Fuzzing the PHP Interpreter via Dataflow Fusion

Yuancheng Jiang, Chuqi Zhang, Bonan Ruan, Jiahao Liu, Manuel Rigger, Roland Yap, and Zhenkai Liang

National University of Singapore

{yuancheng, chuqiz, r-bonan, jiahao99, rigger, ryap, liangzk}@comp.nus.edu.sg

Abstract

PHP, a dominant scripting language in web development, powers a vast range of websites, from personal blogs to major platforms. While existing research primarily focuses on PHP application-level security issues like code injection, memory errors within the PHP interpreter have been largely overlooked. These memory errors, prevalent due to the PHP interpreter's extensive C codebase, pose significant risks to the confidentiality, integrity, and availability of PHP servers. This paper introduces *FlowFusion*, the first automatic fuzzing framework to detect memory errors in the PHP interpreter. FlowFusion leverages dataflow as an efficient representation of test cases maintained by PHP developers, merging two or more test cases to produce fused test cases with more complex code semantics. Moreover, FlowFusion employs strategies such as test mutation, interface fuzzing, and environment crossover to increase bug finding. In our evaluation, FlowFusion found 158 unknown bugs in the PHP interpreter, with 125 fixed and 11 confirmed. Comparing FlowFusion against the official test suite and a naive test concatenation approach, FlowFusion can detect new bugs that these methods miss, while also achieving greater code coverage. FlowFusion also outperformed state-of-the-art fuzzers AFL++ and Polyglot, covering 24% more lines of code after 24 hours of fuzzing. FlowFusion has gained wide recognition among PHP developers and is now integrated into the official PHP toolchain.

1 Introduction

PHP is a scripting programming language that is tailored for web development. Known for its flexibility and practicality, PHP powers a vast number of websites, ranging from personal blogs to global platforms. According to various reports [14, 24, 25, 52], PHP is used by over 70% of websites worldwide, making it one of the most popular programming languages for web deployment.

Although several PHP implementations are available, the official PHP interpreter is the most widely used. Its codebase, mainly in C, exceeds one million lines. As C is a low-level language without memory safety, the PHP interpreter is potentially vulnerable to memory errors that attackers may exploit.

Identifying memory errors in the PHP interpreter is challenging. Existing research mainly focuses on script security issues, such as application bugs, including SQL injection and file inclusion. Limited effort has been dedicated to detecting memory errors in the underlying PHP interpreter. To the best of our knowledge, no existing approach is specifically tailored for fuzzing the PHP interpreter. Some fuzzing approaches [2, 8, 18, 48] rely on grammar-guided program generation and have found memory errors in the PHP interpreter. Nevertheless, they may not be sufficiently effective in finding such bugs because *grammar generation focuses more on ensuring syntactic correctness than code semantics*. As a result, it struggles to create programs with complex semantic behavior and is unlikely to produce test cases targeting specific modules of the PHP interpreter.

Previous wisdom in fuzzing research has proven its practicality in uncovering memory errors [5, 17]. In terms of the PHP interpreter, the PHP community is maintaining a high-quality test suite consisting of over 19K test cases. Such test cases cover broad PHP features over 80 modules, such as in-memory databases or sessions, coming with plug-and-play running environments and configurations.

Observation 1: The test cases from PHP's official test suite yield higher code coverage than existing fuzzing approaches, thereby naturally forming a "golden testbed" for fuzzing.

Despite the intuition to leverage PHP official test cases for fuzzing, there still remains a key challenge—*how to enrich code semantics of test cases to reveal memory errors?* Specifically, even though the test cases are well-maintained with valid syntax, they are limited to simple code semantics given their unit-test-like nature. Ideally, we can extend the official test suite to create a larger and more comprehensive test suite. One approach is to craft semantically enriching code transformations, but doing so either remains inefficient if implemented as byte-level mutations or requires significant expertise and manual effort. To develop an automated fuzzing

Listing 1: 20 Year Old Memory Error Found by FlowFusion.

```
/* Test A */
/* Test A dataflow: $dom → $ref → $nodes */
  $dom = new DOMDocument;
  $dom->loadXML(..);
  $ref = $dom->documentElement->firstChild
  $nodes = $ref->childNodes;
  ----------------------------------------------------
/* Test B */
/* Test B dataflow: $values → $str → $enc */
  $values = array(..);
  foreach($values as $str)
    { $enc = base64_encode($str); }
  ----------------------------------------------------
/* Our Fused Test */
/* Dataflow fusion:$nodes → $fusion ← $values */
  $dom = new DOMDocument;
  $dom->loadXML(..);
  $ref = $dom->documentElement->firstChild
  $fusion = $ref->childNodes;
  $values = array(..);
  foreach($fusion as $str)
    { $enc = base64_encode($str); }
/* AddressSanitizer: heap-use-after-free */
```

strategy, we propose an alternative method of merging test cases. However, simply concatenating two existing tests is ineffective, as it yields the same outcome as running them independently. To extend existing test cases into more effective ones, the essential way is to capture and extend their code semantics. A standard approach is to extract the control flow and data flow of PHP programs in the official test suite.

Observation 2: Most official tests (96.1%) exhibit sequential control flow (*i.e.,* no branches)—the code semantics of such programs can be effectively represented by mere dataflow.

To address the challenge of generating semantics-rich test cases, we use *dataflow interleaving* to bridge the dataflow of two official tests, creating new code interactions (semantics) that were previously non-existent. For example, in Listing 1, test A verifies DOM objects and their entity references, and test B checks base64 encoding. However, the official test suite does not account for more complex code semantics, such as encoding DOM-related objects using base64, which might occur in real-world scenarios. To this end, we combine these cases, alongside their semantics, thereby generating new code functionalities. We first present their dataflows consisting of dataflow A ([`$dom`→`$ref`→`$nodes`]) and dataflow B ([`$values`→`$str`→`$enc`]). Then, we interleave their dataflows by connecting a variable from dataflow A (*e.g.,* `$nodes`) to a variable from dataflow B (*e.g.,* `$values`). The connected variables are fused with a new one named `$fusion`. As such, this real-world bug triggers an unknown 20-year-old use-after-free memory error (*i.e.,* it crashes PHP v5.0.0 released in 2004 and later versions) during the `foreach` iteration. By carefully crafting a malicious DOM, an adversary might exploit the memory vulnerability and compromise the host server supporting the PHP interpreter.

In this work, we present *FlowFusion*, a novel approach to automate the discovery of memory errors in the PHP interpreter with the sanitizer oracle.[1] FlowFusion generates fused tests[2] which contain enriched code semantics from dataflow interleaving. We term our dataflow-driven test merging approach as *dataflow fusion*. Additionally, FlowFusion employs several complementary strategies to more effectively uncover memory errors as follows.

- Test mutation. It mutates official test cases before their dataflows are interleaved. Test mutation is based on expression replacement, either replacing existing constants with special values, or replacing variables interchangeably.
- Interface fuzzing. It further makes use of more complex code semantics in fused test cases by calling random PHP functions with variables from fused test cases as arguments.
- Environment crossover. It merges the execution environments (*e.g.,* required modules, configurations) of fused test cases and further inserts random valid execution environment options collected from the official test suite.

We conducted experiments to assess the effectiveness of our approach. Remarkably, we detected 158 unique and previously unknown bugs in the PHP interpreter, of which 125 have been fixed and 11 confirmed. These bugs span 10 different *Common Weakness Enumerations* (CWEs) and affected over 80 individual source files in the PHP interpreter with the fixes changing over 5k lines of code.

We compared the effectiveness of FlowFusion against the official test cases and a simple concatenation approach of these test cases. The results demonstrate that FlowFusion not only detects more memory errors but also achieves higher code coverage. Additionally, we compared FlowFusion with state-of-the-art fuzzing techniques known for uncovering memory errors in the PHP interpreter. Specifically, FlowFusion outperformed AFL++ [12] and PolyGlot [8] by covering 24% more lines of code after 24 hours of fuzzing under the same execution conditions.

We performed an ablation study of our primary approach—dataflow fusion—alongside other strategies such as test mutation, interface fuzzing, and environment crossover. The results provide a deeper understanding of how each strategy contributes to our approach's overall effectiveness. FlowFusion is recognized by the PHP developers and has been integrated into the official toolchain.[3]

In summary, we make the following contributions:

- We propose a novel approach to discover memory errors in the PHP interpreter, called dataflow fusion. Based on the good quality and quantity of official test cases in the PHP

[1]We use Address Sanitizer [46] (*i.e.,* ASan) and Undefined Behavior Sanitizer (*i.e.,* UBSan). Most of the ASan and UBSan reports belong to memory errors except arithmetic overflow.

[2]Fused tests exhibit inherent diversity: combining pairs from the 19K official tests can produce over 300 million new tests.

[3]FlowFusion is available at https://github.com/php/flowfusion

interpreter, we expand the test suite with fused test cases in new code semantics by interleaving their dataflows.

- We implement our approach along with test mutation, interface fuzzing, and environment crossover as the first automatic fuzzing framework, FlowFusion, which effectively achieves comprehensive coverage in the PHP interpreter.
- FlowFusion has proven effective in discovering new bugs in the PHP interpreter. In total, we identified 158 unknown bugs, of which 125 have been fixed and 11 confirmed.

2 Background

In this paper, the PHP interpreter refers to the official implementation.[4] It comprises three primary components: The Zend engine (*i.e.,* the *zend* directory) is PHP's core execution engine, encompassing the bytecode compiler, runtime executor, and memory management routines. It transforms PHP scripts into opcodes and executes them. The core modules (*i.e.,* the *ext* directory) contain PHP's built-in and bundled extensions (*e.g.,* session, sqlite3), with each subfolder providing source code that extends the language beyond the core engine. The main functions (*i.e.,* the *main* directory) hold essential PHP runtime and infrastructure code that initializes and orchestrates the interpreter, including entry points, configuration loading, and the main execution loop.

