff`, for automatically discovering logic bugs in the PHP interpreter with three key techniques: *program state probing*, *JIT-aware program mutation*, and *dual verification*. To reveal internal miscomputations (**C3**), program state probing injects lightweight probes into each candidate program, which records additional program states (*e.g.,* variable values, class attributes). These states enable a more fine-grained comparison between non-JIT and JIT executions, revealing silent logic errors that would otherwise slip through. To exercise JIT functionality sufficiently (**C2**), JIT-aware program mutation applies a suite of tactics (tweaking configuration flags, injecting loop nests, and restructuring functions) to raise execution frequency and force compilation, thus triggering a broader spectrum of JIT optimizations. To mitigate the impact of non-deterministic behaviors (**C1**), dual verification executes each test case twice under both execution modes. Specifically, it first runs the original program P in interpreted mode and its JIT-compiled counterpart P_{jit}, obtaining results $R(P)$ and $R(P_{\text{jit}})$. Then, it creates logically equivalent replicas P' and P'_{jit} and reruns them to collect $R(P')$ and $R(P'_{\text{jit}})$. Beyond the naive oracle that flags discrepancies between $R(P)$ and $R(P_{\text{jit}})$, dual verification enforces consistency across repeated runs: $R(P) = R(P')$ and $R(P_{\text{jit}}) = R(P'_{\text{jit}})$. Any test that fails this criterion is considered non-deterministic and discarded. This enhanced oracle significantly reduces false positives, improving the reliability of bug reports.

Listing 1: Unknown logic bug found by `ZendDiff` in Zend

```
<?php
  $root = simplexml_load_string('../></root>');
  $spattr = $root->child->attributes('special-ns');
  var_dump(Dom\import_simplexml($spattr));
Non-JIT: Error.. Return value must be of type ..
JIT: object(Dom\Attr)#2 (22) { .. }
```

Listing 1 presents an example of a previously unknown logic bug found by `ZendDiff`. This logic bug is associated with the Document Object Model (DOM) handling in the PHP interpreter. We observe that the result of non-JIT execution shows an error with the return value of `dom_import_simplexml()`. However, the JIT execution inconsistently returns the value normally, without errors. Our analysis indicates that a missed type annotation in the function definition causes an unexpected type error under non-JIT execution, thereby preventing the interface from operating with `DOMAttr` variables. The PHP developers quickly fixed this issue and clarified that the behavior of non-JIT execution was unexpected. Notably, this bug has persisted in the PHP interpreter for over four years.

We implement our framework with over 2,000 lines of Python code. We conducted comprehensive experiments to assess the effectiveness of our approach. `ZendDiff` detects 51 unique and previously unknown logic bugs in the PHP engine, of which 37 have been fixed and 3 confirmed. We note that these bugs cover both the non-JIT and JIT components of the PHP engine. We compared the effectiveness of `ZendDiff` against the official test suites. The results demonstrate that `ZendDiff` not only detects more logic bugs but also increases the diversity of executed Zend opcodes and achieves higher code coverage. Furthermore, we conduct an ablation study to showcase the contributions of each component. `ZendDiff` has been acknowledged by the PHP developers. In summary, we make the following contributions:

- We design an oracle for discovering logic bugs in the PHP engine by identifying the execution inconsistencies between non-JIT and JIT modes, combined with a novel dual verification mechanism to minimize false positives from non-deterministic behaviors.
- We present JIT-aware program mutation to sufficiently exercise PHP's JIT functionality and program state probing to collect fine-grained internal states, and further implement

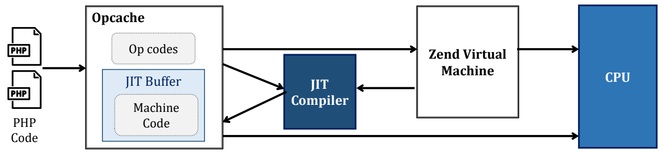

Fig. 1: Just-In-Time (JIT) Overview in the PHP Interpreter

the first automatic framework, `ZendDiff`[1], for detecting logic bugs in the PHP interpreter through differential testing.

- `ZendDiff` has identified 51 unknown logic bugs, with 37 fixes merged and 3 confirmations from PHP developers. Our bug reports were positively received and acknowledged by the developers.

II. Background

PHP Interpreter and JIT Compilation. PHP is an interpreted language, with the underlying Zend Virtual Machine for executing the PHP program at runtime. This interpreted execution mode enables easier development but results in slower execution. In contrast, Ahead-of-Time (AOT) compiled languages, like C, require the code to be fully compiled into machine code before execution, offering faster performance at the cost of an extra compilation step. Just-In-Time (JIT) [30] compilation stands in the middle ground of these two execution modes. JIT compiles parts of the code "just in time" during execution, rather than all at once beforehand.

PHP added JIT compilation in version 8.0. Figure 1 outlines the PHP engine. The `Opcache` extension translates PHP source into platform-agnostic "opcodes" and caches them in memory. Without JIT, the Zend VM interprets these opcodes. With JIT, it monitors the *hot code* paths frequently executed by the Zend VM, then compiles them into optimized machine code on the fly, storing them in a JIT buffer. The CPU then executes the optimized machine code directly, bypassing the need for interpretation by the VM.

In PHP, JIT compilation is configured via the `php.ini` file with key settings, such as `opcache.jit_buffer_size`, which specifies the memory allocation for native code generation. The core setting of JIT is `opache.jit`, consisting of four configurable options, denoted by four decimal digits for `CRTO`: (i) `C` whether CPU-specific optimizations are applied, with options to disable these optimizations or enable advanced instruction sets like AVX. (ii) `R` defines the strategy for register allocation, including no allocation, block-local allocation, and global register allocation. (iii) `T` determines the JIT trigger, which governs when code undergoes JIT compilations. Options include compiling all functions upon script load, triggering compilation on first execution, after profiling specific requests, or dynamically during profiling and tracing, *etc.* (iv) `O` indicates the optimization level, indicating the extent and methodology of JIT compilation. It offers configurations such as minimal JIT, type inference-based compilation, call graph-based optimization [35], *etc.*

[1]`ZendDiff` is available at https://github.com/YuanchengJiang/ZendDiff.

Official PHP Test Suite. The PHP community maintains an official test suite for automated continuous integration, which helps uncover logic bugs and security vulnerabilities. This suite comprises over 18,000 distinct test cases, each designed to verify a broad range of code semantics with valid syntax. These cases span more than 80 unique modules within the interpreter and include additional security-focused cases based on all known issues. Each test is a `.phpt` file split by `--SECTION--` markers; more than 30 section types exist [36]. Listing 2 shows an example that verifies the correctness of DOM-related functions. Key sections include: (i) `--TEST--` section for a brief description; (ii) `--EXTENSIONS--` section lists the required extensions; (iii) the `--FILE--` section contains the PHP program; and (iv) the `--EXPECT--` section specifies the expected results (a mismatch signals a bug).

Existing PHP Program Generator. While test program generation is orthogonal to our approach, it remains a crucial research challenge in the testing programming language implementations. As the PHP program generator, `ZendDiff` adopts FlowFusion [7], the state-of-the-art and only generator built for PHP. FlowFusion aims to uncover memory errors within the PHP interpreter through fuzzing. To better explore the PHP execution space, it works by fusing official test-suite cases at the data-flow level: it treats cases from the official suite as seeds, interleaves their data flows, and produces new programs that exercise new execution paths and have not been previously explored.

