USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

Zipper: Static Taint Analysis for PHP Applications with Precision and Efficiency

Xinyi Wang and Yeting Li, *{CAS-KLONAT, BKLONSPT}, Institute of Information Engineering, Chinese Academy of Sciences and School of Cyber Security, University of Chinese Academy of Sciences;* Jie Lu, *SKLP, Institute of Computing Technology, Chinese Academy of Sciences;* Shizhe Cui, *School of Informatics, The University of Edinburgh;* Chenghang Shi, *SKLP, Institute of Computing Technology, Chinese Academy of Sciences and School of Computer Science and Technology, University of Chinese Academy of Sciences;* Qin Mai, *{CAS-KLONAT, BKLONSPT}, Institute of Information Engineering, Chinese Academy of Sciences and School of Cyber Security, University of Chinese Academy of Sciences;* Yunpei Zhang, *School of Information and Software Engineering, UESTC;* Yang Xiao, Feng Li, and Wei Huo, *{CAS-KLONAT, BKLONSPT}, Institute of Information Engineering, Chinese Academy of Sciences and School of Cyber Security, University of Chinese Academy of Sciences*

https://www.usenix.org/conference/usenixsecurity25/presentation/wang-xinyi

This paper is included in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

ZIPPER: Static Taint Analysis for PHP Applications with Precision and Efficiency

Xinyi Wang[†§], Yeting Li[†§✉], Jie Lu[‡], Shizhe Cui[♮], Chenghang Shi[‡#], Qin Mai[†§], Yunpei Zhang[¶], Yang Xiao[†§], Feng Li[†§], Wei Huo[†§]

[†]{CAS-KLONAT, BKLONSPT}, Institute of Information Engineering, Chinese Academy of Sciences
[§]School of Cyber Security, University of Chinese Academy of Sciences
[‡]SKLP, Institute of Computing Technology, Chinese Academy of Sciences
[#]School of Computer Science and Technology, University of Chinese Academy of Sciences
[♮]School of Informatics, The University of Edinburgh
[¶]School of Information and Software Engineering, UESTC

Abstract

PHP-based web applications constitute a significant portion of the Web infrastructure and are frequently targeted by attackers exploiting taint-style vulnerabilities. While static analysis has emerged as a preferred approach for detecting these vulnerabilities, two major challenges persist: accurately inferring dynamic values from PHP's dynamic features, and efficiently detecting taint vulnerabilities in large-scale applications. This paper presents ZIPPER, a novel static analysis framework that addresses these challenges through two key innovations. First, we introduce a context-sensitive, flow-sensitive value-set algorithm that precisely infers dynamic values by leveraging input validation patterns and framework API characteristics. Second, we implement an efficient, on-demand approach to taint analysis that incorporates object-sensitive and array index-sensitive analyses while maintaining efficiency through sparse data dependency graphs. Evaluation on 429 known taint-style vulnerabilities demonstrates ZIPPER's effectiveness with the highest precision of 68.34% and an impressive recall of 98.14%, outperforming existing approaches. Furthermore, application of ZIPPER to 100 popular PHP applications led to the discovery of 11 previously unknown vulnerabilities, resulting in 6 CVE assignments.

1 Introduction

PHP-based web applications, including popular platforms such as WordPress (content management system) [6] and Wikipedia (online encyclopedia) [1], form a significant part of the Web infrastructure [48]. Their widespread adoption and the sensitive data they host make them prime targets for attackers, increasing the risk of security breaches. These security risks stem from a variety of vulnerabilities, among which taint-style vulnerabilities are particularly concerning. Such vulnerabilities are not only highly prevalent in PHP-based websites but are also frequently exploited by attackers [30].

✉Yeting Li is the corresponding author.

Taint-style vulnerabilities occur when user-supplied data reaches sensitive functions without sufficient validation, potentially leading to various attack vectors such as command injection [2], SQL injection [4], and Cross-Site Scripting (XSS) [3]. These vulnerabilities can have severe consequences, including the exposure of sensitive data and, in more critical cases, system compromise.

To address these security challenges, researchers have proposed numerous detection methods, broadly categorized into dynamic and static approaches. Dynamic approaches [19–21,35,47,49] involve injecting attack payloads into HTTP requests and analyzing responses or application states to identify vulnerabilities, such as detecting XSS through payload reflection. While effective in certain scenarios, dynamic analysis approaches face significant limitations, including manual configuration requirements, incomplete code coverage, potentially resulting in false negatives.

In contrast, static analysis approaches [9, 12, 16, 18, 24, 30, 51, 53] offer comprehensive code coverage and automation capabilities, facilitating seamless integration into development workflows. Consequently, static analysis has emerged as a widely adopted and effective approach for detecting taint-style vulnerabilities in PHP applications.

1.1 Challenges

The effectiveness of static analysis in detecting taint-type vulnerabilities in PHP applications depends on constructing precise and complete call graphs and data dependency graphs. However, PHP's dynamic features make this objective particularly challenging. Furthermore, the large-scale nature of modern PHP applications poses significant efficiency challenges to vulnerability detection.Here, we introduce these two main challenges:

Challenge 1: How to statically infer dynamic values arising from PHP's dynamic features? PHP dynamic features such as dynamic method invocation, variable variables, and dynamic property access, are resolved at runtime. Consider the following code example:

```
1 $methodName=/*dynamic value*/
2 $result = $object->$methodName();
```

The methodName variable contains a dynamic string value that is only determined at runtime, and PHP will invoke the corresponding method on the object using this string as the method name. Without the ability to precisely determine the potential values of methodName during static analysis, an incomplete or imprecise call graph is generated, consequently compromising the accuracy of taint analysis results.

To infer dynamic values, previous research has identified three sources of dynamic values: (1) constant strings in code, (2) external inputs from users, configuration files, databases, etc., and (3) combinations of constant strings and external inputs. Existing methods [16, 30] represent dynamic values using regular expressions. When the dynamic value is a constant string, the regular expression is simply that constant string. For external inputs, existing works use (.*) as a representation. For combinations, the regular expression combines constant strings with (.*). For example, the value of methodName might be "get(.*)", and functions whose names match this regular expression become the target functions.

Although this approach of using regular expressions to represent dynamic values has been widely adopted in existing works, we have identified three key limitations in their value inference process. First, while values may be constants, they are actually context-dependent—the same constant can have different values in different contexts. Existing works lack context sensitivity, leading to imprecise value inference. Second, existing works' simplistic use of (.*) to represent external inputs can result in over-matching when matching function names. While some works address this by adopting a strategy of not matching anything when encountering (.*), this leads to under-matching issues. Third, dynamic values may be derived through complex string operations involving loops and regular expressions. Existing works simply reduce these to (.*), resulting in imprecise or unsound results.These limitations underscore the significant challenge in precisely inferring dynamic values through static analysis.

Challenge 2: How to precisely and efficiently detect taint vulnerabilities in large-scale PHP applications? Precise vulnerability detection necessitates accurate modeling of taint propagation; however, current approaches exhibit several significant limitations. First, their handling of taint propagation for arrays and objects relies on an over-approximation strategy, whereby tainting a single array element results in the entire array being marked as tainted. Second, these approaches fail to adequately address implicit taint flows introduced by PHP's dynamic features, particularly when accessing global variables through the $GLOBALS superglobal array or when utilizing variable variables to access properties, where property names are determined dynamically based on the value of another variable, leading to false negatives in detection. Moreover, implementing precise taint propagation tracking through these elements demands computationally intensive techniques such as alias analysis, which can result in prohibitive analysis times. Furthermore, existing taint propagation algorithms, including IFDS [38], face performance bottlenecks when applied to large-scale applications [10, 22, 28, 29]. Efficiently and precisely taint tracking presents a significant challenge.