Memory errors in the PHP interpreter. Statistics show that PHP ranks top 50 in the ranking for projects with the most vulnerabilities [9]. Given its complex codebase and written in the memory-unsafe language (*i.e.,* C), it is prone to memory-related errors such as buffer overflows and use-after-free. We conducted a study of public issues reported from 2022 to 2024 in the official PHP GitHub repository, up to August 2024. Out of 567 verified and closed issues, we identified 191 as memory errors, representing a significant proportion (33.7%) of all resolved bugs.

To better understand the root cause of these memory errors, we conducted a triage analysis for the 191 reported bugs. Due to the lack of issue normalization and diverse bug descriptions, we manually categorized these bugs into 10 categories based on the issue reports, conversations, and patches. The result indicates that *null dereference*, *memory leak or exhaustion*, and *buffer overflow or underflow* are the three most common memory errors in PHP interpreter, accounting for more than 10% each. In addition, *use-after-free* bugs also occur occasionally, reaching 7%. These bugs can lead to the aforesaid various security risks. In particular, *use-after-free* and *buffer overflow* ranked first and second respectively in the "2023 CWE Top 10 Known Exploited Vulnerabilities (KEV) Weaknesses" list [33], indicating their high exploitability and the urgency of detection.

The official PHP test suite. The PHP test suite has over 19K distinct test cases, encompassing a wide range of code semantics with valid syntax, which aid in verifying both the correctness and robustness of the PHP interpreter. These test cases cover over 80 unique modules in the PHP interpreter and integrate with all existing bug-triggering reproducers as additional security test cases. While the overall test suite has diverse code semantics, the individual test case only verifies the functionality or the correctness of a single component. Therefore, 96.1% of these test cases exhibit sequential control flow and execution without branches. Listing 2 shows one example test case, which verifies the correctness of pointer comparison in the Foreign Function Interface (FFI) component of the PHP interpreter.

Listing 2: Example Test Case in the Official Test Suite

```
--TEST--
  FFI 007: Pointer comparison
--EXTENSIONS--
  ffi
--INI--
  ffi.enable=1
--FILE--
  <?php
    $ffi = FFI::cdef();
    $v = $ffi->new("int*[3]");
    $v[0] = $ffi->new("int[1]", false);
    $v[1] = $ffi->new("int[1]", false);
    $v[2] = $v[1];
    $v[1][0] = 42;
    var_dump($v[0] == $v[1]);
    var_dump($v[1] == $v[2]);
    FFI::free($v[0]);
    FFI::free($v[1]);
  ?>
--EXPECT--
  bool(false)
  bool(true)
```

The official test cases are in a special format consisting of sections delimited by `--section--` and formatted in `.phpt` file. It contains over 30 sections[5] representing different meanings. Referring to Listing 2, we introduce some important sections as follows: (i) `--test--` section is a brief description of the test; (ii) `--extensions--` section details the extensions required for the test; (iii) `--ini--` section gives the specific configurations needed; (iv) `--file--` section contains the PHP program; and (v) `--expect--` section specifies the expected test results.

Dataflow of PHP programs. Dataflow analysis [23] involves tracking how data values propagate and are manipulated across different parts of a program. This includes identifying which variables are defined, used, or modified at different points in the code and understanding how these changes impact program behavior.

Given a node n in the control flow graph of the program, dataflow analysis considers the following four properties: the

[4] https://github.com/php/php-src

[5] https://qa.php.net/phpt_details.php

Gen Set ($\mathrm{GEN}(n)$) is the set of definitions generated (*i.e.,* initialized) at a particular node n; the Kill Set ($\mathrm{KILL}(n)$) is the set of definitions that are killed (*i.e.,* overwritten or invalidated) by the execution of a node n; the In Set ($\mathrm{IN}(n)$) represents the set of dataflow facts coming into of a node n; the Out Set ($\mathrm{OUT}(n)$) represents the set of dataflow facts going out of a node n.

In this work, we treat each statement in the test program as a node and denoted as n_i, where i indicates the sequence of statements. Given a program P with N statements, we consider the P's dataflow as the collection of In Sets and Out Sets from all N statements, as shown in Equation 1. Based on these sets of dataflow facts, one can infer the dataflow graph (we omit it for simplicity) of the program.

$$\mathrm{Flow}(P) = \{\mathrm{IN}(n_i), \mathrm{OUT}(n_i) \mid \forall i \in N\} \tag{1}$$

For calculating the In Set, considering the sequential control flow of test programs, only the preceding statement has the coming-in dataflow facts. In Equation 2, the In Set containing the dataflow facts coming in is represented as the empty set when the first statement, and propagated from going out dataflow facts of preceding statement otherwise.

$$\mathrm{IN}(n_i) = \emptyset \text{ if } i = 0 \text{ else } \mathrm{OUT}(n_{i-1}) \tag{2}$$

To calculate the Out Set, we consider the definitions that are newly generated or eliminated, as represented by the Gen Set and the Kill Set. For dataflow facts related to function calls, the conventional approach is to either perform dataflow analysis within the function or utilize function summaries. Our method assumes that return values have data dependencies on function arguments for better efficiency. We notate these dataflow facts using the Fun Set ($\mathrm{FUN}(n)$). Accordingly, we express the Out Set as shown in Equation 3. Specifically, the outgoing dataflow facts comprise elements from the incoming set, the generated set, and the function call set, while excluding those definitions that have been killed.

$$\mathrm{OUT}(n_i) = \mathrm{GEN}(n_i) \cup (\mathrm{IN}(n_i) \setminus \mathrm{KILL}(n_i)) \cup \mathrm{FUN}(n_i) \tag{3}$$

In this paper, we use the text notation ($[v_1] \rightarrow [v_2] \rightarrow \ldots [v_{m-1}] \rightarrow [v_m]$) to concisely represent dataflow from variable v_1 to v_m in PHP programs.

3 Threat Model

Memory errors pose inherent security risks and frequently lead to vulnerabilities in the PHP interpreter. According to vulnerability statistics for PHP, [44], more than 80% (179/220) of PHP CVEs in the past decade resulted from overflows or memory corruptions. Google OSS-Fuzz [45] project also considers PHP an interesting target for finding memory errors. Malicious users can attack PHP servers in the following ways:

(a) Malicious input interactions. Attackers may craft specific inputs to PHP applications, resulting in direct user data leakage. For instance, there are interfaces from officially disclosed vulnerabilities, one heap buffer over-read [36] in the *mysqlnd* PHP extension and another overflow [37] in `phar_dir_read()`. With such vulnerable interfaces, the attackers are allowed to craft SQL queries or *phar* files to retrieve data. Such memory errors are linked to user inputs and can be exploited in common web applications via malicious data.

(b) Malicious code injections. Another way for attackers is to exploit non-interactive memory errors by using custom PHP script execution. Such code injections are common in online PHP editors or coding sandboxes. In these environments, users can run arbitrary code yet are prevented from executing system commands, since dangerous functions are usually disabled on well-maintained servers. Although under such restricted conditions, memory errors offer an alternative attack surface to bypass security measures or mitigation [11,43]. Previous work, Polyglot [8], provides a detailed example of how a memory error can be exploited to escape PHP sandboxes.

4 Approach

In this section, we first present an overview of our approach in Section 4.1, outlining the multiple steps to detect memory errors in the PHP interpreter. Then, we provide a detailed explanation of our key method, dataflow fusion, in Section 4.2. Additionally, we describe other important strategies employed in this work to facilitate memory error detection in Section 4.3.

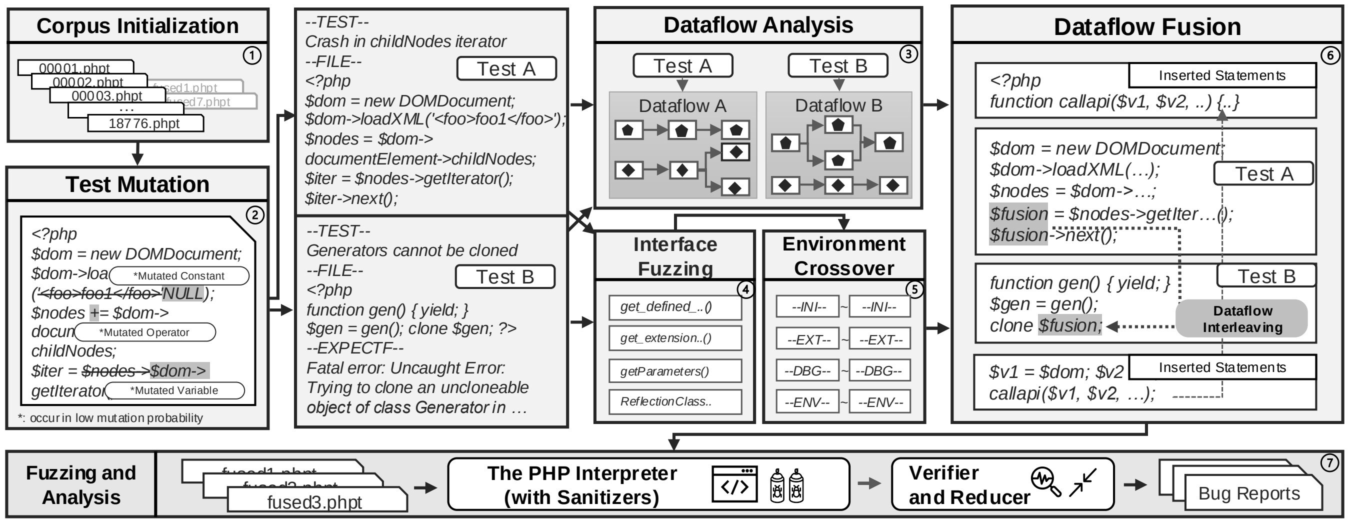
4.1 Approach Overview

Our approach is designed to uncover memory errors in the PHP interpreter. We utilize existing official test cases as seed programs to generate numerous new fused tests for continuous automatic fuzzing. The advantages of reusing the official test suite are twofold: (i) these tests are well-maintained over time with valid grammar and rich semantics, and (ii) their large quantity provides a substantial basis for test generation. By combining two or more tests, one can create a very much larger set of test cases which can also inherit the quality of the original tests. Our approach is inherently compatible and sustainable with the continuous addition of official test cases by PHP developers, which can automatically cover new features and bugs, ensuring more comprehensive testing.