III. Approach

We aim to detect logic bugs that produce incorrect computation results in the PHP interpreter. Our key insight is to treat PHP's JIT compilation as a semantics-preserving alternative execution implementation to the standard interpreter, enabling differential testing. We introduce a PHP-specific differential-testing oracle that runs each program in interpreter and JIT modes and compares their observable results. Discrepancies between the two executions indicate potential logic bugs, allowing us to uncover subtle miscomputations that do not manifest as explicit crashes or memory errors.

Listing 2: Example Test Case in the Official Test Suite

```
--TEST--
Delayed freeing comment node
--EXTENSIONS--
dom
--FILE--
<?php
  $doc = new DOMDocument;
  $comment = $doc->appendChild($doc->createElement('
      container'))
    ->appendChild($doc->createComment('comment'));
  echo $doc->saveXML(), "\n";
  $comment->parentNode->remove();
  echo $doc->saveXML(), "\n";
  echo $doc->saveXML($comment), "\n";
  var_dump($comment->parentNode);
?>
--EXPECT--
<?xml version="1.0"?>..<!--comment-->NULL
```

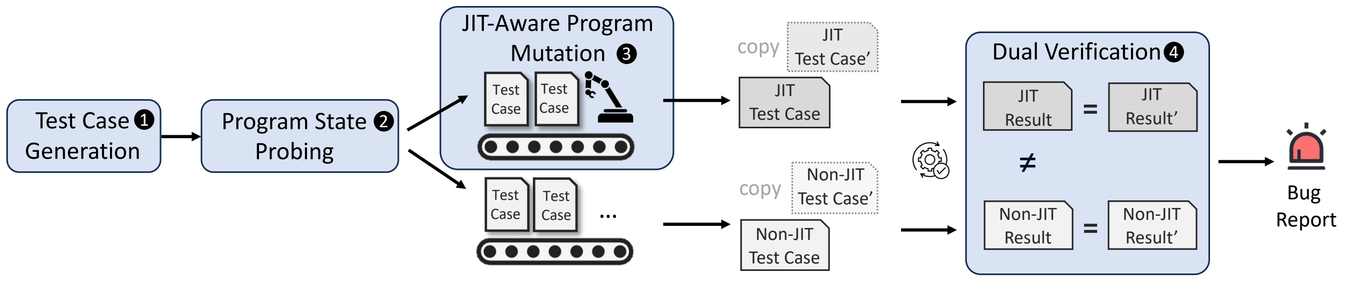

Fig. 2: The Overview of `ZendDiff`'s Differential Testing Approach

We present `ZendDiff`, a differential testing approach for detecting logic bugs in the PHP interpreter. Figure 2 illustrates the overall workflow of `ZendDiff`: *Test Case Generation* (❶) leverages an off-the-shelf test case generation approach [7] to create a large and diverse corpus. *Program State Probing* (❷) injects probes into the generated cases to capture fine-grained program states. These instrumented test cases are then replicated into two groups: one group is processed by *JIT-Aware Program Mutation* (❸) to sufficiently exercise JIT functionality of PHP engine, while the other group acts as non-JIT cases. These two groups form test pairs, which are then passed to the *Dual Verification* (❹), which cross-checks the outputs of the differential executions from both groups to detect potential discrepancies. Finally, `ZendDiff` generates bug reports upon detecting discrepancies in differential test results. Each bug report comprises the bug-inducing test case, the observed unexpected result, the expected result, and the reproducing configuration.

A. *Differential Input Preparation*

To generate unique and high-quality test cases for differential testing, `ZendDiff` leverages an existing state-of-the-art PHP test case generation technique, FlowFusion [7] (see Section II). However, directly applying these cases to differential testing does not work well because they often contain random and unstable behaviors for fuzzing purposes, such as randomized API invocations, variable accesses, and execution configurations. Such random elements can lead to differences in runtime behaviors, causing challenges for reliable differential testing. Hence, `ZendDiff` performs a processing step to stabilize such generated randomness. Specifically, `ZendDiff` performs a lightweight code analysis to identify the declared variables and explicitly specifies them in test cases. Similarly, `ZendDiff` replaces the use of random API invocations with specific API invocations to ensure stable and comparable executions.

B. *Program State Probing.*

Motivation and Scope. Differential testing fundamentally relies on comparing execution results across different testing targets. However, relying solely on the program's final output can often be insufficient to expose underlying logical errors in different execution modes. This limitation arises when some key intermediate variables, which could potentially indicate discrepancies, may be overwritten during program execution or not reflected in the final output. Therefore, capturing fine-grained program states throughout the execution is crucial to enable effective comparison and increasing the likelihood of identifying bugs. We empirically find that lightweight yet informative profiling of state information in PHP includes (i) the values bound to program variables after each definition-use site, (ii) the text emitted through `stdout`, and (iii) run-time diagnostics such as warnings and notices, whereas deeper artifacts (*e.g.,* reference-count deltas or allocator metadata) yield negligible new findings but markedly reduced execution throughput. We therefore implement a lightweight state-probing mechanism based on the above observations, striking a practical balance between informative semantics and performance overhead.

Lightweight Probe Injection. To expose intermediate states without incurring whole-program rewriting, we perform a lightweight single-pass static analysis. The analysis first marks key probing points that may write to a variable or one of its attributes. Then it injects probes for these marked points to reveal their intermediate states, including their values and class attributes. To reveal the deep states, each probes must (i) reveal rich metadata—values, type and length, (ii) recursively dump nested structures, arrays and objects.

C. *JIT-Aware Program Mutation*

As outlined in Section II, the PHP engine employs a variety of heuristics to decide *when* and *how* to compile code just-in-time—ranging from multiple trigger modes (the *T-flags*) to iteration- and call-count thresholds and code-pattern constraints. The extent to which a test case exercises these heuristics directly determines the effectiveness of differential testing. Simply enabling "global" JIT compilation is insufficient: it can bypass hot-function profiling and tracing optimizations we wish to examine. To obtain a *broad* coverage of JIT behaviors, we adopt three mutation strategies.

- *Fine-Grained JIT Trigger Controls*. PHP's JIT supports distinct modes (T = 0, 1, 2, 3, 5), each exercising different code paths (compile-all, first-call, profile-compile, on-the-fly hot, tracing). As such, we randomly vary the T-flag in combination with other CRTO bits to cover each trigger mode, and apply different strategies to it.
- *Threshold-Dependent Activation*. In PHP's `function` JIT

mode, the engine compiles all functions at load time, whereas in `tracing` JIT mode it waits until loops exceed `jit_hot_loop` iterations and functions surpass `jit_hot_func` calls. Therefore, test inputs need to drive loops/functions across those thresholds; otherwise tracing and profiling code paths remain inactive. However, it is prohibitively expensive for large-scale differential testing, if we repeatedly invoke the same loop or function to surpass the default threshold. Instead, we mutate the input in two ways: (i) wrap the original payload in an artificial hot loop or dispatch function, and (ii) lower the threshold values themselves. This lightweight transformation reliably triggers the tracing without introducing excessive runtime overhead.