1.2 Solutions

Effective Dynamic Value Inference To address Challenge 1, we propose a novel approach based on two key characteristics of PHP's dynamic features. First, dynamic values from user inputs undergo validation checks before use, generating constraints that aid in value determination. Second, complex string operations and data retrieval from databases or configuration files are typically encapsulated within framework APIs. These APIs generate dynamic values while invoking corresponding functions, promoting cleaner application code focused on business logic.

We observe that APIs belonging to the same class share similar dataflow characteristics during invocation. By clustering based on these dataflow patterns, we can group APIs by their respective classes and subsequently identify specific API calls through their class membership analysis. This approach circumvents the need to parse complex string operations or database/configuration file access while maintaining precise dynamic value determination.

For constant value inference, we introduce a context-sensitive, flow-sensitive value-set algorithm that incorporates the aforementioned characteristics to identify user inputs and framework-induced dynamic values. This comprehensive approach enables precise computation of potential dynamic values, significantly improving upon existing methods.

Precise and Efficient Taint Analysis To achieve precise taint propagation tracking, we implemented object-sensitive and array index-sensitive analyses, along with the required alias analysis. For efficient analysis, we developed a sparse data dependency graph and implemented an on-demand taint analysis algorithm that constructs the graph only when taint propagates through function parameters. Similarly, we adapted both the alias analysis and call graph construction to operate on-demand. Since both call graph construction and data dependency graph generation require dynamic values, we further enhanced efficiency by transforming our value-set algorithm for dynamic value computation into an on-demand approach. This comprehensive transformation of core components—including taint analysis, alias analysis, and value-set computation—into on-demand operations significantly improves the overall analysis efficiency while maintaining analytical precision.

1.3 Contributions

We implemented our solutions in a new tool named ZIPPER and evaluated it on 429 taint-style vulnerabilities. The results

demonstrate that ZIPPER successfully identified 421 of these vulnerabilities, achieving a low false negative rate of 1.86% (8/429). The analysis reported 195 false positives, resulting in a false positive rate of 31.25% (195/624). Both false positive and false negative rates outperform existing state-of-the-art approaches. Furthermore, we applied ZIPPER to 100 popular PHP applications, discovering 11 previously unknown vulnerabilities, which led to the assignment of 6 CVE identifiers.

The contributions of this paper are summarized as follows:

1. We present ZIPPER, a taint vulnerability detection tool that implements context-sensitive and flow-sensitive precise value-set analysis to handle PHP's dynamic features. The tool enhances analysis efficiency through on-demand alias analysis, call graph construction, and data dependency graph generation.
2. ZIPPER demonstrates superior performance compared to existing tools in terms of both false positives and false negatives, as well as in its capability to detect new vulnerabilities.
3. We release the source code of ZIPPER on our website [5] to facilitate future research.

2 Motivating Example

This section introduces a motivating example to illustrate how PHP's features facilitate flexible application development while presenting challenges for static analysis.

2.1 Example Overview

The artificially constructed example in Figure 1 (d) and (e) demonstrates a basic MVC (Model-View-Controller) application implemented in PHP. It processes requests and performs data interactions through the coordinated efforts of key components, such as the `Dispatcher`, `SliderController`, and `SliderModel`.

The `Dispatcher` class (line 2 in Figure 1 (d)) functions as the front controller, responsible for handling HTTP requests and dynamically loading and executing the corresponding controller classes based on URL parameters. Its primary functions include: ⓐ **Parsing URL Parameters.** Extracting the controller name and method name from (`$_GET['controller']` and `$_GET['action']`) (lines 9-10). ⓑ **Dynamic File Inclusion.** Dynamically generating the file path based on controller name and using `include` statement to load the controller file (line 15). ⓒ **Controller Instantiation and Method Invocation.** Dynamically instantiating the controller class and invoking the action method (line 16).

The `SliderController` in Figure 1 (e) is responsible for handling user input and interacting with the model, which performs data operations. Its functionalities include: ⓐ **Handling Slider-Related Requests.** Processing requests from the frontend through public methods (e.g., `info()` and `update()`), which are dynamically invoked by the `Dispatcher`. ⓑ **Interacting with SliderModel.** Delegating the `SliderModel` to perform data operations and provide it with user input when necessary, which allows it to dynamically construct SQL queries using variables (e.g., `"INSERT ... $data ..."` on line 4 of `SliderModel.php`).

The `Uri` class provides host-related information to interact with other components. It is instantiated by both `Dispatcher` and `SliderController` using server data, then stored globally for dynamic URL generation and displaying welcome messages. The `TestController` class, a basic controller for testing, validates controller loading and method invocation.

2.2 PHP Features

PHP provides several key features that support flexible development of applications. These features in Figure 1 include:

Dynamic File Inclusion. PHP's file inclusion allows files to be loaded flexibly during program execution through the `include` statement (e.g., `include "$controllerClass.php"` in line 15 of Figure 1 (d)). This enables the application to load different controller files based on user input, making it more modular and configurable.

Dynamic Call. Method Call in PHP can be dynamic, where method names are stored in variables and called using the `->$action()` syntax as seen in line 16 of Figure 1 (d). This powerful feature allows flexible routing, enabling the application to choose methods based on dynamic values.

Dynamic Object Creation. PHP also supports dynamic object creation using the syntax `new $controllerClass()`, where the class name is stored in a variable (e.g., in line 16 of Figure 1 (d)). This feature allows the application to create different types of objects based on dynamic values, supporting flexible factory patterns and dynamic controller instantiation based on user requests.

Global Data Sharing. The `$GLOBALS` array in PHP provides a way to share data across different parts of the application. As demonstrated by `$GLOBALS['gUri']` in line 12 of Figure 1 (e), it allows data like the URI to be accessed from any scope within the application, facilitating data sharing between different components without explicit parameter passing.

All these features' behaviors depend on the dynamic values. Therefore, a precise value inference is essential for analyzing these features.

2.3 Challenges of Static Analysis Due to PHP Features

In this section, we illustrate the limitations of existing approaches through two representative taint vulnerability detection cases.

Vulnerability Propagation Paths. The example illustrates two distinct vulnerability propagation paths. The first (XSS) begins in the `info` method of `SliderController` (line 6), where user input is passed to `$uri->host`, and its alias is