We designed our approach with the following seven steps: (1) corpus initialization, (2) test mutation, (3) dataflow analysis, (4) interface fuzzing, (5) environment crossover, (6) dataflow fusion, and (7) result analysis. We present the overview of our approach in Figure 1 with a real unknown segmentation fault[6] found by FlowFusion.

At step ①, FlowFusion initializes the test corpus with the 19K unique test cases from the official test suite. We first randomly select two seed tests from the corpus (for simplicity, we consider two tests, but more can be used). At step ②,

[6] https://github.com/php/php-src/issues/14741



```
<?php
$dom = new DOMDocument;
$dom->loadXML(~~('<foo>foo1</foo>'~~'NULL);
$nodes += $dom->
documentElement->childNodes;
$iter = ~~$nodes->~~$dom->
getIterator();
```

```
--TEST--
Crash in childNodes iterator
--FILE--
<?php
$dom = new DOMDocument;
$dom->loadXML('<foo>foo1</foo>');
$nodes = $dom->
documentElement->childNodes;
$iter = $nodes->getIterator();
$iter->next();
```

```
--TEST--
Generators cannot be cloned
--FILE--
<?php
function gen() { yield; }
$gen = gen(); clone $gen; ?>
--EXPECTF--
Fatal error: Uncaught Error:
Trying to clone an uncloneable
object of class Generator in …
```

```
<?php
function callapi($v1, $v2, ..) {..}

$dom = new DOMDocument;
$dom->loadXML(…);
$nodes = $dom->…;
$fusion = $nodes->getIter…();
$fusion->next();

function gen() { yield; }
$gen = gen();
clone $fusion;

$v1 = $dom; $v2
callapi($v1, $v2, …);
```

Figure 1: Approach Overview — Illustrated with Segfault Error Found by FlowFusion in the Zend Engine

FlowFusion applies test mutations to these chosen tests. The key idea behind test mutation is to introduce additional randomness or perturbations (in a low probability) to the seed tests before they are fused. This is done by interchangeably replacing operands or expressions, or substituting them with special values.

At step ③, FlowFusion employs a variable-level dataflow analysis on the program of each test. In Figure 1, FlowFusion finds the dataflow ([`$dom`→`$nodes`→`$iter`]) in test A and the dataflow ([`gen()`→`$gen`]) in test B. The extracted dataflows will be used at step ⑥ later. Next, FlowFusion prepares statements for fuzzing interfaces provided by the PHP interpreter at step ④. It reuses variables from the fused code semantic as the arguments to call random functions implemented by PHP developers. We insert statements above and below the fused test to define random function calls, randomly select variables as arguments, and invoke function interfaces accordingly.

At step ⑤, FlowFusion first merges the execution environments of seed tests and then inserts random configurations from the test suite. As described in Section 2, the official test cases include not only the PHP program but also other sections, such as required modules and configuration settings. Environment crossover begins by merging other sections of two seed test cases and then randomly inserts random configurations that are collected from the official test suite. This process ensures that the prerequisite conditions for fused tests are met by combining these additional sections, while also introducing variability into the execution environments through random configuration insertions.

At step ⑥, dataflow fusion links some dataflows from test A with test B to connect two seed tests. The key insight is to interleave dataflow by connecting variables across seed tests. As shown in Figure 1, our approach creates a new test by replacing variable `$iter` and `$gen` with a bridging variable `$fusion` by the guidance of extracted dataflow in the previous step. The code semantic is changed accordingly, now cloning the DOM node rather than previously a function object, which is not tested in the official test suite. Details of dataflow fusion are explained in Section 4.2.

Finally, in step ⑦, after our approach completes dataflow fusion, the PHP interpreter is invoked to execute the fused test cases and check for any sanitizer violations. We rely on the sanitizer oracle (*i.e.,* ASan and UBSan) to detect bugs. Note that although we focus on detecting memory error bugs, as we are using an undefined behavior sanitizer, we also catch other bugs like integer overflows. The execution generates logs of standard inputs and errors, as well as reproducing scripts. We use these outputs to deduplicate crashes by checking the crash site and mapping the backward stack trace. Once verified, FlowFusion reduces the reproducer of sanitizer violations to a minimal version using delta debugging [57]. Last, we assess the potential security impact of the discovered bugs and follow the PHP community's disclosure policy: (i) reporting low-severity bugs (*e.g.,* null pointer dereference) directly at issues and (ii) reporting high-severity bugs (*e.g.,* stack overflow, heap use-after-free) at the security page, where vulnerability reports remain private until analyzed and fixed.

4.2 Dataflow Fusion

We introduce our core insight, dataflow fusion, by referring to Algorithm 1. Dataflow fusion aims to combine two programs, A and B, extracted from two random seed tests, into a single fused program F. The process begins by calculating the IN and OUT sets of dataflow facts for each program using the `ComputeDataflowSets` function. This function initializes the IN and OUT sets for each node within a program based on GEN and FUN sets and iteratively computes these sets using Equation 2 and Equation 3. Specifically, for each node n_i, the IN set is derived from the OUT set of the preceding node, while the OUT set is computed using a union of GEN, IN, and FUN sets minus the KILL set.

Algorithm 1 Pseudocode of Dataflow Fusion

```
 1: Input: programs $A$, $B$ extracted from two random seed tests
 2: Output: fused program $F$
 3: function ComputeDataflowSets($P$)
 4:     Initialize $\text{IN}(n_1) = \emptyset$ and $\text{OUT}(n_1) = \text{GEN}(n_1) \cup \text{FUN}(n_1)$
 5:     for $i = 2$ to $N$ do
 6:         Compute $\text{IN}(n_i) = \text{OUT}(n_{i-1})$
 7:         Compute $\text{OUT}(n_i) = \text{GEN}(n_i) \cup (\text{IN}(n_i) \setminus \text{KILL}(n_i)) \cup \text{FUN}(n_i)$
 8:     end for
 9:     return $\{\text{IN}(n_i), \text{OUT}(n_i) \mid \forall i \in \{1, 2, \ldots, N\}\}$
10: end function
11: do create a new shared variable $fusion$ that connects to both programs
12: let $FS^A \leftarrow$ ComputeDataflowSets($A$)              ▷ sets of dataflow facts
13: do identify all dataflows $DF^A$ in $FS^A$ by dataflow propagation
14: let $df_i^A \in DF^A$ be each dataflow in $DF^A$
15: let $w_i^A \leftarrow$ number of variables for each dataflow $df_i^A$
16: let $df^A_{\text{selected}} \leftarrow$ weighted random selection using $\{w_1^A, w_2^A, \ldots, w^A_{|DF^A|}\}$
17: let $V^A \leftarrow$ the set of variables in $df^A_{\text{selected}}$
18: let $v^A \leftarrow$ random variable selection from $V^A$ to be replaced
19: do identify all locations $L^A = \{l_1, l_2, \ldots, l_m\}$ in $A$ where $v^A$ appears
20: for each $l \in L^A$ do
21:     do replace $v^A$ with $fusion$ at location $l$ with a probability $p$
22: end for
23: let $A' \leftarrow A$                              ▷ new program after replacements
24: do repeat lines 12-22 for program $B$ and get $B'$
25: let $F \leftarrow A' + B'$        ▷ with shared variable, concatenate new programs
26: return $F$                      ▷ finish dataflow fusion, return fused program
```

The fusion process involves creating a shared variable called `fusion` to connect programs A and B. The algorithm identifies all dataflows within program A by propagating among sets of dataflow facts via dataflow analysis. Next, it assigns weights to these dataflows based on the number of variables. The algorithm then randomly selects a dataflow $df^A_{selected}$ from program A using a weighted selection process. Within this selected dataflow, a variable is chosen randomly, and all its locations in program A are identified.

The final steps of the algorithm involve randomly replacing occurrences of the selected variable in program A with the shared variable `fusion` with probability p (we used p=0.5), resulting in a modified program A'. The same process is repeated for program B, yielding a modified program B'. The fused program F is then created by concatenating A' and B' with the shared variable, completing the dataflow fusion. The algorithm returns the fused program F as the final output.

We highlight that compared to a naive approach to bridge the last variable of the first test to the first variable of the second test, we introduce more diversity (and randomness) to create many more possible versions of the fused test case which in turn leads to new coverage as shown by the experiments. We introduce the following heuristics: (i) Random dataflow. Each test can have multiple dataflows. We assign different weights to each dataflow and randomly select the fused dataflow based on the weights. (ii) Random variable. Each dataflow can have multiple variables. FlowFusion randomly picks one variable as the connection variable to be replaced by an intermediate variable. (iii) Random replacement. When interleaving dataflow, FlowFusion randomly picks one or more places of each variable from the source code and replaces them with an intermediate variable that connects to both tests.

The example presented in Figure 1 shows the power of dataflow interleaving and the heuristics above. Our approach changes the semantics of the test B by replacing the last occurrence of `$gen` (*i.e.,* `clone $gen` becomes `clone $fusion`). Note that this particular fusion has maintained the semantics of `gen()` using yield. We contrast with a naive dataflow fusion that connects the last variable of test A and the first variable of test B (*i.e.,* `$iter` and `$gen`), this does not find the memory error since the semantics is different. Another possible fusion is to link the final `clone` statement with an empty object. We have tested and such a trivial fusion does not trigger the bug.