- *Mutation of Opcache Configurations*. PHP's JIT behavior is governed by various `opcache` settings [37]. We randomly sample valid values and combine these settings to exercise a wider range of JIT behaviors. For instance, we set an undersized JIT buffer to test the behavior where early regions in the buffer are evicted.

D. Differential Testing with Dual Verification

In differential testing, an identical input is expected to yield consistent output within the same testing environment. However, this assumption does not always hold for PHP program execution due to the presence of non-deterministic elements, such as pseudo-random number generators and time-varying functions. This non-determinism poses a significant challenge in differential testing, as it can lead to a large number of false positives and further obscure true discrepancies in the detection process. To address this challenge, `ZendDiff` introduces a *dual verification* mechanism to minimize the impact of non-deterministic behaviors, thereby enhancing the accuracy of differential testing outcomes.

Common Non-Deterministic Behaviors. Common sources that can lead to inconsistent outputs across multiple executions include: (1) random number generators, which produce different outputs based on seed values, (2) time-dependent functions, which yield varying results based on system time, and (3) environment-dependent variables, which introduce variability based on external factors, such as memory usage, process states, and system configurations.

Dual Verification. As shown in Algorithm 1, `ZendDiff`'s dual verification mechanism aims to eliminate the impact of non-deterministic behaviors. It takes as input the instrumented PHP program P and P_{jit} obtained from the previous steps, and outputs the test result as either *Detected* or *Passed*. Specifically, the mechanism consists of the following steps. First, `ZendDiff` checks the execution results of the non-JIT and JIT programs, denoted as $R(P)$ and $R(P_{jit})$, respectively. If the results of the non-JIT and JIT executions are consistent, `ZendDiff` concludes that no logic bugs are present (line 6-7). Otherwise, `ZendDiff` proceeds to the next step. `ZendDiff` then replicates the input cases as "check tests" (line 9-10). Replication refers to another clone version of the input which should have the same output. Then, it re-executes these test cases in both non-JIT and JIT modes to obtain the corresponding results (line 11-12). Finally, `ZendDiff` compares the results of the check tests with the original test cases to ensure consistency across different executions. If the results of the check tests do not match those of the original test cases, `ZendDiff` ignore them as non-deterministic behaviors present (line 14). Otherwise, `ZendDiff` reports a logic bug, as the discrepancy indicates an inconsistency in the PHP interpreter's execution behavior (line 15).

Algorithm 1 Differential Testing with Dual Verification

```
1: Input: non-JIT program $P$, JIT program $P_{jit}$
2: Output: the test result Detected or Passed
3: function DualVerification($P$, $P_{jit}$)
4:     let $R(P) \leftarrow$ the execution result of program $P$
5:     let $R(P_{jit}) \leftarrow$ the execution result of program $P_{jit}$
6:     if $R(P) = R(P_{jit})$ then
7:         return Passed    ▷ no bug detected if differential results are equal
8:     else
9:         let $P' \leftarrow$ Replicate($P$)                ▷ clone of program $P$
10:        let $P'_{jit} \leftarrow$ Replicate($P_{jit}$)        ▷ clone of program $P_{jit}$
11:        let $R(P') \leftarrow$ the execution result of program $P'$
12:        let $R(P'_{jit}) \leftarrow$ the execution result of program $P'_{jit}$
13:        if $R(P') \neq R(P)$ or $R(P'_{jit}) \neq R(P_{jit})$ then
14:            return Passed              ▷ ignore if the results are unstable
15:        else
16:            return Detected             ▷ detected after dual verification
17:        end if
18:    end if
19: end function
```

IV. Implementation

We developed our differential testing framework, `ZendDiff`, with over 2,000 lines of Python code.

Processing Generated Test Cases. Original test cases generated by FlowFusion [7] often invoke dynamic introspection APIs such as `get_defined_functions()` and `get_defined_vars()`, which randomly access internal functions and variables. While the use of randomness can be useful for a fuzzer, it yields unstable and non-comparable executions that undermine differential testing. `ZendDiff` refines these cases by eliminating such random and unstable behaviors with fixed targets to ensure reproducible results.

State Probes. Our probes internally invokes PHP's built-in `var_dump()` to serialize variables into a canonical string that captures both type and value. When dealing with arrays and objects, it recursively traverses nested elements, exposing the full structure of multi-dimensional arrays and complex objects. This detailed output is far more informative for debugging than basic print utilities such as `echo` or `print_r()`.

Bug Verifier and PHP Program Reducer. We use the reproducing scripts from PHP official to verify detected logic errors. We implemented our PHP program reducer using delta debugging [38]. It systematically comments out specific lines or groups of lines to determine whether the bug oracle continues to trigger this issue. If the issue persists, those lines are discarded, resulting in a reduced version of the program. This process is repeated iteratively until no smaller reproducer can be found. In practice, our reducer can reduce the test program to $\approx 10\%$ of its original size. Other tools like C-

Reduce [39] can be applied to further reduce the size.

V. Evaluation

In this section, we evaluate `ZendDiff` by answering the following research questions:

- **RQ 1: Effectiveness on logic bug detection.** How effectively can `ZendDiff` uncover logic bugs in the PHP engine via the proposed test oracle? Can `ZendDiff` discover logic bugs in each execution mode? (Section V-A)
- **RQ 2: Comparison with existing approaches.** Relative to prior baselines, how does `ZendDiff` improve the effectiveness on logic bug detection? (Section V-B)
- **RQ 3: Ablation study**. To what extent do (i) state probing, (ii) JIT-aware program mutation, and (iii) dual verification contribute to `ZendDiff`'s effectiveness? (Section V-C)

PHP Settings. All experiments are conducted on the commit (*ce51bfac759dedac1537f4d5666dcd33fbc4a281*) following the PHP interpreter version (*v8.3.8*) available at the time of this work. We compiled the PHP interpreter using clang with debug symbols (sanitizers disabled). Detailed component options are listed in our artifacts.

Evaluation Metrics. We evaluate and compare `ZendDiff` with prior work using three well-defined metrics: (i) *Number of logic bugs*. The logic bugs we detected manifest as discrepancies between non-JIT and JIT execution results. To ensure accuracy, we employ a deduplication process, where multiple bugs identified with the same result are treated as duplicates and counted only once. This metric serves as the primary indicator of the detection capability and the practicality of the testing techniques. (ii) *Code coverage*. Code coverage is a widely adopted metric in evaluating fuzzing and testing approaches [40, 41], as it provides a clear and quantifiable way to evaluate which parts of the code have been executed during tests. In particular, we report line code coverage using gcovr tool [42], following the recommended configuration from the official PHP Makefile. (iii) *Zend opcodes diversity*. This metric emphasizes the ability to explore a wide range of opcode executions. Similar to query plans in database system testing, a broader exploration of Zend opcodes provides deeper insights into the interpreter's behavior and increases the likelihood of discovering potential vulnerabilities.

Experimental Infrastructure. All experiments were conducted on an AMD EPYC 7763 server (64 physical / 128 logical cores at 2.45 GHz, 512 GB RAM) running Ubuntu 22.04. By default, `ZendDiff` is designed to utilize a moderate amount of computational resources, allocating 32 CPU cores and up to 32 GB of RAM, which enables it to detect various logic bugs within a 24-hour timeframe.