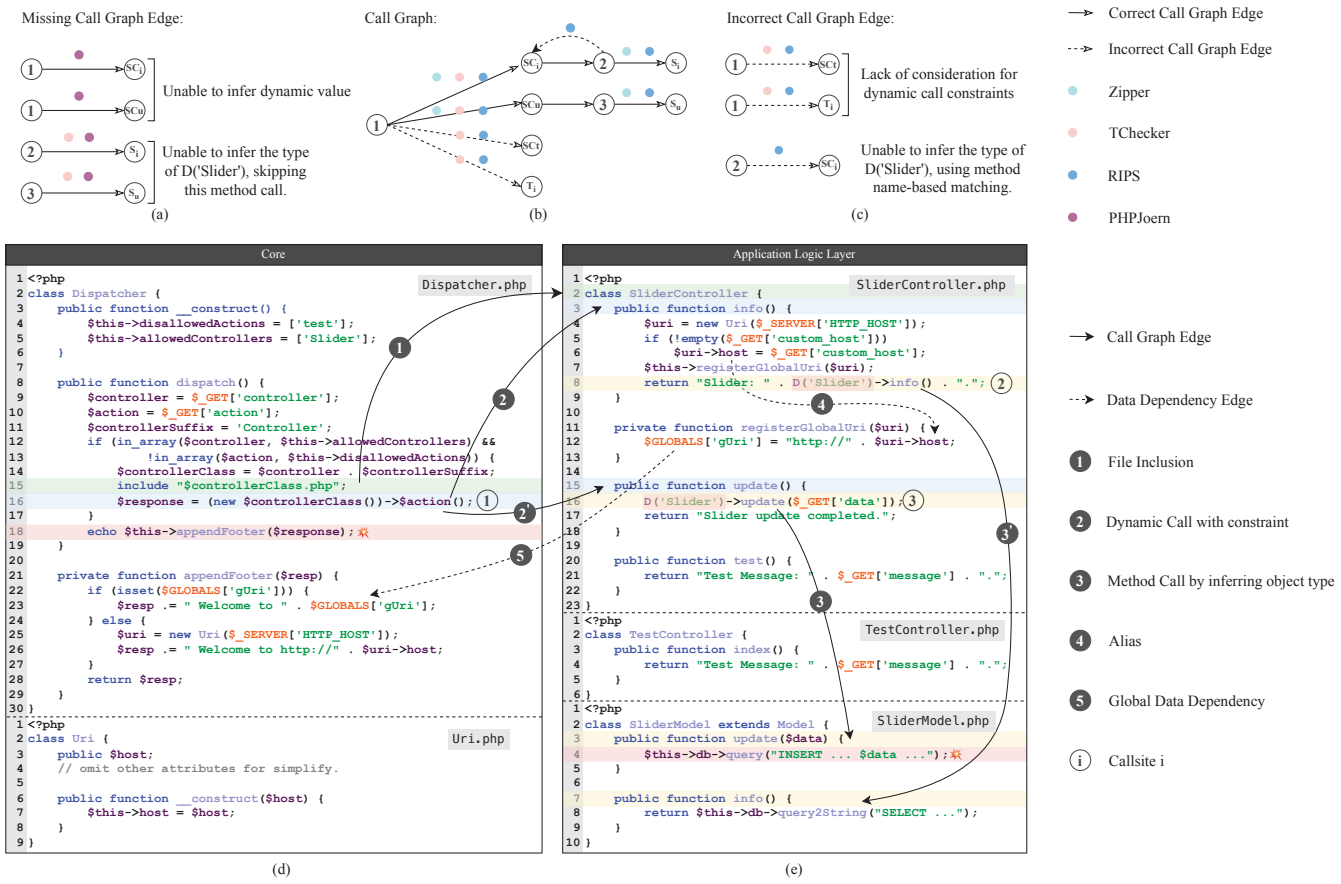

Figure 1: Motivation Example. Subfigures (d) and (e) depict an MVC-based application with call and data dependency edges, while (a), (b), and (c) show false negative and false positive analyses of call graphs generated by four static analysis tools.

then assigned to the global variable `gUri` via `$GLOBALS` (line 12). The `Dispatcher` later prints `gUri` wrapped by `appendFooter`, causing an XSS vulnerability due to the lack of sanitization. To detect this, call edges ❶ and ❷, along with data dependency edges ❹ and ❺ in Figure 1, are crucial. The second (SQL injection) arises in `update` method of `SliderController`, where user input is passed to `D('slider')`'s `update` method (call edge ❸). Since this input is directly included in an SQL query without sanitization, it leads to SQL injection.

While PHP features offer developer convenience, they create significant challenges for static analysis. As illustrated in Figure 1, we compare ZIPPER with three static analysis tools (TChecker [30], RIPS[1] [16], and PHPJoern [12]) in their handling of PHP features through a representative example. The analysis showed that existing approaches have notable shortcomings in both value inference and taint propagation.

Challenges in Static Analysis. *Challenges of dynamic value inference*:ⓐ **Dynamic values from constants**: Constants can be used across multiple function calls and handled by various PHP features. For example, in line 4 of `Dispatcher.php` (Figure 1(d)), a constant is stored in a dynamic array, assigned to an object property, and later used in the `dispatch()` method. TChecker fails to infer the value due to its partial inter-procedural analysis, which cannot reach the constructor of `Dispatcher` (line 3). RIPS also fails due to its intra-procedural analysis. Furthermore, dynamic values can have different values in different calling contexts. ⓑ **Dynamic values from external inputs or complex string operations**: Existing methods use (.*) to model these situation, but this leads to imprecision in handling dynamic features. For instance, in line 16 of `Dispatcher.php`, where a dynamic object is created depending on the value of `$ControllerClass`, both TChecker and RIPS model it as `(.*)Controller`, causing all classes ending with `Controller` to be instantiated.

Challenges of taint propagation: ⓐ Existing methods often adopt on over-approximation strategy when tracking taint propagation through objects and arrays. For example, TChecker's object-insensitive taint analysis leads to a false positive: after tainting the `$uri->host` at line 6 of `SliderController.php`, it incorrectly propagates the taint

[1]Following TChecker, we refer to RIPS' proprietary version as RIPS-A [18]. We are unable to apply RIPS-A because its source code is not publicly available.

to the `host` property at line 26 in `Dispatcher.php`. ⓑ Additionally, current methods fail to account for implicit data flows introduced by dynamic features, such as the data flow edge ❺ in Figure 1 (d, e) that is propagated through `$GLOBALS` rather than conventional variable assignments or parameter passing. Achieving accurate taint analysis without excessive computation time remains a significant challenge.

Static Analysis Tool Comparison. Figure 1 presents three call graphs (a, b, c) generated by four static analysis tools, where $T/SC/S$ represents TestController/SliderController/SliderModel respectively, with method name abbreviations as subscripts (e.g., SC_s for SliderController::show). ⓐ **False Negatives.** While RIPS identified the SQL injection vulnerability, it may lead to false positives. The missed detections are caused by two main issues: first, incomplete call graphs. PHPJoern misses the call edges in Figure 1 (a) because it doesn't handle dynamic calls. TChecker and PHPJoern both struggle with complex framework semantics from `D('slider')`, causing type inference failures and hindering SQL injection detection. Second, these tools lack implicit data flow tracking, missing data flow edge ❺ in Figure 1, which leads to missing the XSS vulnerability. ⓑ **False Positives.** Figure 1(c) shows that both RIPS and TChecker generate incorrect call edges from ① to SC_t and T_i due to partial class name matching, leading to the instantiation of all classes ending with `Controller`. Additionally, TChecker connects call edges to all methods in the class because the `$action` variable comes entirely from user input. RIPS generates a call edge from ② to SC_i due to type inference failure, incorrectly associating all methods named `info`. All these incorrect call edges may cause potential false positives. Additionally, even though TChecker can generate correct call graphs, its object-insensitive taint analysis leads to a false positive on line 26 of `Dispatcher.php`.

Key Observations. ⓐ **Constraints on External Input.** We observe that dynamic values from external inputs are often constrained by program conditions that limit their possible ranges. For example, in lines 12-13 of `Dispatcher.php`, the `if` statements constrain $controller \in \{\text{"slider"}\} \wedge action \notin \{\text{"test"}\}$, which can help ZIPPER determine their value sets. ⓑ **Characteristics for Same API Invocation.** APIs within the same class exhibit similar data flow characteristics during invocation. For example, in lines 8 and 16 of `SliderController.php`, the same receiver object is used to invoke the `info` method. So we can cluster API invocations based on data flow characteristics to jointly infer the receiver object's type. ⓒ **Sparse Access on Sensitive Parts.** Access to object properties, array elements and implicit data flows is far less frequent than for normal variables. Based on this observation, we prebuilt the global data dependency on a sparse graph to accelerate construction speed and defer alias calculation until the actual usage points, enabling efficient and precise alias analysis.