4.3 Further Strategies

We introduce three helpful strategies along with dataflow fusion when combining seed tests.

Test mutation. In addition to dataflow fusion, FlowFusion adds mutations to seed programs before they are analyzed and fused to provide more code semantics. Our approach introduces small differences by replacing expressions and operands. We list some demonstrative test mutations in the following table with examples. They are effective while maintaining valid syntax. Each test mutation is applied at a small rate to introduce extra randomness to the seed programs.

ID	Expression	Replacements	Example
01	arithmetic	other arithmetic operands	$a = $b ~~(%)~~+ 1
02	assignment	other assignment operands	$a ~~(=)~~*= $b + 1
03	logical	other logical operands	$a = $b ~~(and)~~or 1
04	integer	special values like int_max	$a = $b + ~~(1)~~int_max
05	string	special values like null	$a = $b.~~("a")~~null
06	variable	other variables	$a = ~~($b)~~$c + 1

Interface fuzzing. Dataflow fusion introduces new, more complex code semantics. FlowFusion fuzzes internal PHP interfaces (*i.e.,* existing functions implemented by PHP developers) in the enriched code semantics of fused tests. The insight is to test the robustness of function implementations with more complex code semantics, which might not have been tested previously. Specifically, FlowFusion inserts additional statements both above and below the fused program. The inserted statements include a fuzzing function that invokes random PHP functions and collects variables (using `get_defined_vars()`) from the fused test as the function arguments. For obtaining available PHP function interfaces, FlowFusion uses internal functions (*e.g.,* `get_defined_functions()`, `get_loaded_extensions()`, and `get_extension_funcs()`) to dynamically fetch around 1,682 functions (excluding posix-related functions) as the interface candidates. FlowFusion uses the `ReflectionFunction` PHP class and its method getNumberOfParameters(), getParameters(), and getType() to determine the number and types of function parameters dynamically. FlowFusion also inserts various logging statements to record execution for bug reproduction.

```
<?php
  class A { public $prop { get {} } }
  class B extends A { }
  get_class_vars('B');
/* UBSan: apply non-zero offset to null pointer */
```

The reduced program above shows an unknown segmentation fault FlowFusion found via interface fuzzing. We call the function (*i.e.,* get_class_vars()) by reusing the variable (reduced) with the value "B" from the fused code.

Environment crossover. FlowFusion merges other sections of test cases and inserts additional randomness of PHP configurations. Other significant sections contain the --extension-- and --ini--, which specifies the execution environments of executing the tests. When fusing dataflow, FlowFusion also performs a crossover mutation by merging other sections. Additionally, for --ini-- section, the official test cases provide a wide range of options so that FlowFusion collects them as a dictionary and applies configurations randomly. Interesting configurations include memory limits, JIT mode, optimization levels, opcache,[7] and script preload. FlowFusion fuzzes the PHP interpreter with fused code semantics along with changing environments. This strategy brings extra effectiveness to memory error detection. For example, we detect 3 memory errors related to phpdbg[8] by merging the debugging instructions in --phpdbg-- sections.

5 Implementation

We developed our automatic fuzzing tool, FlowFusion, with over 3,000 lines of Python code.

FlowFusion fuzzer. FlowFusion's fuzzer is built upon the official testing script used in the PHP interpreter. This script checks all ".phpt" files, parses each section, and invokes the PHP interpreter to gain execution results. However, the script is not designed to handle customized ".phpt" files and occasionally terminates the fuzzing process unexpectedly. To address this, we patched the script by commenting out several lines, increasing its tolerance for execution failures, and ensuring our fuzzing completes successfully in parallel.

Bug verifier and PHP program reducer. We use the reproducing scripts generated by the official script to verify detected memory errors. After verification, we rerun verified memory errors in the normal PHP interpreter without sanitizer to observe their outputs (*i.e.,* crash or not). We implemented our PHP program reducer based on the principles of delta debugging [57]. The algorithm systematically comments out specific lines or groups of lines to determine whether the bug oracle (sanitizers) continues to trigger an abort. 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. Our reducer effectively reduces the bug-inducing program (typically reducing it to $\sim$ 10% of its original size). Other tools like C-Reduce [39] can be applied to further reduce the program size.

6 Evaluation

In this section, we answer the following questions to assess various important aspects of FlowFusion:

- **New memory errors**. How effective is FlowFusion in discovering new memory errors in the PHP interpreter?
- **Improved effectiveness**. To what extent does FlowFusion improve the effectiveness compared to (i) the official test suite and (ii) test concatenation?
- **Comparison with existing approaches**. What is the improvement in fuzzing the PHP interpreter with the state-of-the-art fuzzing approaches?
- **Ablation study**. What is the impact of each strategy of FlowFusion on contributing to the overall effectiveness?

PHP version and compilation. In Section 6.1, as well as during daily fuzzing, we compiled the latest commit of PHP interpreter using clang-15,[9] with debug symbols, address sanitizers, and undefined behavior sanitizers enabled. In other sections (6.2, 6.3, and 6.4), we performed evaluations on a specific commit (*i.e.,* v8.3.3, 3a832a2aad405466c24a5e8e5798cf9de14fda14) of the PHP interpreter. For Sections 6.2 and 6.4, we compiled the PHP interpreter using Ubuntu 22.04's default CC (gcc-11.4.0) for better compatibility, with debug symbols, sanitizers, and gcov support enabled. In Section 6.3, we instead used afl-gcc for comparison with existing work, while similarly enabling debug symbols, sanitizers, and gcov support.

Evaluation metrics. We explain two evaluation metrics we used in this section to measure the effectiveness of FlowFusion and related approaches. (i) Code coverage. Code coverage is a widely used metric for evaluating the effectiveness of fuzzing approaches [41, 42]. We collect code coverage using gcovr [13]. We follow the suggested gcovr from the official Makefile to measure the overall code coverage. We only report line code coverage in this work and omit the other coverage metrics because they follow a similar trend. (ii) Number of unique crash sites. We define the crash site as the unique file path of the PHP interpreter and line number from the sanitizer abort, which we use to approximate the number of detected bugs. For example, the motivating example in Listing 1 has the following sanitizer abort "AddressSani-

[7]https://www.php.net/manual/en/book.opcache.php
[8]https://www.php.net/manual/en/book.phpdbg.php

[9]We chose Clang given it has greater and more accurate sanitizer support, also shown by evaluation of memory error defenses [21].

tizer: heap-use-after-free */php-src/ext/dom/php_dom.c:311*" with the crash site being highlighted in italics. After a period of fuzzing, the number of unique crash sites can be used to de-duplicate similar aborts, hence demonstrating the effectiveness of various fuzzing approaches. Based on our experience, most previously unknown crash sites can be reported as bugs to the PHP interpreter and are likely to be fixed.

Experimental infrastructure. All experiments were conducted on an AMD EPYC 7763 processor with 64 physical cores and 128 logical cores, clocked at 2.45 GHz. The test machine ran Ubuntu 22.04 and was equipped with 512 GB of RAM. By default, FlowFusion uses a modest computing resource, 32 CPU cores with up to 32 GB of RAM, capable of finding various crash sites within 24 hours.

6.1 Discovering Unknown Memory Errors

To discover unknown memory errors in the PHP interpreter, we intermittently tested the latest version of the PHP interpreter, compiled from the php-src repository, over a six-month period. This approach aligns with standard methodologies for evaluating the effectiveness of automatic bug-finding tools [22, 42]. Next, we reduced the merged test cases and verified whether the issues had already been reported on issue trackers to avoid duplicate bug reports. Bug-inducing test cases generated by FlowFusion are often complex; we automatically simplified them to smaller, bug-inducing versions using delta debugging [57]. Below, we illustrate memory errors using these reduced PHP programs.

Results. Table 1 shows information about the first 50 bugs found by FlowFusion. The column *Bug Type* shows the corresponding Common Weakness Enumeration (CWE) type of bugs[10] to concisely present their root causes. All reported memory errors were identified with the sanitizer oracles, and the *Crash* column shows bugs that could crash the program even without sanitizers enabled, highlighting cases with more severe impacts. The *Bug Location* provides the accurate bug locations of all fixed bugs. The *Issue ID* column provides the official GitHub issue number of the corresponding bug. The *Status* shows the bug status. We classified the bug status into the following disjoint categories:

- *Pending* (`Pd`) bugs refer to the bugs submitted but awaiting further investigation to confirm the root cause.
- *Expected* (`Ep`) bugs refer to the bugs been confirmed but developers believe they are expected behaviors.
- *Confirmed* (`Cf`) bugs refer to the bugs that have been confirmed but have not been fixed.