A. Discovering Previously Unknown Logic Bugs

To uncover previously unknown logic bugs, we intermittently tested the latest PHP interpreter built from the official repository [43]. This testing was carried out over a four-month period, following well-established evaluation methodologies for automated bug-detection tools [41, 44]. To streamline the analysis of complex test cases and facilitate pinpointing their root causes, we employed delta debugging [38] to reduce each case to its minimal and bug-inducing form. Furthermore, we cross-checked the reduced test cases with existing issue trackers to prevent the redundant submission of bug reports.

Results. Table I summarizes all confirmed or fixed logic bugs detected by `ZendDiff` and verified through manual analysis. The *Engine* column indicates the affected component: ◐ marks JIT-only bugs, ◑ denotes non-JIT bugs, and ● represents bugs impacting both (as confirmed by the official PHP core team). This demonstrates that differential testing enables the JIT and non-JIT subsystems to expose each other's defects. The *Bug Location* column lists the source file in which each bug was detected. The *Issue ID* column gives the official issue tracking number for each bug. The *Status* column indicates whether each bug has been fixed (**`Fx`**) or confirmed and awaiting a patch (**`Cf`**). The *Fixes* column reports patch size (lines added and removed), and the *Description* column provides a concise explanation of each bug.

In total, `ZendDiff` identified 51 previously unknown logic bugs caused by inconsistencies between PHP's JIT and non-JIT execution modes. Out of these, the majority of bugs (37) have already been fixed, 3 have been confirmed, 5 are marked as duplicates, and 6 are categorized as expected. Specifically, among the confirmed and fixed bugs, our testing revealed 34 bugs localized to the JIT compiler, 3 bugs in the non-JIT interpreter, and 3 bugs that impact both components, underscoring the breadth of `ZendDiff` 's differential-testing capability. The *Bug Location* column in Table I further provides a detailed breakdown of the specific source code files where we found many of them are closely related to Zend engine or JIT compilation. Totally, the identified bugs are distributed across around 30 distinct files. To address these bugs, developers introduced modifications spanning a total of 1,958 lines of code (1,802 additions and 156 deletions).

Case Study. The remainder of this section highlights representative bugs, analyzing their root causes and impacts. Each case incorporates our observations and feedback from the official PHP development team, offering a comprehensive understanding of these issues.

Unexpected Null Value in JIT Mode. This Bug (ID: 21) pertains to an inconsistency in PHP's session management, specifically involving the `session_set_cookie_params()` function. In JIT mode, this function unexpectedly returns a `NULL` value, instead of the expected boolean value, as illustrated in Listing 3. This issue is traced to a misplaced conditional check that precedes the Zend Parse Parameters (ZPP) validation and an incorrectly placed `return` statement. In non-JIT mode, this misalignment leads to a fatal error when the function's return type does not conform to the expected boolean type. However, in JIT mode, the function silently returns `NULL`, resulting in inconsistent behavior and potentially leading to unexpected outcomes in dependent operations. The patch for this bug ensures that parameter validation occurs in the correct order and explicitly returns a boolean value, with a warning introduced to notify developers of incorrect behavior.

PHP Reflection Bug in Non-JIT Mode. This bug (ID: 07) oc-

TABLE I: Confirmed and Fixed Logic Bugs Discovered by `ZendDiff`

ID	Engine	Bug Location	Issue ID	Status	Fixes	Description
01	◐	zend_jit.c, ...	15652	**Fx**	+51 -5	Incorrect handling of dynamic property checks
02	◐	ZendAccelerator.c	15657	**Fx**	+35 -0	Conflict between memory protection and JIT compilation
03	◐	zend_jit.c	15658	**Fx**	+21 -0	Missing implementation of `ZEND_MATCH` VM handler
04	◐	ir_cfg.c	15662	**Fx**	+2 -1	Incorrect handling of an infinite loop with ENTRY instructions
05	◐	zend_jit.c	15820	**Fx**	+1 -1	JIT misconfiguration caused by missing implementation
06	◐	zend_inference.c	15821	**Fx**	+21 -1	Incorrect type inference for `ZEND_FRAMELESS_ICALL_N`
07	◑	php_reflection.c	15902	**Fx**	+126 -5	Improper overwriting of constant values
08	◐	ir_cfg.c	15903	**Fx**	+34 -0	Missing handling of `IR_BB_LOOP_WITH_ENTRY` flag
09	◐	ir_cfg.c	15909	**Fx**	+17 -1	Improper processing in IR CFG construction
10	●	zend_stack.c	15496	**Cf**	-	Improper output buffering
11	◐	zend_jit_ir.c, ...	15972	**Fx**	+42 -1	Improper arguments passing
12	◐	zend_jit_x86.dasc, ...	15973	**Fx**	+30 -2	Missing condition check when compiling `op_array`
13	◐	zend_execute.c	15981	**Fx**	+34 -5	Incorrect complex handler call in minimal JIT
14	◑	array.c	15982	**Fx**	+14 -1	Missing dereference operation in `ZVAL_COPY`
15	◐	zend_jit_ir.c	16009	**Fx**	+46 -6	Incorrect updating of operand variables
16	◐	zend_jit.c, ...	16186	**Fx**	+95 -0	Improper handling of the scope of `op_arrays`
17	◐	zend_execute.c	16321	**Cf**	-	Improper resource freeing after destructor
18	◐	ir_gcm.c	16355	**Fx**	+19 -0	Incorrect handling of invariant instructions in GCM
19	●	inner_outer_html_mixin.c	16356	**Fx**	+142 -0	Incorrect serializing of sibling nodes of `$outerHTML`
20	◐	zend_jit.c	16358	**Fx**	+341 -39	Missing handling of static method call
21	◐	session.c	16385	**Fx**	+22 -4	Missing implementation of debug arginfo checks
22	◐	fiber.c	16388	**Fx**	+13 -0	Incorrect cloning of instances of test class
23	◐	zend_jit.c	16393	**Fx**	+28 -5	Incorrect handling of JIT triggers
24	◐	ffi.c	16397	**Fx**	+26 -1	Missing initialization of comparison handler
25	◐	pass1.c	16408	**Fx**	+25 -3	Incorrect handling of runtime warnings during optimization
26	◑	php_dom.stub.php	16473	**Fx**	+16 -3	Incorrect implementation of `dom_import_simplexml`
27	●	zend_compile.c	16509	**Fx**	+22 -3	Incorrect line number information of `lineno`
28	◐	zend_jit_ir.c	16572	**Fx**	+26 -3	Missing argument validation when jumping to a basic block
29	◑	zend_vm_execute.h	16574	**Fx**	+28 -13	Incorrect changing of object pointer by `get_method` handler
30	◐	zend_jit_trace.c	16770	**Fx**	+42 -0	Incorrect JIT tracing type inferring
31	◐	zend_jit.c, ...	16879	**Fx**	+50 -1	Incorrect JIT dead code handling
32	◐	zend_optimizer.c	17106	**Fx**	+24 -0	Incorrect Zend `ZEND_MATCH_ERROR` optimization
33	◐	zend_jit_trace.c	17140	**Fx**	+34 -1	Missing handler for `ZEND_FETCH_DIM_FUNC_ARG`
34	◐	zend_inference.c	17144	**Cf**	-	Incorrect type inference of `ZEND_FETCH_DIM_W`
35	◐	ir_gcm.c	17190	**Fx**	+47 -0	Incorrect JIT IR handling
36	◐	sccp.c	17246	**Fx**	+50 -2	Unexpected nested SHM protections
37	◐	zend_jit_vm_helpers.c, ...	17257	**Fx**	+47 -4	Outdated opcode when IP is not stored
38	◐	zend_jit_ir.c	17428	**Fx**	+38 -3	Incorrect update on `ZEND_DO_FCALL` call level
39	◐	ir_fold.h	17430	**Fx**	+25 -25	Unrobust folding of the index and ZVAL size
40	◐	zend_jit_trace.c, ...	18262	**Fx**	+168 -17	Unrobust type guard of JIT loop