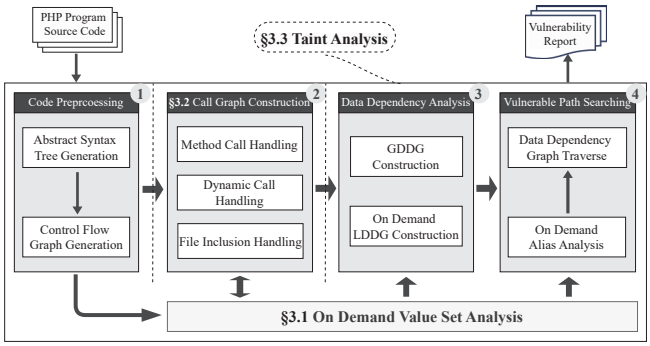

Figure 2: Overview of ZIPPER

2.4 Threat Model

In this paper, we focus on taint-style vulnerabilities in web applications and WordPress plugins developed using PHP. Our threat model assumes that an attacker can provide malicious input which, due to missing validation or sanitization, propagates to sensitive functions, leading to vulnerabilities like command injection, SQL injection, or Cross-Site Scripting. These attacks typically occur through web interfaces and can result in sensitive data disclosure or system compromise. For our approach, we make several assumptions about the target codebase: our tool requires complete access to the source code, assumes the code follows recognizable patterns (such as wrapper functions for string operations), is not crafted to attack the tool (e.g., with adversarial constructs to exhaust resources), and is not obfuscated to mask its true logic.

3 Methodology

Figure 2 illustrates the value-set analysis framework for PHP applications within the context of ZIPPER, a precise and efficient taint analysis tool for vulnerability detection. The framework accurately models diverse PHP features to effectively track value flows. To ensure high efficiency, it adopts an on-demand strategy that minimizes redundant computations.

ZIPPER starts by building a precise call graph that handles complex and dynamic language semantics including method calls, dynamic calls, and file inclusion. Subsequently, the taint analysis is conducted in two phases for efficiency: (1) IFDS-based global data dependency tracking, and (2) alias-aware vulnerable path searching.

Section 3.1 introduces the core idea of our on-demand value-set analysis (VSA). Section 3.2 demonstrates how to support on-the-fly construction for call graph. Finally, Section 3.3 elaborates on the design of taint analysis.

3.1 Value-Set Analysis

VSA statically determines the possible values of a program variable, which can include integers, strings, arrays, class ob-

Program	P	:=	F^+
Function	F	:=	$f(\$x, \$y, \ldots)\{S;\}$
Statement	S	:=	$\$x = new\ \tau \mid \$x = e$
			$\mid ArrStore(\$x, \$y, \$z) \mid ObjStore(\$x, \$f, \$y) \mid VarStore(\$x, \$y)$
			$\mid f(\$v_1, \$v_2, ...) \mid return\ \$x \mid if(\$x)\{S_1;\}else\{S_2;\} \mid S_1; S_2$
Expression	e	:=	$\$y \mid UaryOp\ \$y \mid BinaryOp(\$y, \$z)$
BinaryOp		:=	$ArrLoad \mid ObjLoad \mid + \mid -\ldots$
UaryOp		:=	$VarLoad \mid \neg \mid -\ldots$

Figure 3: A simplified language.

Locations $\ell \in L$	Variables $\$x, \$y, \cdots \in V$
Fields $f \in F$	Integers $i \in \mathcal{V}_{int}$
Strings $s \in \mathcal{V}_{str}$	Arrays $a \in \mathcal{V}_{arr}$
Objects $o \in \mathcal{V}_{obj}$	Symbolic values $\mathcal{V} = \mathcal{V}_{int} \cup \mathcal{V}_{str} \cup \mathcal{V}_{arr} \cup \mathcal{V}_{obj}$
Environment $\mathbb{E} := V \mapsto 2^{\mathcal{V}}$	Array state $\mathbb{A} := \mathcal{V}_{arr} \times (\mathcal{V}_{int} \cup \mathcal{V}_{str}) \mapsto 2^{\mathcal{V}}$
Object state $\mathbb{O} := \mathcal{V}_{obj} \times F \mapsto 2^{\mathcal{V}}$	

Figure 4: Analysis domains.

jects, or boolean values (represented as integers).When a variable's value consists of both concrete (e.g., "get") and symbolic values (e.g., "$v"), we employ regular expressions (e.g., "get (.*)") for its representation.

3.1.1 Basic Definition

A Simplified Language. For illustration purpose, we formalize our analysis with a simple language in Figure 3, which uses different notations from PHP. Each loop on the control flow graph is unrolled once. A new statement takes the form $\$x = new\ \tau()$, where τ refers to types such as integer, string, array, or object. Unlike PHP, where a new statement creates an object of type τ and invokes the constructor function $\tau :: \_\_construct()$, our simplified language separates these two operations and requires an explicit call to the constructor for easier formulation.

In PHP, $\$\$x$ is a *variable variable*, meaning that the value of $\$x$ is used as the name of another variable. For example, if $\$x$ holds a string "a", the statement $\$\$x = \$y$ stores the value of $\$y$ in the variable $\$a$, while $\$y = \$\$x$ loads the value from $\$a$ and assign it to $\$y$, denoted as $\cdots = VarLoad\ \$x$ and $VarStore(\$x, \ldots)$, respectively. Binary operators include arithmetic operations as well as read operations on arrays and objects, denoted as *ArrLoad* and *ObjStore*, respectively. Moreover, $ArrStore(\$x, \$y, \$z)$ represents a write operation to an array, corresponding to $\$x[\$y] = \$z$ in PHP, while $ObjStore(\$x, f, y)$ indicates a write operation to an object property f, corresponding to $\$x\text{->}f = \$y$. This language handles only simple function calls of the form $f(\$x, \$y, \ldots)$, assuming a prebuilt call graph is available. Section 3.2 illustrates how to enable on-the-fly construction to support more complex PHP language features.

Analysis Domains. Figure 4 presents the analysis domains. The environment $\mathbb{E}$ tracks the *symbolic value set* for each variable, where a *symbolic value* (or value) can be of type integer ($\mathcal{V}_{int}$), string ($\mathcal{V}_{str}$), array ($\mathcal{V}_{arr}$), and (class) object ($\mathcal{V}_{obj}$). Additionally, $\mathbb{A}(a \times k) \mapsto \{v\}$ means that the element of array a indexed by k may be symbolic value v, where k can

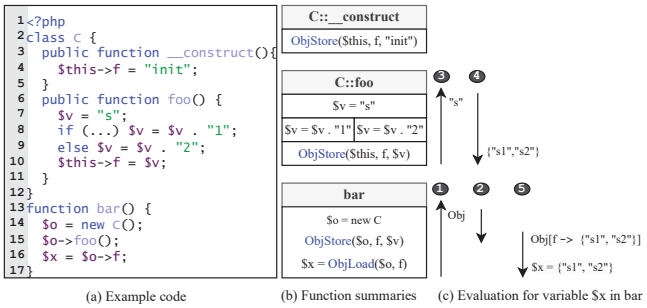

Figure 5: Example for value set analysis.

be an integer or a string. Similarly, $\mathbb{O}(o \times f) \mapsto \{v\}$ denotes that the field (i.e., property) f of object o may hold value v.

In essence, when processing an on-demand VSA query at program point p, the analysis first performs a backward traversal along the control flow graph starting from p. During the graph traversal, the set of relevant program statements is collected concisely using function summaries (Section 3.1.2). Later, these symbolic summaries are instantiated to obtain the analysis results (Section 3.1.3).

3.1.2 Function Summary Generation

Similar to previous work [16–18, 51], We construct a symbolic *function summary* to capture the semantics of a method, comprising *block summaries* for each of its basic blocks. A block summary represents its program statements using two forms: *symbolic expressions* and *symbolic side effects*.

Symbolic Expression. A symbolic expression e has a nested, tree-like structure that effectively models program expressions, such as binary operations on its sub-expression(s). A leaf node is a program variable defined outside the current block, which may also be a parameter. By layering operators through nesting, a symbolic expression concisely captures statements along with their data dependencies.