Table 1: First 50 Bugs Found by FlowFusion

ID	Bug Type	Crash	Bug Location	Issue ID	Status	Fixes
01	CWE-825	✓	session.c	13680	Fx	+28 -2
02	CWE-476	✓	phpdbg_watch.c	13681	Fx	+51 -6
03	CWE-476	✓	spl_directory.c	13685	Fx	+80 -18
04	CWE-121	-	-	13768	Ep	
05	CWE-476	✓	phpdbg_frame.c	13827	Fx	+41 -2
06	CWE-476	✓	phar.c	13833	Fx	+68 -20
07	CWE-476	✓	zend_jit.c, ..	13834	Fx	+29 -24
08	CWE-476	✓	stream.c	13836	Fx	+36 -2
09	CWE-476	✓	mod_user_class.c	13856	Fx	+23 -2
10	CWE-190	-	-	13881	Pd	
11	CWE-124	-	-	13903	Ep	
12	CWE-476	✓	main.c	13931	Fx	+52 -0
13	CWE-122	✓	sqlite_driver.c	13984	Fx	+19 -1
14	CWE-457	✓	sqlite_driver.c	13998	Fx	+25 -1
15	CWE-401	-	compat.c	14044	Fx	+35 -5
16	CWE-190	-	-	14075	Pd	
17	CWE-476	✓	-	14082	Cf	
18	CWE-824	✓	xml.c	14124	Fx	+28 -0
19	CWE-121	✓	-	14164	Ep	
20	CWE-476	✓	spl_iterators.c	14290	Fx	+24 -3
21	CWE-401	-	document.c	14343	Fx	+24 -1
22	CWE-121	✓	zend_compile.c, ..	14361	Fx	+54 -8
23	CWE-476	✓	zip.c	14603	Fx	+1 -1
24	CWE-476	✓	simplexml.c	14638	Fx	+57 -24
25	CWE-476	✓	spl_observer.c	14639	Fx	+29 -4
26	CWE-824	✓	basic_functions.c	14643	Fx	+5 -2
27	CWE-824	✓	php_dom.c	14652	Fx	+21 -1
28	CWE-825	✓	spl_directory.c	14687	Fx	+42 -1
29	CWE-825	✓	libxml.c	14698	Fx	+29 -3
30	CWE-190	-	-	14709	Pd	
31	CWE-476	✓	zend_execute.c, ..	14712	Fx	+32 -1
32	CWE-190	-	-	14732	Pd	
33	CWE-825	✓	zend_interfaces.c	14741	Fx	+18 -0
34	CWE-190	-	basic_functions.c	14774	Fx	+29 -0
35	CWE-190	-	array.c	14775	Fx	+16 -0
36	CWE-190	-	file.h	14780	Fx	+62 -7
37	CWE-476	✓	output.c	14808	Fx	+18 -1
38	CWE-476	✓	text.c	15137	Fx	+14 -1
39	CWE-190	-	-	15150	Pd	
40	CWE-121	✓	-	15168	Cf	
41	CWE-121	✓	-	15169	Cf	
42	CWE-476	✓	url_scanner_ex.re	15179	Fx	+24 -1
43	CWE-476	✓	output.c	15181	Fx	+19 -0
44	CWE-825	✓	-	15187	Cf	
45	CWE-825	✓	php_dom.c	15192	Fx	+77 -1
46	CWE-824	✓	phpdbg_bp.c	15208	Fx	+57 -0
47	CWE-416	✓	phpdbg.h, ..	15210	Fx	+75 -3
48	CWE-122	✓	phpdbg_info.c	15210	Fx	+34 -6
49	CWE-416	✓	nodelist.c	15143	Fx	+29 -2
50	CWE-416	✓	php_pcre.c, ..	15205	Fx	+58 -47

- *Fixed* (`Fx`) bugs refer to the bugs that have been confirmed and patched by the developers.

Bug diversity. In total, we identified 158 previously unknown bugs. Of these, 18 are still pending triage, 4 were expected, 11 have been confirmed, 0 were marked as duplicates, and 125 have already been fixed. All 158 errors were detected by sanitizers, and 39 of them caused the PHP interpreter to crash when compiled normally without sanitizers. We present the first 50 bugs listed in Table 1, which covers nearly all common types of memory errors. This also demonstrates that FlowFusion can significantly enhance the security of the PHP interpreter finding many different errors across dif-

[10]The bugs found by FlowFusion covered the following CWEs: CWE-121 Stack-based Buffer Overflow, CWE-122 Heap-based Buffer Overflow, CWE-124 Buffer Underwrite, CWE-190 Integer Overflow or Wraparound, CWE-401 Missing Release of Memory after Effective Lifetime, CWE-416 Use After Free, CWE-457 Use of Uninitialized Variable, CWE-476 NULL Pointer Dereference, CWE-824 Access of Uninitialized Pointer, CWE-825 Expired Pointer Dereference

Listing 3: Heap Use-after-free with PCRE in PHP

```
<?php
  $array = new ArrayIterator(..);
  $regex = new RegexIterator($array, '/Array/');
  foreach ($regex as $match) { }
  $fusion = $regex;
/* AddressSanitizer: heap-use-after-free */
```

Listing 4: Heap Overflow with SQLite Extension in PHP

```
<?php
  $dbfile = $GLOBALS[array_rand($GLOBALS)];
  $db = new PDO('sqlite:'.$dbfile, null, null, ..);
/* AddressSanitizer: heap-buffer-overflow on .. */
```

ferent modules in PHP. From the bugs in Table 1, FlowFusion identified 8 instances of out-of-bounds errors (*e.g.,* stack/heap buffer overflows/underflows) and 3 use-after-free bugs. Additionally, FlowFusion detected 18 null-pointer dereference bugs, 1 use-of-uninitialized-variable bug, 2 expired-pointer dereference bugs, and 6 access-of-uninitialized-pointer bugs, all of which involve "bad" pointer access that can cause the PHP interpreter to crash unexpectedly. We also detected several non-crash bugs, including 3 memory leaks and 9 signed integer overflows.

We found bugs affecting diverse functionalities within the PHP interpreter, as illustrated in the locations of fixed bugs, which are also shown in Table 1. In total, patches generated by FlowFusion have impacted 40 individual files, including critical source files (*e.g.,* zend_*.c, main.c). Specifically, the developers changed over 1,539 lines of code (*i.e.,* 1,338 additions and 201 deletions) as a direct response to our reports for security improvements in the official PHP interpreter repository. The bugs found by FlowFusion mostly have existed in the PHP interpreter from several months to years. Notably, we found one use-after-free bug (the motivating example shown in Listing 1) that has existed for more than 20 years, which developers described as "*an ancient bug*".

Next, we highlight notable bugs found by FlowFusion, describing their root causes and security impacts based on our analysis and developers' feedback.

Bug analysis 1: heap use-after-free with PCRE. The PHP Perl Compatible Regular Expressions (PCRE) module facilitates pattern matching in PHP scripts using functions like `preg_match()`. A heap use-after-free vulnerability, illustrated in Listing 3, arises due to the premature shutdown of the PCRE module before all live objects are destroyed. Consequently, when the Standard PHP Library (SPL) attempts to clean up a regular expression object, it operates on freed memory, leading to a use-after-free error.

FlowFusion discovered this bug by reassigning the regular expression variable after completing all regular expression operations, guided by dataflow fusion. The original query included a series of standard regular expression statements along with another script containing typical assignments. As described in Section 4.2, FlowFusion randomly replaces variables to create bridging connections for dataflow fusion, potentially linking regular expression variables with subsequent unrelated assignments, which then triggered this bug.

Listing 5: Null pointer dereference in Zend compiler

```
<?php
  register_shutdown_function(function() {
    var_dump(eval("return␣1+3;")); });
  eval(<<<EVAL
    function f(){ try { break; } finally {}} f();
    EVAL);
/* UBSan: applying zero offset to null pointer */
```

Listing 6: Segmentation fault in JIT and opcache

```
<?php
  class Foo { public static function test() {
      static $i = 0; var_dump(++$i); } }
  Foo::test();
/* AddressSanitizer: SEGV on unknown address */
```

Bug analysis 2: heap overflow in PHP SQLite module. SQLite is a widely used lightweight database system embedded in the PHP interpreter as an in-memory storage solution. Listing 4 illustrates a heap overflow error in the SQLite module within the PHP interpreter. This bug-triggering program first assigns a database variable as the SQLite file and then creates a new SQLite object. However, because the buffer size is not checked before *memcmp*, the allocated buffer exceeds the expected size in the heap, triggering an alert from ASan. FlowFusion found this issue by interleaving unrelated variables with SQLite statements.

Bug analysis 3: null pointer dereference in the Zend compiler. The Zend engine contains a null pointer dereference vulnerability, as shown in Listing 5. This bug first causes the compiler to terminate due to a fatal error, leaving its data structures with stale values. During the next compilation, elements from the previous stack are incorrectly reused, leading to a segmentation fault because wrong instructions are emitted due to the stale data.

Bug analysis 4: segmentation fault in JIT and opcache. The PHP interpreter supports just-in-time compilation through its JIT compiler and accelerates execution using opcache. Listing 6 presents a related issue that causes the Zend compiler to crash. This issue was identified in a specific environment with an opcache preload configuration. Developers noted that "*the issue arises because caller_info, callee_info, and possibly call_map are allocated in the arena but are not reset before being used by the next request.*"

Bug analysis 5: segmentation fault in the session module. The session module in the PHP interpreter preserves data across visits by assigning each visitor a unique session ID. This support allows data storage between requests using the `$_SESSION` superglobal array. Listing 7 gives a segmen-

Listing 7: Segmentation fault in session extension

```
<?php
  ob_start();
  ini_set("session.serialize_handler", ..);
  session_start();
  $result1 = session_decode('foo|s:3:"bar";');
  class Test extends DateTime {
    public static function createFromFormat(
    $format, $datetime, $timezone = null): Wrong{}
  }
/* AddressSanitizer: SEGV on unknown address */
```

tation fault found in the session module. This fault arises from changes made to the session decode process to prevent writing incomplete sessions. According to the developers, "*it is illegal to return from a bailout because that does not restore the original bailout data*", the fix makes the termination outside of the original data.

Discussion—false alarms. We observed three false alarms during our experiment that are linked to the expected (**`Ep`**) bugs in Table 1. There are two reasons for these false alarms: (i) FlowFusion relies on sanitizers as the bug oracle. However, it is known that sanitizers can have incorrect results [30]. One of the expected bugs is explained by developers as "*This is an ASAN false positive ... gives us a memory region that is adjacent to the region where the fiber stack used to be, but that still has the old shadow memory, and so we get a false ASAN warning*". (ii) Another cause of false alarms is that the sanitizer may abort before the overflow handler is triggered. The PHP interpreter has a conservative stack limit, which can sometimes be reached after the sanitizer issues an alert. This results in false alarms for the remaining two expected bugs.