Listing 3: Unexpected Null Value in JIT execution

```
<?php
var_dump(session_set_cookie_params(3600, "/foo"));
 // Non-JIT: Fatal error .. Return value must be of
    type bool ..
 // JIT: NULL (Unexpected)
```

Listing 4: Unexpected Assertion in Non-JIT Execution

```
<?php
  class C { public stdClass $a = FOO; }
  $reflector = new ReflectionClass(C::class);
  $c = $reflector->newLazyGhost(function () { });
  function f() { define('FOO', new stdClass);} f();
  try { var_dump($c->a); } catch (\Error $e) {}
  $fusion = $reflector;
  $s = 'C:11:"Object":'.strlen($p).':{'.$fusion.'}';
Non-JIT: Assertion `zval_get_type(&(*(value))) ==
    11' failed (Unexpected)
JIT: object(stdClass)#4 (0) {}
```

curs within PHP's reflection functionality. It involves incorrect handling of class property default values, as demonstrated in Listing 4. Unlike typical cases where non-JIT results serve as a reference to identify bugs in JIT executions, this bug occurs solely in non-JIT mode, highlighting `ZendDiff` 's capability to reveal bugs across both non-JIT and JIT execution modes.

Listing 5: Incorrect Error Message Line Numbers in both Non-JIT and JIT Executions

```
<?php
  include __DIR__ . '/test.inc';
  include __DIR__ . '/test.inc';
<?php
  function test() {  // line 3
    echo 'foo'; // line 5
  }
Non-JIT: Fatal error: Cannot redeclare function
    test() .. in test.inc on line 4 (Incorrect)
JIT: Fatal error: Cannot redeclare function test()
    .. in test.inc on line 8 (Incorrect)
```

Following this discovery, the development team determined that the critical issue stemmed from an incorrect implementation of constant property initializers within the interpreter's reflection functionality.

Incorrect Results in both JIT and Non-JIT Modes. In the previous two test cases, we demonstrated `ZendDiff`'s capability to identify bugs in either non-JIT or JIT execution modes. In this case, we analyze a bug (ID: 27) that results in incorrect and inconsistent outputs across both modes. A critical aspect of programming in PHP is the accurate generation of warning messages, which are essential in diagnosing potential issues within the source code by indicating the precise line number

Listing 6: Unexpected JIT Error Missed by FlowFusion

```
<?php
  function dumpType(ReflectionType $rt) {
    var_dump($rt::class); dumpType(null); }
  function test1(): int { }
  dumpType((new ReflectionFunction('test1'))->
    getReturnType());
Non-JIT result: Fatal error: .. Argument #1 ($rt)
    must be of type ReflectionType, null given
JIT result: Fatal error: Uncaught TypeError: Cannot
    use ``::class'' on null (unexpected)
```

associated with the root cause. However, inaccuracies in the reported line number can significantly diminish the diagnostic value of these warnings. As illustrated in Listing 5, this bug leads to erroneous warning line numbers being produced by both non-JIT and JIT execution modes. The misalignment of line numbers not only introduces inconsistencies but also hampers developers' ability to effectively trace and rectify source code issues. This bug was identified through our approach and acknowledged by the development team. They traced the root cause to an incorrect usage of the line number from the first instruction in a function, rather than from the actual start of the function itself.

B. *Improved Effectiveness of* `ZendDiff`

We first evaluate `ZendDiff`'s improved effectiveness through a comparison with FlowFusion [7], the state-of-the-art PHP-specific fuzzer. Our results show that `ZendDiff` is capable of uncovering previously undetected logic bugs that FlowFusion fails to detect. Then, to further assess the enhanced capability of `ZendDiff` 's test oracle, we compare its results against the official PHP test suite.

Comparison with FlowFusion. In principle, `ZendDiff` provides a complementary oracle for logic bugs compared with the memory/crash oracle typically used in fuzzers. For instance, as the FlowFusion fuzzer relies on memory errors as the bug detection oracle, it does not deal with logic bugs. Evaluations show `ZendDiff` and FlowFusion achieve comparable code coverage. However, coverage does not directly translate into bug-finding effectiveness. Without a robust oracle for logic bugs, FlowFusion does not detect the bugs found here.

We illustrate with two representative logic bugs discovered by `ZendDiff` (one in non-JIT mode and another in JIT mode) where the PHP engine inconsistently reports diagnostic messages, which are essential for effective debugging and developer productivity. However, conventional fuzzers often overlook issues in such diagnostic messages, as it is difficult to assess their correctness using crash-based fuzzing oracles.

The first bug (ID: 28) involves an unexpected discrepancy in the behavior of the dumpType function under JIT execution, as shown in Listing 6. This issue stems from a logic flaw when handling recursive calls with invalid arguments. Specifically, the dumpType function is designed to accept a ReflectionType argument. However, a type mismatch arises when null is passed in a recursive call, triggering an argument validation error due to the unexpected type.

Listing 7 presents another bug (ID: 29), discovered by

Listing 7: Unexpected non-JIT Error Missed by FlowFusion

```
<?php
  $i = new ArrayIterator(array(1,1,1,1,1));
  $i = new CachingIterator($i,CachingIterator::
      FULL_CACHE);
  $fusion = $i; $fusion->doesnotexist("x");
Non-JIT: Fatal error: Uncaught Error: Call to
    undefined method ArrayIterator::load()
    (Unexpected)
JIT: Fatal error: Uncaught Error: Call to undefined
    method CachingIterator::load()
```

`ZendDiff` in non-JIT execution mode. This bug manifests as incorrect diagnostic output for an "undefined method" error. In this case, the error message erroneously references a method associated with the ArrayIterator class, despite the variable having already been updated to reference a new class, CachingIterator. The root cause is that the non-JIT interpreter fails to correctly reflect the variable update in its error reporting, misleading developers during debugging.

These examples highlight `ZendDiff` 's ability to detect subtle, non-crashing logic bugs that are easily missed by existing fuzzers like FlowFusion.

Comparison with the Official PHP Test Suite. As described in Section II, the official test suite includes a substantial collection of well-maintained test cases with expected outputs, comprising over 18,000 distinct cases. We evaluated the improved effectiveness of our differential testing approach based on two key criteria: (i) covering a greater number of unique opcodes, and (ii) achieving higher code coverage.

More unique DynASM opcodes covered. DynASM [45] is a dynamic assembler used in the PHP interpreter for JIT compilation or code generation on the fly. Counting the unique patterns of DynASM opcodes that have been executed can reflect the coverage of fuzz testing approaches. We use the configuration `opcache.opt_debug_level` as suggested on the official page[37]. We set this value to `0x10000` to dump opcodes as the compiler produced them before any optimization in non-JIT executions and to `0x20000` output optimized codes in JIT executions. We patch the official tests and `ZendDiff` by adding this specific configuration and collect unique opcode patterns within 24 hours.