Symbolic Side Effect. As the name suggests, a symbolic side effect captures program statements that produce side effects, such as *VarStore*, *ArrStore*, and *ObjStore*, which modify $\mathbb{E}$, $\mathbb{A}$, and $\mathbb{O}$, respectively. Side effects can also contain a symbolic expression as a sub-expression. For instance, $ObjStore(e_1, f, e_2)$ indicates that the symbolic objects evaluated from expression e_1 are updated with values obtained from evaluating e_2.

Summary Generation. A block summary maps each variable x to a set of symbolic expressions S_{expr} and a set of symbolic side effects S_{effect}. If $e \in S_{expr}$, it indicates that the evaluation result of e–a set of symbolic values–contributes to the value set of v. Similarly, each symbolic side effect in S_{effect} may perform write operations to the value of v. As discussed above, the types of side effect operation can be *ArrStore*, *ObjStore*, or *VarStore*. For example, $ObjStore(\$x, f, \ldots)$ indicates a write operation to the property f of objects refer-

$$\frac{\ell : \$x = new\ \tau()}{\mathbb{E}(\$x) = \{v_\tau\}} \quad [\textsc{New}]$$

$$\frac{\$x = e}{\mathbb{E}(\$x) = eval(e)} \quad [\textsc{Assign}]$$

$$\frac{e ::= \$y \quad v \in \mathbb{E}(\$y)}{v \in eval(e)} \quad [\textsc{VarExp}]$$

$$\frac{e := UaryOp\ e' \quad v \in eval(e') \quad v' = UaryOp^*\ v}{v' \in eval(e)} \quad [\textsc{UnaryExp}]$$

$$\frac{e := BinaryOp(e_1, e_2) \quad v_1 \in eval(e_1) \quad v_2 \in eval(e_2) \quad v = BinaryOp^*(v_1, v_2)}{v \in eval(e)} \quad [\textsc{BinaryExp}]$$

$$\frac{ObjStore(e_1, f, e_2) \quad o \in eval(e_1) \quad v \in eval(e_2)}{v \in \mathbb{O}(o \times f)} \quad [\textsc{ObjStore}]$$

$$\frac{ArrStore(e_1, e_2, e_3) \quad a \in eval(e_1) \quad k \in eval(e_2) \quad v \in eval(e_3)}{v \in \mathbb{A}(a \times k)} \quad [\textsc{ArrStore}]$$

$$\frac{VarStore(e_1, e_2) \quad s \in eval(e_1) \quad \text{let } \$x \text{ be the variable named } s \quad v \in eval(e_2)}{v \in \mathbb{E}(\$x)} \quad [\textsc{VarStore}]$$

Figure 6: Rules for value set analysis. A symbol $*$ denotes an abstract version of an operation.

enced by $\$x$.

To ensure flow sensitivity, ZIPPER constructs block summaries for each basic block by traversing its statements in control flow order. These block summaries are then composed to create function summaries. When a function is first encountered, its summary is generated and cached. For a call site, the corresponding function summary is recursively constructed. Mutual recursion between functions is handled by iteratively merging their symbolic summaries until a fixed point is reached.

Example 3.1. Figure 5(a) and (b) depict a code snippet and the corresponding function summaries for three methods. Specifically, the constructor for class `C`, i.e., `C::__construct`, contains a single block summary with an *ObjStore*, while `C::foo` retains four block summaries. Lastly, we incorporate the summaries of these two methods to form the summary for `bar`. Notably, the side effect $ObjStore(\$o, f, "init")$ from the constructor is hidden by $ObjStore(\$o, f, \$v)$ from `C::foo` to model flow-sensitivity.

3.1.3 On-Demand Value Set Analysis

In this subsection, we first illustrate an on-demand, intraprocedural analysis, and then show how to perform it interprocedurally.

Intraprocedural Analysis. When processing a query for variable $\$x$ at program point p, a backward traversal starting from p along the control flow graph is performed to collect symbolic expressions and side effects. These symbolic summaries are then interpreted using rules in Figure 6. Rule [NEW] creates a singleton set containing a symbolic value of type τ. Rule [ASSIGN] evaluates expression e using function *eval* and updates the value set of the variable $\$x$.

The auxiliary function *eval* utilizes three kinds of expressions (as defined in Figure 3), namely [VAREXP], [UNARYEXP], and [BINARYEXP] to evaluate variables, unary expressions, and binary expressions, respectively. Here, an operator with $*$ denotes its abstract version. For instance, the abstract binary operator $ArrLoad^*(v_1, v_2)$ retrieves values held by the element of array v_1 indexed by v_2, i.e., $\mathbb{A}(v_1 \times v_2)$. Additionally, rules [OBJSTORE], [ARRSTORE], and [VARSTORE] handle three types of symbolic side effects. For PHP built-in functions, similar to the rules above, ZIPPER evaluates them based on their semantics. For these rules, kill operations (also known as strong updates) are performed following previous work [27].

Example 3.2. Continuing from Example 3.1, we now demonstrate how to determine the values of `$x` at line 16. Specifically, we interpret the method summary of `bar`, as illustrated in Figure 5 (c). During this interpretation, since the variable $\$v$ is defined in `foo`, we further evaluate the symbolic expressions retrieved from `foo`'s summaries to resolve its values, which are $\{"s1", "s2"\}$. These values are subsequently stored in the field f of $\$o$ and eventually loaded to assign $\$x$.

Interprocedural Analysis. To support interprocedural analysis, we annotate the queried variable with a call stack. This is crucial when the query depends on the value of a formal parameter p in the current method, which receives values from the caller's actual arguments. In such cases, the query is propagated to the corresponding call site, facilitating the evaluation of parameter p. ZIPPER handle variables not defined in the current scope, such as global variables, in the same way.

3.1.4 Inference Phase

The analysis presented so far may fail to determine the value set of a variable when there are data dependencies on user input values or complex string operations. We further propose an inference phase to tackle such cases to enable a comprehensive analysis.

Inference by Constraints. When the result of VSA is an empty set or contains a string, we first apply this rule. The idea behind this rule is to narrow down the variable's value range by collecting string constraints on the variable along the path. To achieve this, we follow three steps: ⓐ Collect Path Constraints. We start from the basic block containing the evaluated variable and traverse the control dependency graph to collect path constraints in symbolic expression form that reach this basic block. ⓑ Extract Related Constraints. For each symbolic expressions, we apply VSA on them and filter expressions that involve the evaluated variable. ⓒ Transform Constraints to Regular Expression. For each remaining symbolic expression, we convert it into an automata based on its semantic, such as $x \in \{s_1, s_2, ...\}$ will translate to $s_1|s_2|...$.

Finally, all such automata are intersected to produce a regular expression that conservatively approximates the value space of the variable under the collected constraints.

Example 3.3. When evaluating `$action` variable at line 16 of `Dispatcher.php` in Figure 1(d), the result of VSA is an empty set because its value comes from user input. To address this, we collect the path constraints reaching this basic block (lines 12-13). After evaluating by VSA and filtering, we obtain the constraint $not(in\_array(action, \{'test'\}))$. Based on its semantics, we generate an automaton that does not match the `test` string and convert it into a regular expression.

Inference by Characteristics. When value inference fails and the variable to be evaluated is the receiver object of a method call, this rule is applied. The rule is based on our observation that code files with similar functionalities (e.g., Controller files) often contain receiver objects of the same type with similar characteristics. These receiver objects may share similar names or be assigned by similar function calls, allowing us to cluster them based on these features and infer their types collectively. To achieve this, we first collect all method calls from folder containing the receiver object. For each receiver object, we traverse the control flow graph to collect its data flow characteristics on block summaries, including its name, assignment information (such as variable names, function names and arguments). We then cluster those with same data flow characteristics. For each cluster, ZIPPER identifies class definitions that include all method names found within that cluster. Finally, a fake object with the inferred type is created to facilitate method call resolution. Notably, when multiple classes define methods matching names in a cluster, the rule may introduce spurious call edges, resulting in false positives, as discussed in Section 4.2.