6.2 Improvement on Official Test Suite

One way to evaluate the effectiveness of FlowFusion is by assessing how well it improves upon the official test suite in the PHP interpreter and the naive approach of test concatenations. To achieve this, we manually analyzed the first 50 bugs identified by FlowFusion, as detailed in Table 1. Our findings reveal that only 3 bugs can be reproduced using test concatenation, and none can be reproduced using the official test suite, highlighting the remarkable effectiveness of our new approach.

Next, we performed a 24-hour fuzzing run, as recommended in previous research [26], under three different setups: (a) the official test suite, (b) concatenations of official test cases, and (c) dataflow fusion. Note we merge other sections to ensure a fairer comparison when evaluating the test concatenation. The enhanced effectiveness of FlowFusion is demonstrated through two key outcomes from the 24-hour fuzzing: (i) FlowFusion discovers more unique crash sites that the official test suite and the test concatenation cannot detect, and (ii) FlowFusion achieves higher code coverage compared to the official test suite and the test concatenation.

More unique crash sites. FlowFusion is effective in discovering unique crash sites in the PHP interpreter. We track the number of crash sites of three approaches. Our evaluation shows that FlowFusion can uncover over 20 unique crash sites on average, whereas the official test cases detect none, and test concatenation detects only 2. We expect the official test cases to only rarely detect these crash sites, as they are tested daily in automated continuous integration. Test concatenation can reveal some crash sites by merging other non-program sections, which creates different execution environments.

```
<?php
  class Test { public string $prop {
    set => strtoupper($value); } }
  $test = new Test();
  var_dump($test);
  foreach ($test as $longVal) {}
/* AddressSanitizer: SEGV on unknown address */
```

The example test above illustrates where the memory error found by FlowFusion is missed by both the official test suite and test concatenation. This error occurs in the Zend allocator and is triggered when a non-related class object from one seed test is assigned to a `foreach` statement in another test. The official test cases lacked a test to check such code semantics while test concatenation fails to establish connections between merged tests. Thus both approaches fail to find this bug.

Higher code coverage. Figure 2 illustrates that FlowFusion outperforms both the official test suite and the concatenation tests in terms of code coverage. Consistent with previous observations, the well-maintained official test cases consistently achieve a high code coverage of 76% over 24 hours. This result, while impressive and surpassing existing grammar-based fuzzers, remains static due to the limited size of the test suite.

Test concatenation, on the other hand, begins with lower coverage than the official test suite, likely due to compatibility issues or syntax errors introduced during the merging process (*e.g.,* unresolved namespace declarations), which cause some tests to fail. However, the coverage for test concatenation gradually increases over 24 hours, eventually surpassing that of the official test suite. In general, test concatenation demonstrates similar effectiveness to the official test suite.

FlowFusion, in contrast, achieves 80.4% with 4.3% higher code coverage than the official test suite after 24 hours, with coverage continuing to grow. FlowFusion explores a vast

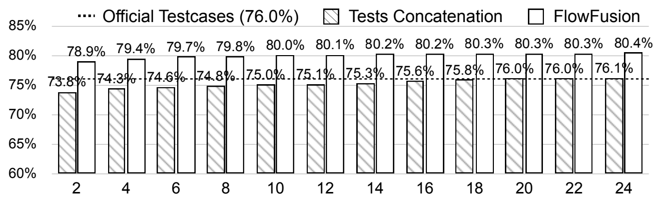


Figure 2: Code coverage over Test Suite and Concatenation

space of generated test cases to uncover more memory errors. This approach sacrifices some short-term efficiency for long-term gains in code coverage. Notably, FlowFusion continues to increase coverage beyond the 24-hour mark. For example, FlowFusion reaches 82.1% code coverage after 7 days, while the test suite's coverage remains unchanged.

6.3 Comparison with Existing Approaches

We assess the effectiveness of FlowFusion in identifying memory errors in the PHP interpreter compared to existing methodologies. We included AFL++ [12] since it is often considered the most effective general-purpose fuzzer. However, given that the PHP interpreter expects highly structured input, we also consider grammar-based fuzzing approaches, which we would expect to generate more valid inputs. To the best of our knowledge, no existing approach specifically focuses on PHP interpreter fuzzing. We thus consider the following fuzzers designed for general programming languages. Polyglot [8], which incorporates grammar guidance and semantic validation, has uncovered over 30 memory errors in the PHP interpreter, making it the most effective existing approach. Among other grammar-based approaches previously used to detect memory errors in the PHP interpreter, Langfuzz [18] is not publicly accessible. Nautilus [2] and Gramatron [48] found 3 and 4 memory errors in the PHP interpreter, respectively, indicating their limited effectiveness. Therefore, we select the most effective approach Polyglot as the state-of-the-art evaluation baseline.

To conduct a fair comparison, we extract all PHP programs from the official test suite and make them the seed inputs for AFL++ and Polyglot. We perform a 24-hour fuzzing evaluation by running AFL++, Polyglot, and FlowFusion under the same environment (limited to 16 cores for each approach) to compare their code coverage—the direct metric associated with fuzzing effectiveness. We compile three copies of the PHP interpreter using the same AFL-specific compiler[11] and compiler options (with gcov option and sanitizers).

Higher code coverage. Figure 3 shows the results of a 24-hour fuzzing experiment, where FlowFusion significantly outperforms both AFL++ and Polyglot. Notably, while using all official test cases as seed programs, AFL++ and Polyglot achieve code coverages of around 60%, which falls short compared to the 78% coverage achieved by the official test suite alone. This discrepancy arises due to two main reasons. First, AFL++ and Polyglot are not PHP-specific fuzzers, and face difficulties in fuzzing various sections of the test cases, such as the `--ini--` and `--extensions--` sections, as illustrated in Listing 2. Consequently, they often fail to dynamically meet the additional module requirements of the generated programs and to mutate the execution environments specified in these configuration sections. This limitation results in inadequate testing of core PHP interpreter features, such as JIT compilation. Second, as the seed programs are already diverse in their semantics and syntactically correct, the room for improvement using grammar guidance and semantic validation in AFL++ and Polyglot is limited. In contrast, FlowFusion enhances the official test suite by generating more complex code semantics through dataflow fusion, thereby increasing its semantic diversity.

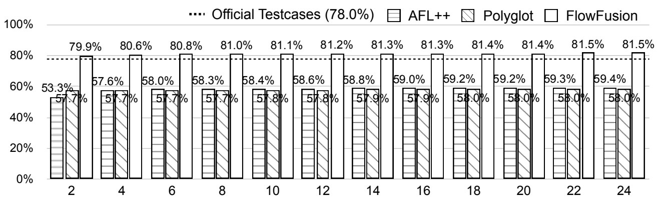


Figure 3: Coverage Comparison against AFL++ and Polyglot

[11]Coverage results differ from Section 6.2 as we change to AFL-clang compiler.

More memory errors. We also counted the number of memory errors detected by these three approaches over a 24-hour period. AFL++ and Polyglot detected 49 and 21 unique "crashes", respectively, during this time (Polyglot's efficiency is reduced with semantic validation enabled, resulting in fewer detected crashes). These "crashes" are not the result of sanitizer aborts, rather they are all "Fatal errors" from the PHP interpreter. However, such fatal errors can be intentional rather than true bugs (*e.g.,* if the test case has an expected abort result, it is not an error). We found most "Fatal errors" from AFL++ and Polyglot belong to expected aborts. For example, in Figure 1, test B is expected to produce the following failure: "`Fatal error: Uncaught Error: Trying to clone an uncloneable object of class Generator in ...`". Through manual verification, we confirmed that all "Fatal errors" reported by AFL++ and Polyglot are unrelated to memory errors. In contrast, FlowFusion identified 16 unique crash sites flagged by sanitizers, primarily related to memory safety issues. FlowFusion can also detect all "Fatal errors" from the PHP interpreter, however, this would result in a high number of false positives which is undesirable.

6.4 Ablation Study

We examined the impact of key strategies used in our approach on overall effectiveness via an ablation assessment. Specifically, we evaluated three strategies introduced in Section 4.3: (i) test mutation, (ii) interface fuzzing, (iii) environment crossover, and the key method (iv) dataflow fusion. For this assessment, we configured FlowFusion by individually disabling each of these strategies, using a fully enabled version of FlowFusion as the reference. To ensure consistency, we compiled five identical copies of the PHP interpreter using the same compiler and options, and evaluated these metrics over 24 hours of fuzzing at the same time, with each configuration limited to 16 parallel threads.