We analyze the unique single DynASM opcode operands, as well as the unique pairs of adjacent DynASM opcodes, to highlight the improved effectiveness. DynASM opcode pairs represent all unique combinations of two consecutive opcodes. A higher count of executed operands or pairs indicates that an approach triggers more diverse and potentially interesting behaviors of the PHP interpreter.

Figure 3 illustrates the comparison of executed DynASM opcodes between the official test suite and `ZendDiff`. We observe that `ZendDiff` surpasses the official test suite in each setting after 24 hours of testing, despite initially lagging in the first few hours. In terms of consecutive opcodes, `ZendDiff` can have a higher improvement up to 25% in 24 hours. The increased execution of DynASM opcodes in `ZendDiff` is due to two factors: (i) `ZendDiff` preserves the diversity of generated PHP test programs, and (ii) `ZendDiff` instruments

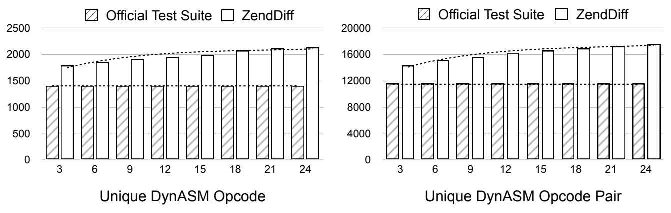

Fig. 3: DynASM Opcodes or Pairs Coverage Comparison

these programs to dump states for differential cross-checks, introducing additional operations in DynASM opcodes.

Higher line code coverage. Code coverage is a widely used metric to evaluate the effectiveness of testing approach. We evaluate ZendDiff's line code coverage against the official test suite over 24 hours (as suggested in previous work [46]). Figure 4 shows the coverage results. We notice that ZendDiff outperforms the official test suite over the 24 hours of testing. The official test suite remains almost unchanged due to its static nature with a limited number of test cases, while ZendDiff is capable of continuous testing with an increasing trend of line code coverage after 24 hours. Our code coverage can achieve even higher (*e.g.,* 82.0%) after 7 days' testing. For quick testing, the official test suite might have higher efficiency in detecting potential inconsistencies within minutes, but ZendDiff provides the capability of automatic and continuous testing of the PHP interpreter with higher efficiency in the long run.

C. Ablation Study of ZendDiff

We perform an ablation study to analyze the contribution of our three key strategies: JIT-aware program mutation, program state probing, and dual verification.

JIT-aware Program Mutation. We evaluate its effectiveness in increasing the intensity of JIT execution. Specifically, we use a set of 3,000 mutated test cases generated by FlowFusion and execute them under various JIT configurations: default tracing JIT, function-level JIT (*i.e.,* whole-script JIT), tracing JIT with a minimal hot function threshold, and tracing JIT with a minimal hot loop threshold.

To assess its impact, we propose two metrics: *JIT-aware memory usage* and the *probability of JIT trace initiation*, which can be observed by applying specific bitmasks in `opcache.jit_debug` configuration. JIT memory usage serves as an indirect indicator of JIT activity—higher memory consumption generally reflects more frequent or extensive JIT compilation. The trace initiation probability is computed by analyzing execution logs for the presence of the `"TRACE 1 start"` marker, which indicates the start of a tracing JIT compilation.

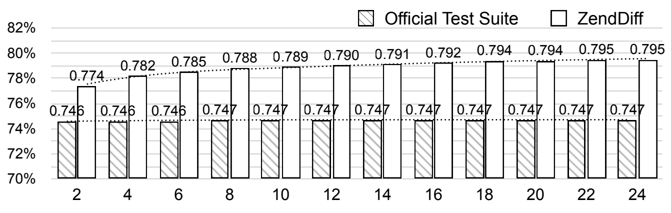

Fig. 4: Code Coverage Improvement of ZendDiff

TABLE II: JIT-Aware Program Mutation Analysis

	JIT Memory Usage (MB)	JIT Tracing (%)
Non-JIT	N/A	N/A
Tracing JIT	2403.72	0.06
Function JIT	19667.04	0
Hot Function	5893.90	0.96
Hot Loop	5538.27	0.93

Results in Table II show that mutations targeting hot functions and hot loops significantly increase the probability of triggering JIT tracing. These cases often involve more complex logic, which is more likely to contain subtle bugs. Compared to the default tracing JIT, both hot function and hot loop configurations exhibit higher JIT memory usage, indicating more intensive JIT activity. Although function JIT, which compiles the entire script at once, tends to consume more memory overall, tracing JIT performs on-the-fly optimizations, resulting in more sophisticated and dynamic JIT executions.

Program State Probing. To evaluate the effectiveness of program state probing, we run ZendDiff with and without this feature enabled, measuring the number of bug alarms triggered after executing a fixed number of test cases. Using randomly generated PHP inputs, both configurations are tested under identical conditions, 5,000 cases per hour for 24 hours, to simulate real-world usage. Dual verification is enabled throughout the experiment to filter out unstable results. The results show that ZendDiff without state probing reports many fewer bug alarms. In contrast, enabling program state probing improves detection efficiency, uncovering over 50% more bug alarms on average with the same number of test cases. This demonstrates that state probing increases the likelihood of detecting logic bugs in the PHP interpreter.

Dual Verification. The dual verification strategy aims to reduce false alarms. To evaluate its effectiveness, we compare three configurations: (i) $\texttt{ZendDiff}_1$ with a naive differential oracle (no verification), (ii) $\texttt{ZendDiff}_2$ with dual verification, and (iii) $\texttt{ZendDiff}_3$ with triple verification, which adds a third execution for consistency checks. We run each setting for 24 hours to compare false positive rates and overall efficiency. This evaluation leverages test case preprocessing and program state probing (Section III) to mitigate nondeterminism and capture fine-grained execution states.

Figure 5 compares three configurations: differential test oracle (I), dual verification (II), and triple verification (III), showing the total number of processed test cases and the distribution of passed (functional) versus failed (buggy) cases. Naive $\texttt{ZendDiff}_1$ (I) classifies around 90% of cases as bugs, but manual inspection reveals most are false positives due to unstable executions. Dual verification in $\texttt{ZendDiff}_2$ (II) effectively filters out these unstable cases, significantly reducing false alarms. Triple verification $\texttt{ZendDiff}_3$ (III) yields similar results to $\texttt{ZendDiff}_2$, with only 9 additional unstable cases filtered. Over a 24-hour run, $\texttt{ZendDiff}_2$ processes the most test cases (237,908), compared to $\texttt{ZendDiff}_1$

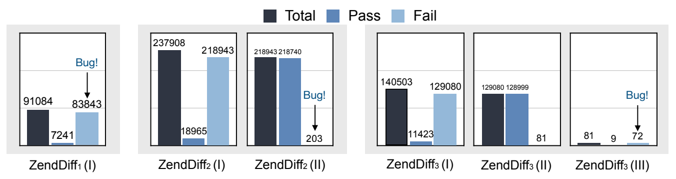

Fig. 5: Effectiveness and Efficiency Comparison Among `ZendDiff`$_1$, `ZendDiff`$_2$, and `ZendDiff`$_3$

(91,084) and `ZendDiff`$_3$ (140,503), demonstrating its better efficiency. Since `ZendDiff` logs all detected bugs along with reproducing inputs and outputs, the high false positive rate in `ZendDiff`$_1$ leads to time-consuming logging (83,843 cases), while `ZendDiff`$_2$ and `ZendDiff`$_3$ store only 203 and 72 cases. Although `ZendDiff`$_3$ improves stability, its extra verification step reduces throughput. Overall, dual verification achieves the best balance of accuracy and efficiency.