Example 3.4. In `SliderController.php` of Figure 1 (e), line 8 and line 16 contain two method calls and their receiver objects have identical data flow characteristics: the same function name `D` and the argument `"Slider"`. Since ZIPPER clusters them together and identifies that only the `SliderModel` class contains both the `update` and `info` methods, it successfully infers their type as `SliderModel`.

3.2 Call Graph Construction

This subsection demonstrates how we construct a precise call graph on the fly during value set analysis. In PHP applications, a function can be invoked in several ways:

- *Method call* (Section 3.2.1) in the form of `$x->m(...)`, where the member function m in x's class definition is invoked.
- *Dynamic call* (Section 3.2.2) based on callable types such as variable functions [2].

[2] https://www.php.net/manual/en/functions.variable-functions.php

$$\frac{\ell : \texttt{\$x->m(...)} \quad o \in \mathbb{E}(\$x) \cap \mathcal{V}_{obj} \quad m' = Resolve(o, m)}{\ell \xrightarrow{call} m'} \quad \text{[MCALL]}$$

$$\frac{\ell : \texttt{\$x(...)} \quad s \in \mathbb{E}(\$x) \cap \mathcal{V}_{str} \quad m = Resolve(s)}{\ell \xrightarrow{call} m} \quad \text{[DCALL-STR]}$$

$$\frac{\ell : \texttt{\$x(...)} \quad o \in \mathbb{E}(\$x) \cap \mathcal{V}_{obj} \quad m = Resolve(o, "\_invoke")}{\ell \xrightarrow{call} m} \quad \text{[DCALL-OBJ]}$$

$$\frac{\ell : \texttt{\$x(...)} \quad a \in \mathbb{E}(\$x) \cap \mathcal{V}_{arr} \quad o \in \mathbb{A}(a \times 0) \cap \mathcal{V}_{obj} \quad s \in \mathbb{A}(a \times 1) \cap \mathcal{V}_{str} \quad m = Resolve(o, s)}{\ell \xrightarrow{call} m} \quad \text{[DCALL-ARR]}$$

$$\frac{\ell : \texttt{include \$x} \quad s \in \mathbb{E}(\$x) \cap \mathcal{V}_{str} \quad p = PHPFile(s) \quad m = EntryFunc(p)}{\ell \xrightarrow{call} m} \quad \text{[INCLUDE]}$$

Figure 7: Rules for call graph construction.

- *File inclusion* (Section 3.2.3) utilizes the PHP operator `include` to open a specified file, invoking its top-level functions.

Given an invocation statement, ZIPPER identifies the corresponding call targets by utilizing the results of VSA. For example, to resolve the target methods of the call statement `$x->m(...)`, it is necessary to determine the possible values that variable `x` may hold. However, since VSA also relies on (interprocedural) control flow information provided by the call graph, an iterative resolution process is employed to address this mutual dependency.

This process begins at the program's entry point, progressively resolving new reachable functions through invocation statements and incorporating additional callsites. The iteration continues until a fixed point is reached, where no further reachable methods can be discovered. Figure 7 presents a set of rigorous rules for call graph construction.

3.2.1 Method Call

For ease of discussion, we introduce an auxiliary function, $Resolve(o, s)$, which resolves the method named s based on object o's class definition. When o is not specified, ZIPPER searches the target method named s in the global namespace.

Given a method call statement, ℓ: `$x->m(...)` Rule [MCALL] is employed to determine the potential callees, which requires the value set of `$x`. For each symbolic object o held by `$x`, method m' is resolved based on o's class definition.

3.2.2 Dynamic Call

A dynamic call statement takes the form of ℓ: `$x()`, where call targets depend on the runtime value of variable `$x`. This behavior is modeled using three rules: [DCALL-STR], [DCALL-OBJ], and [DCALL-ARR].

Rule [DCALL-STR] resolves functions named s by invoking $Resolve(s)$, where s is a symbolic string value held by `$x`.

Rule [DCALL-OBJ] addresses the scenario where a symbolic object o is referenced by `$x`, considering the magic method `__invoke` in o's class as a potential call target.

Rule [DCALL-ARR] applies when `x` holds an array a, interpreting the first and second elements of a as a class object and a method name, respectively, to resolve the callee m.

Notably, built-in functions to facilitate callback mechanisms like `call_user_func` [3] can be handled in a similar way by translating `call_user_func($callback, ...)` to `$callback()`.

3.2.3 File Inclusion

In PHP, the `include` operator is used to insert the content of one PHP file into another during script execution. It allows code reuse and better modularity by separating logic into different files. If the specified file p is found, its code is included and executed. For convenience, we call the evaluated code the *entry function* of p. Let us introduce two more functions to facilitate the formalism:

- $PHPFile(s)$ gives the PHP file with name s.
- $EntryFunc(p)$ returns the entry function of a PHP file p.

We then utilize rule [INCLUDE] to simulate this process. For a symbolic string s held by `x`, we search the PHP file named s, then a call edge from line callsite ℓ to the entry function of the matched file.

3.2.4 Extension

The rules in Figure 7 assume ZIPPER is able to determine the exact value for a string variable s. Following previous work [16, 30], when this assumption is broken (e.g., we may only know the prefix of s), we perform fuzzy matching through regular expressions to resolve the function or file candidates.

3.3 Taint-style Vulnerability Checking

The program under analysis is represented as a data dependency graph (Section 3.3.1), and alias-aware taint analysis is formalized as a graph reachability problem [9, 12, 15, 41–43, 46] (Section 3.3.2).

3.3.1 Data Dependency Graph

The data dependency graph (DDG) consists of a Global DDG (GDDG), which models the data dependencies among global variables, and Local DDGs (LDDGs), which are generated on demand for individual methods during graph traversal to capture data flows between local variables within functions. Here, static properties of classes are also considered as global variables.

GDDG. In PHP, the lifetimes of global variables span the entire execution of the script. Global variables can be accessed within functions either via the `global` keyword or the `$GLOBALS` superglobal array, while static properties of a class can be accessed using the `::` operator.

To ensure a precise and efficient taint analysis, ZIPPER first analyzes the def-use relationships for global variables, and then builds the GDDG to facilitate taint propagation involving these variables. For efficiency, we construct a sparse interprocedural control graph [37] including only statements that write or read global variables. Subsequently, an IFDS-based reaching definition analysis [34, 38] is performed to construct the GDDG.

LDDG. When taint analysis reaches a new function for the first time, a corresponding LDDG is constructed on demand, effectively tracking the def-use relationships for local variables. This process utilizes the VSA to handle the dynamic features in PHP such as variable variables and the `extract()` function.

3.3.2 Graph Traversal

As is standard practice, the taint analysis takes a set of sources, sinks, and sanitizers as input [11]. A taint propagation process starts from each source node, and traverses the DDG. When reaching a call site, the propagation steps into all possible callees, with the corresponding parameters tainted. Upon reaching a sink, a potential vulnerable path is built.

On-Demand Alias Analysis. To precisely track data dependencies involving heap data—specifically, write and read operations on object properties and array elements, as detailed in Section 3.1—ZIPPER employs a demand-driven alias analysis during taint checking. When taint propagates to an object property or an array element, an alias query is issued to traverse the control flow graph and identify write operations targeting the aliased memory, guided by the results of the VSA.

Filtering Infeasiable Paths. For each identified vulnerable path p, ZIPPER gathers the conditions along p. It then applies VSA to verify the feasibility of these conditions, filtering out all infeasible paths to enhance precision.

Context Sensitivity. In the process of forward taint propagation, we can naturally maintain a function call stack, which enables us to invoke VSA in a context-sensitive manner.