We use code coverage to assess their contribution to effec-

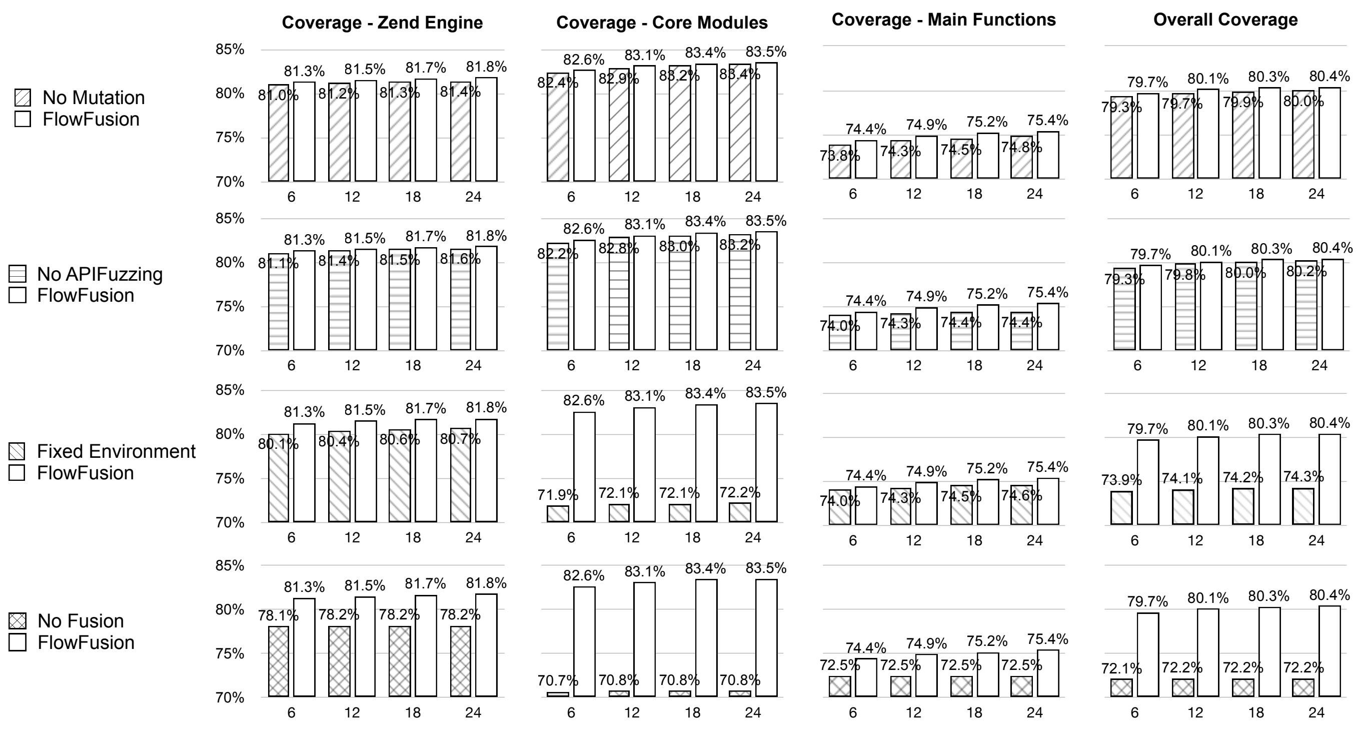


Figure 4: Ablation Study of FlowFusion

tiveness. We provide detailed coverage data for three main components of the PHP interpreter introduced in Section 2: (i) Zend engine, (ii) core modules, and (iii) main functions, along with (iv) the overall coverage. Figure 4 presents the evaluation results with coverage between 70% to 85%. Focusing first on the overall coverage column, we rank the contributions to coverage as follows: dataflow fusion contributes the most with 8% decrease, followed by environment crossover with 6% decrease, and finally, test mutation and interface fuzzing with small 0.2% to 0.4% decreases. Notably, two significant decreases in code coverage occur with the fixed environment and the absence of dataflow fusion. These strategies are directly related to test fusion—one involving the fusion of program sections and the other merging the rest sections.

We then analyze the three components of the PHP interpreter in relation to overall code coverage. Test mutations result in minor increases across all components, with the Zend engine and main functions showing slightly greater gains than the core modules. Interface fuzzing noticeably boosts coverage in the main functions. Environment crossover leads to a significantly larger increase in the code modules compared to other components. Notably, dataflow fusion drives substantial growth across all components, underscoring its key role in enhancing coverage.

7 Discussion

We present several limitations of FlowFusion and discuss possible solutions or future works.

Efficiency. The efficiency of FlowFusion has not been optimized for exploring larger search spaces. We prioritize efficiency over accuracy by abstracting coarse-grained dataflow from the finer-grained taint tracking used in vulnerability detection [20]. However, this broad approach offers an additional opportunity to uncover subtle memory errors. By doing so, we have identified more memory errors, some of which developers have remarked as "*a bit 'nonsensical' but should not cause a crash*" type bug. Introducing semantic validation could be a potential optimization for FlowFusion. This would involve more extensive static and dynamic analysis to guide more accurate test fusion, effectively narrowing the search space, and thereby improving performance but potentially missing some edge cases.

Feedback. We initially considered incorporating coverage feedback into our approach but ultimately decided against it due to the substantial overhead it introduced. The basic concept was to evaluate each fused test case and add it to the test corpus if it covered more lines of the PHP interpreter. This would involve monitoring code coverage after every execution of fused test cases, which resulted in excessive overhead. We leave this optimization for future work.

Bug oracle. We use sanitizers as our primary bug oracle; however, additional bug oracles can be incorporated into our approach to detect further bugs like logic bugs.

Scalability to other programming languages. The high-level insight of FlowFusion is extensible to other programming languages like C/C++ or JavaScript. We have initial work extending FlowFusion to JavaScript, and while it is early, it has already found JavaScript bugs.

Dependency on the test cases. Relying on test cases benefits FlowFusion by automatically reflecting any updates made by developers, but it also constrains FlowFusion's scalability—requiring a well-maintained test suite of high quality and ample coverage.

8 Related Work

In this section, we examine related work on PHP application security, historical information reuse, and compiler/interpreter testing and fuzzing to underscore the importance of analyzing the PHP interpreter and to establish the foundation of our approach, FlowFusion.

PHP application security. As one of the most popular languages for web deployments, the security problems in PHP-related applications have recently attracted increasing attention. To protect these applications, existing solutions focus on attack surface mitigation [3, 4, 19], bug detection [1, 29, 32, 34, 38], and defense mechanism enhancement [7, 31, 40]. For example, Minimalist [19] and AnimateDead [3] use debloating strategies to reduce the code size of applications, such as minimizing critical API calls, thereby increasing the complexity and workload for attackers. WHIP [1] enables static application security testing (SAST) tools to collaborate by sharing information, which helps to trigger more security alerts. TChecker [32] introduces a context-sensitive inter-procedural static taint analysis tool to detect taint-style vulnerabilities in PHP applications. Additionally, Saphire [7] applies the principle of least privilege (PoLP) to PHP applications and proposes a novel, generic approach for automatically deriving system-call policies for individual interpreted programs. BrowserShield [40] utilizes a lightweight middlebox to prevent the exploitation of browser vulnerabilities. FlowFusion is orthogonal to these works, aiming to detect memory errors in the PHP interpreter, providing an additional low-level attack surface alongside common application-level bugs.

Reusing historic bugs or tests. Historical bug reports and test cases offer valuable insights into a system's internal state and potentially vulnerable points. These insights are widely used to construct new, meaningful inputs or seeds that help uncover additional vulnerabilities. For example, Oliinyk et al. [35] and Zhao et al. [58] analyze previous crash or bug reports to facilitate new seed generation, identifying new bugs in BusyBox and the Java Virtual Machine (JVM), respectively. Alternatively, some approaches [27, 28, 49, 50, 59] leverage existing test cases as guidance to explore program states and discover new vulnerabilities. For instance, SQuaLity [49] executes test suites across different Database Management Systems (DBMSs) to uncover new, previously unknown bugs. In the context of C/C++ compiler testing, the Equivalence Modulo Inputs (EMI) approach [27, 28, 50] mutates existing test programs to create semantically equivalent variants, aiding in bug discovery. Additionally, YinYang [54] goes a step further by generating new test inputs through semantic fusion, combining two existing formulas into a new one to detect soundness bugs in Satisfiability Modulo Theory (SMT) solvers. Guided by this, FlowFusion explores the potential of fusing high-quality test cases to discover PHP memory errors, enhancing the security of the PHP interpreter.

Compiler/Interpreter testing and fuzzing. Existing solutions have concentrated on fuzzing or testing popular compilers and interpreters, such as those for C/C++ [10, 55], Rust [47, 51], and JavaScript (JS) [6, 15, 53], to mitigate potential cascading security issues. For instance, GrayC [10] design a greybox, coverage-directed, mutation-based approach to fuzz C compilers and code analyzers using a new set of mutations to target common C constructs. RustSmith [47] executes differential testing between Rust compilers or across optimization levels to identify potential bugs. Comfort [56] leverages the deep learning-based language model to automatically generate JS test code to detect bugs in JS engines. Fuzzilli [15] presents the design and implementation of an intermediate representation (IR) aimed at uncovering vulnerabilities in JIT compilers. FuzzJIT [53] focuses on identifying JIT compiler bugs by triggering the JIT compilation process and capturing execution inconsistencies. CodeAlchemist [16] leverages semantics-aware assembly by merging code bricks from the seed to generate new test cases for fuzzing JavaScript engines. LangFuzz [18], NAUTILUS [2], Gramatron [48], and PolyGlot [8] are existing fuzzing approaches that have found memory errors in the PHP interpreter. Focusing solely on grammar may overlook the code semantics of programs, thereby limiting their bug-discovery capabilities. To address this, we developed FlowFusion to detect memory errors in the PHP interpreter by fusing dataflow from high-quality test cases, generating new and more complex code semantics.

Conclusion

In this paper, we introduced FlowFusion, the first automated fuzzing framework specifically designed to detect memory errors in the PHP interpreter through dataflow fusion and other innovative techniques. Our approach merges test cases by linking their dataflows, creating fused tests capable of uncovering previously undetected bugs. Comprehensive experiments demonstrated FlowFusion's effectiveness, revealing 158 bugs, with 125 successfully fixed and 11 confirmed, outperforming existing methods.

FlowFusion demonstrates its potential as a practical tool for improving the security and robustness of the PHP interpreter. Moreover, the principles behind FlowFusion 's design—especially the dataflow fusion technique detailed in

Algorithm 1—are not limited to PHP. We believe there is broader applicability to other programming languages, where FlowFusion could be effectively used to merge test cases based on dataflow relationships, making FlowFusion a valuable contribution to the field of language interpreter fuzzing.