We analyze the distribution of bug results (particularly false positives) with and without dual verification. False positives primarily fall into four categories: (i) `FileFP`: File-related functions like `fileinode()` return varying results due to dynamic file system behavior. (ii) `DynamicFP`: Dynamic identifiers such as PIDs, ports, session IDs, and memory addresses differ across runs. (iii) `TimeFP`: Timestamps vary due to external timing factors. (iv) `RandomFP`: Randomized functions like `rand()` or `shuffle()` produce non-deterministic outputs.

`ZendDiff` also detects expected differences tied to JIT mode, which are consistent within the same mode and pass our verification: (i) `JITCfgDiff`: Differences in printed JIT configuration due to varying runtime environments. (ii) `GCMemDiff`: Variations in garbage collection memory behavior between non-JIT and JIT. (iii) `ObjRefNumDiff`: Differences in object reference numbering across execution modes.

The use of dual verification in `ZendDiff` significantly improves the identification of actual bugs while reducing false positives, as shown in Figure 6. The pie charts demonstrate that under differential oracle, false positives constitute a large portion (*e.g.,* `FileFP` at 56.6%), while the proportion of detected bugs is low at 0.4%. However, with the implementation of dual verification, the proportion of bugs detected rises to 63.9%, effectively reducing false positives across categories like `FileFP`, `DynamicFP`, and `TimeFP`. These results show that dual verification enhances the accuracy and efficiency of logic bug detection.

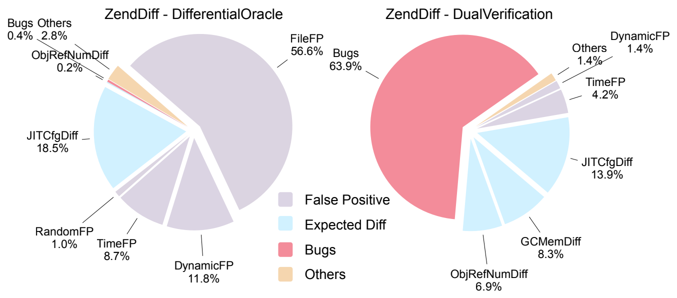

Fig. 6: Bug Result Distribution of `ZendDiff` with (Right) and without (Left) Dual Verification

VI. Related Work

Differential Testing. Differential testing is a well-established approach for detecting semantic discrepancies and vulnerabilities in various software. For instance, Csmith [19] is a random C program generator designed to stress-test C compilers via differential testing while avoiding undefined behavior. For Java Virtual Machine (JVM) implementations, Classfuzz [22] and Classming [23] focus on the JVM startup process and comprehensive JVM testing, respectively. Rust-Smith [24] performs differential testing between Rust compilers and across optimization levels to identify potential bugs. JEST [18] compares JavaScript engine behaviors against the ECMAScript specification through automatically synthesized conformance tests with injected assertions. Frankencerts [20] and Mucerts [21] utilize behavioral differences between test programs to optimize input generation and detect bugs in SSL/TLS-related systems. R2Z2 [25] extends differential testing to web browsers by detecting rendering inconsistencies across versions, while Diffuzz [26] applies differential fuzzing to side-channel analysis by examining resource consumption variations under different secret inputs.

Differential Testing via Compilation Space Exploration. This line of work treats the discrepancies between different compilation optimizations as potential logic bugs. For example, EMI [32] generates input-equivalent program variants by pruning unexecuted code to differentially test compiler optimizations. Artemis [31] strategically mutates test programs with JIT-relevant, yet semantics-preserving code structures to trigger diverse JIT compilation choices of JAVA Virtual Machines (JVMs). JIT-Picking [17], FuzzJIT [47], and Dumpling [34] focus on identifying such optimization bugs in JavaScript engines by comparing the behavior of JS interpreters and JIT compilers. Similarly, our approach targets logic bugs in the PHP engine via detecting discrepancies of non-JIT and JIT executions. Consistent with this line of work that typically distinguishes itself by tackling new domains with tailored oracles which show practical effectiveness, our work likewise focuses on making differential testing practical for PHP through three key techniques (*i.e.,* program state probing, JIT-aware program mutation, and dual verification).

Bug Detection in the PHP Engine. Existing approaches mainly leverage fuzzing techniques to detect bugs in the PHP Engine. For instance, LangFuzz [3], NAUTILUS [4], Gramatron [5], and PolyGlot [6] focus on memory errors in the PHP interpreter. Reflecta [8] gains a diverse set of language features dynamically to reduce manual effort. FlowFusion [7] merges code semantics from two or more seed programs to generate new fuzzing inputs and effectively finds hundreds of memory errors in the PHP interpreter. However, these approaches primarily rely on crashes or sanitizer alerts to detect bugs, which differs from our focus, logic bugs, that do not cause crashes or sanitizer alerts. To fill this gap, we develop `ZendDiff` to detect logic bugs in the PHP interpreter through differential testing of JIT and non-JIT execution modes.

Bug Discovery in Other Compilers or Interpreters. Existing solutions for detecting bugs in other compilers and interpreters mainly concentrate on fuzzing techniques, such as C/C++[48, 19], Rust[24, 49], and JavaScript (JS) [50, 47, 17], to mitigate potential cascading security issues. For instance, GrayC [48] is a greybox, coverage-directed, mutation-based approach to fuzz C compilers and code analyzers by employing a new set of mutations on common C constructs. Comfort [51] leverages a deep learning-based language model to automatically generate JS test code to detect bugs in JS engines. Fuzzilli [50] presents the design and implementation of an intermediate representation (IR) aimed at uncovering vulnerabilities in JIT compilers. FuzzJIT [47] focuses on identifying JIT compiler bugs by triggering the JIT compilation process and capturing execution inconsistencies.

VII. Conclusion

In this paper, we introduced `ZendDiff`, a novel differential testing framework that leverages JIT and non-JIT execution modes to detect logic bugs in the PHP interpreter. Our approach integrates program state probing for capturing fine-grained execution state, JIT-aware program mutation for sufficiently exercising JIT functionality, and dual verification for handling non-deterministic behaviors. These techniques enable effective detection of logic bugs that silently lead to incorrect results. Our evaluation results demonstrated the superior efficacy of `ZendDiff` compared to existing testing approaches. Impressively, it discovered 51 previously unknown logic bugs in the PHP interpreter, with 37 fixed and 3 confirmed by developers, while achieving higher code coverage and executing more Zend opcodes than the official test suite. `ZendDiff` has been acknowledged by the PHP development team and provides a practical tool to automatically discover logic bugs in the PHP interpreter.

Acknowledgements

We thank the anonymous reviewers for their constructive suggestions and the PHP developers for promptly addressing our bug reports and sharing valuable feedback.

References

[1] H. Kiran, https://techjury.net/blog/php-usage-statistics/.

[2] w3techs, https://w3techs.com/technologies/details/pl-php.