Example 3.5. Revisiting our motivating example in Figure 1, we first construct a sparse ICFG with statements manipulating global variables (lines 22-23 in Figure 1 (d) and line 12 in Figure 1 (e)), and build the GDDG, connecting data flow edge ❺. We then begin analysis from taint source at line 6 in `SliderController.php`. Since `info()` is encountered for the first time, we build its LDDG, propagating the taint to `$uri->host` on the same line. As an object property, alias

[3] https://www.php.net/manual/en/function.call-user-func.php

analysis is triggered, and we follow the ICFG to identify its alias at line 12 in the same file. Similarly, we construct an LDDG for `registerGlobalUri` and propagate the taint through it and edge ❺ to `appendFooter` in `Dispatcher.php`, ultimately reaching the sink at line 18, where an XSS vulnerability is detected.

4 Evaluation

We have implemented the ZIPPER prototype with over 11,365 lines of Scala code to detect three common types of taint-style vulnerabilities: Cross-Site Scripting (XSS), SQL injection, and command injection. Our implementation integrates PHP-Parser [36] for AST generation, Joern's php2cpg module [54] for CFG construction, and leverages Heros [14] framework for IFDS-based reaching definition analysis, where we implemented custom flow functions while reusing Heros' generic solver infrastructure. To evaluate the effectiveness of ZIPPER, we formulated the following research questions (RQs):

- **RQ1**: How does the performance of ZIPPER compare to state-of-the-art (SOTA) approaches? (§4.2)
- **RQ2**: Does ZIPPER benefit from the application of augmentation techniques? (§4.3)
- **RQ3**: Can ZIPPER find previously undiscovered vulnerabilities in popular real-world applications? (§4.4)

4.1 Experiment Setup

Evaluation Benchmarks. To ensure a fair and unbiased evaluation of the performance of various tools, we adopted a benchmark dataset consisting of 15 PHP applications from prior works [7–9, 16, 18, 30] , using the same versions that were tested in these studies. When an application appeared in multiple studies, we selected the more recent version for our evaluation. The statistics for these applications are provided in Appendix A.1. Among these 15 applications, five are popular and widely-used, such as WordPress and Joomla, each having garnered over 1,000 stars on GitHub.

Compared Tools. We compare ZIPPER with three up-to-date academic and open-source PHP static analysis tools, namely TChecker [30], RIPS [16], and PHPJoern [12, 53, 54]. All the compared tools are the latest versions. It should be noted that a comparison with other taint analysis tools, such as ChainSaw [8] and RIPS-A [18], is not possible, as they are not open-source, or with NAVEX [9], due to the lack of critical function implementations in its codebase.

Evaluation Metrics. To assess the effectiveness of ZIPPER, the following evaluation metrics are employed:

- **Precision**: The ratio of the number of true positives to the total number of the reported vulnerabilities, which includes both true positives and false positives.
- **Recall**: The ratio of the number of true positives to the total number of all the real vulnerabilities, which includes both true positives and false negatives.

Table 1: **The Overall Evaluation Results of RQ1.** TP/FP/FN stands for true positives/false positives/false negatives respectively. The best value in a column is highlighted in bold.

Approach	TP	FP	FN	Precision	Recall	Time
TChecker	71	150	358	32.13%	16.55%	**11m26s**
RIPS	245	**125**	184	66.22%	57.11%	13m12s
PHPJoern	285	1882	144	13.15%	66.43%	144m27s
ZIPPER	**421**	195	**8**	**68.34%**	**98.14%**	30m45s

Ground Truth. The ground truth was constructed through three stages: candidate collection, manual filtering, and dynamic validation. Candidates originated from four tools (ZIPPER, TChecker, RIPS, PHPJoern) and 60 CVEs, deduplicated by vulnerability type, file path, and sink line number (2,486 total). Manual filtering of 2,073 tool-derived cases excluded false positives, yielding 311 candidates for final validation. Proof-of-concept exploits confirmed 429 true positives (413 CVE-validated + 16 tool-derived) across 15 PHP applications. Appendix A.2 provides a comprehensive breakdown of the construction process, with application-specific vulnerability distribution detailed in Table 4.

Configurations. Experiments were conducted on a machine with an Intel i9-13900KF @ 3.00 GHz, 64GB RAM, and Windows 10. Baselines were configured as in their original papers, with consistent definitions of sources, sinks, and sanitization rules.

4.2 RQ1: Comparison to State-of-the-Art

Overall Results. Comparative analysis (Table 1) demonstrates ZIPPER's exceptional performance across key detection metrics. With 421 true positives and only 8 false negatives, ZIPPER achieves the highest precision of 68.34% and an impressive recall of 98.14%. This represents a significant advancement over existing solutions, detecting nearly all vulnerabilities while maintaining high accuracy.

The performance gap is substantial when compared to baseline tools: RIPS, despite having the lowest false positives (125), only achieves 66.22% precision and 57.11% recall with 245 true positives. PHPJoern detects 285 true positives but generates an overwhelming 1,882 false positives, resulting in the lowest precision of 13.15%. TChecker shows limited effectiveness with only 71 true positives and 358 false negatives, leading to both low precision (32.13%) and recall (16.55%). While TChecker offers the fastest processing time at 11 minutes 26 seconds, ZIPPER's 30-minute processing time represents a reasonable trade-off given its superior detection capabilities.

Ground Truth Coverage and Unique Vulnerabilities. ZIPPER demonstrates its superiority not only in terms of precision and recall but also in the comprehensiveness of its vulnerabil-

ity detection. It successfully identified 421 true positives (TPs) with only 8 false negatives (FNs), significantly outperforming the three baseline tools. Specifically, ZIPPER covered *ALL* 299 vulnerabilities collectively detected by the three baseline tools—PHPJoern, RIPS, and TChecker. Among these, PHPJoern identified 285 vulnerabilities, RIPS detected 245, and TChecker identified only 71.

Additionally, ZIPPER identified 122 unique vulnerabilities that were not detected by any of the baseline tools, as shown in Figure 8. Among these, 16 were previously undiscovered vulnerabilities, meaning they were neither detectable by the baseline tools nor recorded in CVE database. *Notably, 3 of these 16 vulnerabilities have already been assigned CVE-IDs*.

FP Analysis for ZIPPER. The main reasons for false positives in ZIPPER include: ⓐ**Unrecognized Sanitization Operations (168)**: Programs use string operation functions like `substr`, `preg_replace`, etc., to implement custom sanitization. However, ZIPPER fails to identify these code snippets with sanitization semantics, leading to false positives. Identifying these code snippets is outside the scope of ZIPPER and baselines. ⓑ **Incorrect Call Edges (12)**: We infer the type of the receiver object based on its data flow characteristics, which led to 11 false positives in Joomla and 1 in WordPress. For example, in Joomla, multiple receiver objects with same data flow characteristics use the `set` and `get` methods. However, multiple classes, such as `JInput` and `Session`, contain both methods, which leads to false positives. ⓒ **Dead Code (11)**. Vulnerabilities in third-party libraries or test code, which are unreachable at runtime. ⓓ**HTTP Response Header Manipulation (4)**: Developers use `header()` to change the webpage's rendering mode, such as `header("Content-type: application/csv")` in PHPLiteAdmin for CSV rendering.

FN Analysis for ZIPPER. ZIPPER failed to identify a total of 8 vulnerabilities: 7 in Oscommerce2 and one in Joomla. In the case of Oscommerce2, ZIPPER was unable to detect CVE-2020-12058, which encompasses 7 distinct vulnerabilities sharing the same source but with 7 different sink points. We missed detecting these vulnerabilities because the sanitizer was incorrect, rather than missing sanitizer. Regarding Joomla, ZIPPER failed to detect CVE-2024-21726 due to incomplete code inclusion in the project. The relevant taint flow propagation code was located in a third-party library that fell outside the scope of the analysis.