Ethics Statement

In conducting this research, we have carefully considered the ethical implications at every stage, from design through publication. We avoided live experimentation on systems without proper authorization and ensured that any vulnerabilities identified during the research were responsibly disclosed to relevant parties. By following these ethical practices, we aim to minimize harm and ensure that our research contributes positively to the field. These considerations not only align with ethical standards but also promote responsible innovation in the computer security and privacy domains.

Open Science Statement

We are committed to openly sharing all research artifacts associated with this work. Our approach, FlowFusion, is now open source at https://github.com/php/flowfusion under an open-source license. Our commitment to open science aligns with the broader initiative to foster transparency and collaboration within the research community.

References

[1] Al-Kassar, F., Compagna, L., and Balzarotti, D. {WHIP}: Improving static vulnerability detection in web application by forcing tools to collaborate. In *32nd USENIX Security Symposium (USENIX Security 23)* (2023), pp. 6079–6096.

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

[3] Azad, B. A., Jahanshahi, R., Tsoukaladelis, C., Egele, M., and Nikiforakis, N. {AnimateDead}: Debloating web applications using concolic execution. In *32nd USENIX Security Symposium (USENIX Security 23)* (2023), pp. 5575–5591.

[4] Azad, B. A., Laperdrix, P., and Nikiforakis, N. Less is more: Quantifying the security benefits of debloating web applications. In *28th USENIX Security Symposium (USENIX Security 19)* (2019), pp. 1697–1714.

[5] Ba, J., Duck, G. J., and Roychoudhury, A. Efficient greybox fuzzing to detect memory errors. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (2022), pp. 1–12.

[6] Bernhard, L., Scharnowski, T., Schloegel, M., Blazytko, T., and Holz, T. Jit-picking: Differential fuzzing of javascript engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (2022), pp. 351–364.

[7] Bulekov, A., Jahanshahi, R., and Egele, M. Saphire: Sandboxing {PHP} applications with tailored system call allowlists. In *30th USENIX Security Symposium (USENIX Security 21)* (2021), pp. 2881–2898.

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

[9] CVEdetails.com. Top 50 Vendors By Total Number Of &quot;Distinct&quot; Vulnerabilities — cvedetails.com. https://www.cvedetails.com/top-50-vendors.php?year=0., 2024. [Accessed 31-08-2024].

[10] Even-Mendoza, K., Sharma, A., Donaldson, A. F., and Cadar, C. 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.

[11] Fernandez, J. M. A deep dive into disable_functions bypass and PHP exploitation. https://www.tarlogic.com/blog/disable_functions-bypasses-php-exploitation/, 2020. [Accessed 23-08-2024].

[12] Fioraldi, A., Maier, D., Eissfeldt, H., and Heuse, M. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)* (2020).

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

[14] Golofit, P. https://accesto.com/blog/is-php-still-relevant-in-2024/.

[15] Gross, S., Koch, S., Bernhard, L., Holz, T., and Johns, M. Fuzzilli: Fuzzing for javascript jit compiler vulnerabilities. In *NDSS* (2023).

[16] Han, H., Oh, D., and Cha, S. K. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *NDSS* (2019).

[17] Han, W., Joe, B., Lee, B., Song, C., and Shin, I. Enhancing memory error detection for large-scale applications and fuzz testing. In *Network and Distributed Systems Security (NDSS) Symposium 2018* (2018).

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

[19] Jahanshahi, R., Azad, B. A., Nikiforakis, N., and Egele, M. Minimalist: Semi-automated debloating of {PHP} web applications through static analysis. In *32nd USENIX Security Symposium (USENIX Security 23)* (2023), pp. 5557–5573.

[20] Ji, K., Zeng, J., Jiang, Y., Liang, Z., Chua, Z. L., Saxena, P., and Roychoudhury, A. {FlowMatrix}:{GPU-Assisted}{Information-Flow} analysis through {Matrix-Based} representation. In *31st USENIX Security Symposium (USENIX Security 22)* (2022), pp. 2567–2584.

[21] Jiang, Y., Yap, R. H., Liang, Z., and Rosier, H. Recipe: Revisiting the evaluation of memory error defenses. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security* (2022), pp. 574–588.

[22] Kamm, M., Rigger, M., Zhang, C., and Su, Z. Testing graph database engines via query partitioning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2023), ISSTA 2023, Association for Computing Machinery, p. 140–149.

[23] Khedker, U., Sanyal, A., and Sathe, B. *Data flow analysis: theory and practice*. CRC Press, 2017.

[24] kinsta. https://kinsta.com/php-market-share/.

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

[26] Klees, G., Ruef, A., Cooper, B., Wei, S., and Hicks, M. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2018), CCS '18, Association for Computing Machinery, p. 2123–2138.

[27] LE, V., AFSHARI, M., AND SU, Z. Compiler validation via equivalence modulo inputs. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation 49*, 6 (jun 2014), 216–226.

[28] LE, V., SUN, C., AND SU, Z. Finding deep compiler bugs via guided stochastic program mutation. *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications 50*, 10 (2015), 386–399.

[29] LI, P., AND MENG, W. Lchecker: Detecting loose comparison bugs in php. In *Proceedings of the Web Conference 2021* (2021), pp. 2721–2732.

[30] LI, S., AND SU, Z. Ubfuzz: finding bugs in sanitizer implementations. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (2024), pp. 435–449.

[31] LI, Z., TANG, Y., CAO, Y., RASTOGI, V., CHEN, Y., LIU, B., AND SBISA, C. Webshield: Enabling various web defense techniques without client side modifications. In *NDSS* (2011).

[32] LUO, C., LI, P., AND MENG, W. Tchecker: Precise static interprocedural analysis for detecting taint-style vulnerabilities in php applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (2022), pp. 2175–2188.

[33] MITRE. CWE - 2023 CWE Top 10 KEV Weaknesses — cwe.mitre.org. https://cwe.mitre.org/top25/archive/2023/2023_kev_list.html, 2023. [Accessed 24-08-2024].

[34] NEEF, S., KLEISSNER, L., AND SEIFERT, J.-P. What all the phuzz is about: A coverage-guided fuzzer for finding vulnerabilities in php web applications. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security* (2024), pp. 1523–1538.

[35] OLIINYK, Y., SCOTT, M., TSANG, R., FANG, C., HOMAYOUN, H., ET AL. Fuzzing busybox: Leveraging llm and crash reuse for embedded bug unearthing. *arXiv preprint arXiv:2403.03897* (2024).

[36] PHP. https://github.com/php/php-src/security/advisories/GHSA-h35g-vwh6-m678, 2024.

[37] PHP. https://github.com/php/php-src/security/advisories/GHSA-jqcx-ccgc-xwhv, 2024.

[38] RABHERU, R., HANIF, H., AND MAFFEIS, S. Deeptective: Detection of php vulnerabilities using hybrid graph neural networks. In *Proceedings of the 36th annual ACM symposium on applied computing* (2021), pp. 1687–1690.

[39] REGEHR, J., CHEN, Y., CUOQ, P., EIDE, E., ELLISON, C., AND YANG, X. 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] REIS, C., DUNAGAN, J., WANG, H. J., DUBROVSKY, O., AND ESMEIR, S. Browsershield: Vulnerability-driven filtering of dynamic html. *ACM Transactions on the Web (TWEB) 1*, 3 (2007), 11–es.

[41] RIGGER, M., AND SU, Z. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang. 4*, OOPSLA (nov 2020).

[42] RIGGER, M., AND SU, Z. Testing database engines via pivoted query synthesis. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation* (USA, 2020), OSDI'20, USENIX Association.

[43] RYAT. Use after free vulnerability in unserialize. https://hackerone.com/reports/159948, 2016. [Accessed 23-08-2024].

[44] SECURITYSCORECARD. https://www.cvedetails.com/product/128/PHP-PHP.html, 2024.

[45] SEREBRYANY, K. {OSS-Fuzz}-google's continuous fuzzing service for open source software.

[46] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)* (2012), pp. 309–318.

[47] SHARMA, M., YU, P., AND DONALDSON, A. F. 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.

[48] SRIVASTAVA, P., AND PAYER, M. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis* (2021), pp. 244–256.

[49] SU, Y., AND RIGGER, M. Understanding and reusing test suites across database systems. In *Preprint of SIGMOD 2025, International Conference on Management of Data* (2025).

[50] SUN, C., LE, V., AND SU, Z. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications* (2016), pp. 849–863.

[51] TUONG, F., OMIDVAR TEHRANI, M., GABOARDI, M., AND KO, S. Y. Symrustc: A hybrid fuzzer for rust. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (2023), pp. 1515–1518.

[52] W3TECHS. https://w3techs.com/technologies/details/pl-php.

[53] WANG, J., ZHANG, Z., LIU, S., DU, X., AND CHEN, J. {FuzzJIT}:{Oracle-Enhanced} fuzzing for {JavaScript} engine {JIT} compiler. In *32nd USENIX Security Symposium (USENIX Security 23)* (2023), pp. 1865–1882.

[54] WINTERER, D., ZHANG, C., AND SU, Z. Validating smt solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on programming language design and implementation* (2020), pp. 718–730.

[55] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI '11, Association for Computing Machinery, p. 283–294.

[56] YE, G., TANG, Z., TAN, S. H., HUANG, S., FANG, D., SUN, X., BIAN, L., WANG, H., AND WANG, Z. 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.

[57] ZELLER, A., AND HILDEBRANDT, R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering 28*, 2 (2002), 183–200.

[58] ZHAO, Y., WANG, Z., CHEN, J., LIU, M., WU, M., ZHANG, Y., AND ZHANG, L. History-driven test program synthesis for jvm testing. In *Proceedings of the 44th International Conference on Software Engineering* (2022), pp. 1133–1144.

[59] ZHONG, H. Enriching compiler testing with real program from bug report. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (2022), pp. 1–12.