[3] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 445–458.

[4] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars." in *NDSS*, 2019.

[5] P. Srivastava and M. Payer, "Gramatron: Effective grammar-aware fuzzing," in *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis*, 2021, pp. 244–256.

[6] Y. Chen, R. Zhong, H. Hu, H. Zhang, Y. Yang, D. Wu, and W. Lee, "One engine to fuzz'em all: Generic language processor testing with semantic validation," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 642–658.

[7] Y. Jiang, C. Zhang, B. Ruan, J. Liu, M. Rigger, R. H. Yap, and Z. Liang, "Fuzzing the PHP interpreter via dataflow fusion," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 6143–6158.

[8] C. Zhang, G. Lee, Q. Liu, and M. Payer, "Reflecta: Reflection-based scalable and semantic scripting language fuzzing," in *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, 2025, p. 1772–1787.

[9] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases," Department of Computer Science, Hong Kong, Tech. Rep. HKUST-CS98-01, 1998.

[10] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou, "Metamorphic testing: A review of challenges and opportunities," *ACM Comput. Surv.*, vol. 51, no. 1, 2018.

[11] T. Y. Chen, J. W. Ho, H. Liu, and X. Xie, "An innovative approach for testing bioinformatics programs using metamorphic testing," *BMC bioinformatics*, vol. 10, no. 1, pp. 1–12, 2009.

[12] A. Ramanathan, C. A. Steed, and L. L. Pullum, "Verification of compartmental epidemiological models using metamorphic testing, model checking and visual analytics," in *2012 ASE/IEEE International Conference on BioMedical Computing (BioMedCom)*, 2012, pp. 68–73.

[13] W. K. Chan, S. C. Cheung, and K. R. Leung, "A metamorphic testing approach for online testing of service-oriented software applications," *International Journal of Web Services Research (IJWSR)*, vol. 4, no. 2, pp. 61–81, 2007.

[14] F.-C. Kuo, T. Y. Chen, and W. K. Tam, "Testing embedded software by metamorphic testing: A wireless metering system case study," in *2011 IEEE 36th Conference on Local Computer Networks*, 2011, pp. 291–294.

[15] W. K. Chan, T. Y. Chen, H. Lu, T. H. Tse, and S. S. Yau, "Integration testing of context-sensitive middleware-based applications: a metamorphic approach," *International Journal of Software Engineering and Knowledge Engineering*, vol. 16, no. 05, pp. 677–703, 2006.

[16] M. N. Mansur, M. Christakis, and V. Wüstholz, "Metamorphic testing of datalog engines," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, p. 639–650.

[17] L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, and T. Holz, "JIT-Picking: Differential fuzzing of javascript engines," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 351–364.

[18] J. Park, S. An, D. Youn, G. Kim, and S. Ryu, "JEST: N+1 -version Differential Testing of Both JavaScript Engines and Specification." in *International Conference on Software Engineering (ICSE)*, 2021, pp. 13–24.

[19] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011, p. 283–294.

[20] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, "Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations," in *2014 IEEE Symposium on Security and Privacy*, 2014.

[21] Y. Chen and Z. Su, "Guided differential testing of certificate validation in SSL/TLS implementations," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[22] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of JVM implementations," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2016.

[23] Y. Chen, T. Su, and Z. Su, "Deep differential testing of JVM implementations." in *International Conference on Software Engineering (ICSE)*, 2019, pp. 1257–1268.

[24] M. Sharma, P. Yu, and A. F. Donaldson, "RustSmith: Random differential compiler testing for Rust," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1483–1486.

[25] S. Song, J. Hur, S. Kim, P. Rogers, and B. Lee, "R2z2 - detecting rendering regressions in web browsers through differential fuzz testing," in *Proceedings of the 44th International Conference on Software Engineering*. ACM, 2022, pp. 1818–1829.

[26] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, "Diffuzz: Differential Fuzzing for Side-Channel Analysis." in *Software Engineering (SE)*, 2020, pp. 125–126.

[27] Facebook, "HHVM: A virtual machine for executing programs written in hack." https://github.com/facebook/hhvm, 2025.

[28] F. Emmott, "Ending php support, and the future of hack," https://hhvm.com/blog/2018/09/12/end-of-php-support-future-of-hack.html, 2018.

[29] "PHP 8.0 JIT," https://php.watch/versions/8.0/JIT, 2020.

[30] "PHP JIT in depth," https://php.watch/articles/jit-in-depth, 2020.

[31] C. Li, Y. Jiang, C. Xu, and Z. Su, "Validating jit compilers via compilation space exploration," *ACM Trans. Comput. Syst.*, vol. 43, no. 3, Jul. 2025. [Online]. Available: https://doi.org/10.1145/3715102

[32] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 216–226. [Online]. Available: https://doi.org/10.1145/2594291.2594334

[33] S. Zhou, M. Jiang, W. Chen, H. Zhou, H. Wang, and X. Luo, "Wadiff: A differential testing framework for webassembly runtimes," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '23. IEEE Press, 2024, p. 939–950. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00188

[34] L. Wachter, J. Gremminger, C. Wressnegger, M. Payer, and F. Toffalini, "Dumpling: Fine-grained differential javascript engine fuzzing," in *NDSS*, 2025.

[35] P. Li and M. Zhang, "Fuzzcache: Optimizing web application fuzzing through software-based data cache," 2024.

[36] "PHPT structure details," https://qa.php.net/phpt_details.php, 2020.

[37] "PHP opcache runtime configuration," https://www.php.net/manual/en/opcache.configuration.php, 2025.

[38] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[39] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012, pp. 335–346.

[40] M. Rigger and Z. Su, "Finding bugs in database systems via query partitioning," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi-org.libproxy1.nus.edu.sg/10.1145/3428279

[41] ——, "Testing database engines via pivoted query synthesis," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, 2020.

[42] gcovr, https://gcovr.com/, 2024.

[43] "The official code repository of the PHP interpreter," https://github.com/php/php-src/, 2025.

[44] M. Kamm, M. Rigger, C. Zhang, and Z. Su, "Testing graph database engines via query partitioning," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, p. 140–149.

[45] "Dynasm," https://luajit.org/dynasm.html, 2025. [Online]. Available: https://luajit.org/dynasm.html

[46] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, p. 2123–2138.

[47] J. Wang, Z. Zhang, S. Liu, X. Du, and J. Chen, "FuzzJIT: Oracle-enhanced fuzzing for JavaScript engine JIT compiler," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1865–1882.

[48] K. Even-Mendoza, A. Sharma, A. F. Donaldson, and C. Cadar, "grayc: Greybox fuzzing of compilers and analysers for C," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1219–1231.

[49] F. Tuong, M. Omidvar Tehrani, M. Gaboardi, and S. Y. Ko, "Symrustc: A hybrid fuzzer for rust," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1515–1518.

[50] S. Groß, S. Koch, L. Bernhard, T. Holz, and M. Johns, "Fuzzilli: Fuzzing for javascript jit compiler vulnerabilities." in *NDSS*, 2023.

[51] G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, X. Sun, L. Bian, H. Wang, and Z. Wang, "Automated conformance testing for javascript engines via deep compiler fuzzing," in *Proceedings of the 42nd ACM SIGPLAN international conference on programming language design and implementation*, 2021, pp. 435–450.