FP and FN Analysis for PHPJoern & TChecker. The primary cause of false positives lies in the over-approximate taint analysis strategies employed by PHPJoern and TChecker. Specifically, ⓐ for arrays, both PHPJoern and TChecker consider the entire array tainted if any single element is tainted. ⓑ Regarding object handling, PHPJoern considers the entire object tainted if any of its properties are tainted, while TChecker implements type-sensitive taint propagation, resulting in fewer false positives compared to PHPJoern. However, false positives still occur. For example, when `$obj->m` is tainted and `$obj` is of type `MyClass`, TChecker will propagate the taint to all instances of `MyClass::m`, even if they are not directly affected.

Excluding false negatives in ZIPPER, the primary reasons for false negatives in PHPJoern and TChecker lie in call graph construction and taint propagation. ⓐ Call Graph Construction. Due to type inference failures, PHPJoern and TChecker missed method call edges, resulting in 102 and 75 false negatives, respectively. ⓑ Taint Propagation. PHPJoern and TChecker each missed 42 and 32 vulnerabilities, respectively, due to failures in alias or global variable tracking. Additionally, TChecker missed 251 vulnerabilities because of issues in its taint analysis implementation, such as its inability to track taint sources passed as function arguments.

FN Analysis for RIPS. RIPS does not track taint propagation for object properties, which helps avoid the over-approximation issues seen in PHPJoern and TChecker, contributing to its higher precision. However, this also acts as a double-edged sword: while it improves precision, it results in a higher rate of missed vulnerabilities due to its lack of object property tracking.

Time Overhead. As shown in Column 7 of Table 1, TChecker requires the least analysis time (11m26s), followed closely by RIPS (13m12s). Both tools achieve high efficiency due to their unsound and incomplete dynamic value inference, and results in incomplete call graphs and data dependency graphs, enabling faster but less thorough taint detection. PHPJoern, with its over-approximating taint propagation strategy (such as tainting one array element taints the entire array), requires significantly longer analysis time (144m27s). ZIPPER, employing on-demand static analysis techniques and more precise taint propagation (with array-index sensitivity and field sensitivity), achieves a balance between precision and performance. Despite generating more complete call graphs and data dependency graphs through comprehensive value-set analysis, ZIPPER's analysis time (30m45s) remains reasonable—about 1/5 of PHPJoern's time while being only 2.7 times slower than TChecker and 2.3 times slower than RIPS.

Summary to RQ1: ZIPPER outperforms all baseline tools with a highest precision of 68.34% and recall of 98.14%. It identifies 421 true positives and only 8 false negatives, detecting 122 unique vulnerabilities, including 16 previously undetected. Although slightly slower than TChecker and RIPS, ZIPPER offers a favorable trade-off between efficiency and accuracy, demonstrating its superiority.

4.3 RQ2: Ablation Studies

To understand the contributions of the techniques proposed for solving these summarized limitations, we performed ablation studies on each component. We implemented four variants of ZIPPER: ⓐ ZIPPER^{UTV}, which employs TChecker-style value-set analysis, following the call graph and def-use chains of given variables for evaluation. ⓑ ZIPPER^{NGDD}, which

Table 2: **RQ2** Comparison for Four ZIPPER Variants.

Variants	TP	FP	FN	Precision	Recall	Time
ZIPPER^{UTV}	321	245	108	56.71%	74.83%	24m20s
ZIPPER^{NGDD}	403	187	26	68.31%	93.94%	19m39s
ZIPPER^{NAA}	390	154	39	71.69%	90.91%	19m40s
ZIPPER^{NOD}	288	225	141	56.14%	67.13%	1270m46s
ZIPPER	421	195	8	68.34%	98.14%	30m45s

skips GDDG construction and disables handling of variable variables and the `extract()` function during LDDG construction. ⓒ ZIPPER^{NAA}, which disables alias analysis during taint analysis. ⓓ ZIPPER^{NOD}, which disables all on-demand strategies.

Contribution of Value Set Analysis. To evaluate the contribution of VSA, we implemented the variant ZIPPER^{UTV}. Benefiting from our precise semantic modeling of PHP features and inference rules, ZIPPER detected 100 more vulnerabilities than ZIPPER^{UTV}, with an 11.63% improvement in precision, as shown in Table 2. Of these, 82 vulnerabilities were detected using our inference rules, with 75 inferred through receiver object clustering and 7 by collecting constraints to infer user inputs. The remaining 18 were identified through inter-procedural, PHP features well-handled VSA.

Contribution of Multi-Stage Taint Analysis. To evaluate the contribution of taint analysis, we implemented the variants ZIPPER^{NGDD} and ZIPPER^{NAA}. Table 2 shows that compared to ZIPPER, ZIPPER^{NGDD} missed 18 vulnerabilities, with 10 detected through tracking class static properties and 8 through tracking global variables. Although ZIPPER identified 31 more vulnerabilities through alias analysis, its precision dropped by 3.35%. This is due to alias analysis introducing more taint propagation paths, resulting in additional false positives, as discussed in section 4.2.

Contribution of On-Demand Strategy. The ZIPPER^{NOD} variant was used to investigate whether the on-demand strategy makes our analysis scalable. In our evaluation, a 3-hour time limit was set for each project. Ultimately, ZIPPER^{NOD} timed out on five projects with over 35k LLoC, leading to faster degradation of TP than FP, and a resulting decline in precision. When constructing the global object/array data dependency graph, analyzing the point-to set for each access to object properties and array elements becomes computationally expensive, especially when there are numerous access points in the program.

Summary to RQ2: The evaluation demonstrates the effectiveness of each component of ZIPPER. The precise value set and taint analysis improved precision and contributed to the detection of 122 additional vulnerabilities. Additionally, the on-demand strategy enhanced the scalability of our analysis.

4.4 RQ3: Real-world Vulnerabilities

To demonstrate the practical applicability of ZIPPER, we applied it to real-world popular projects. The projects were chosen based on two specific criteria: ⓐ GitHub PHP open-source projects with more than 100 stars; ⓑ WordPress plugins with over 5,000 active installations. Based on these criteria, we randomly selected 80 GitHub open-source projects and 20 WordPress plugins, totaling 100 projects (including well-known projects such as WordPress and phpMyAdmin). As shown in Table 3, ZIPPER identified and reported 11 vulnerabilities and assigned 6 CVE-IDs after two man-months of effort—primarily spent on manual validation, including distinguishing new vulnerabilities from known issues and setting up project-specific environments and PoCs.

False Positive Analysis. ZIPPER produced 195 reports across 100 projects, among which 118 were identified as true positives, resulting in a precision of 60.51%. The 77 false positives align with the causes discussed in Section 4.2: 72 stem from ZIPPER's inability to recognize code snippets with sanitization or validation semantics, and 5 are due to dead code, where the vulnerable modules are never executed at runtime.

Case Study ❶ Onethink is an content management framework built on ThinkPHP. When inferring the type of the receiver object for the method call on line 9 in Figure 9, ZIPPER identifies other method calls with `D('Model')` as the receiver object (line 5). By applying heuristic rules, ZIPPER discovers that the `ModelModel` class contains both `generate` and `getTables` methods, inferring their type as `ModelModel`. This allows ZIPPER to correctly construct the call edges and detect the SQL injection vulnerability on line 20. In contrast, RIPS, which uses name-based matching, finds three methods named `generate` in the project, leading to an imprecise call graph that could result in false positives. Meanwhile, PHPJoern and TChecker fail to infer the types, missing the call edges and consequently missing the vulnerability.

Case Study ❷ Multi-Step-Form is a WordPress plugin for building forms. Figure 10 presents a simplified code snippet for better clarity. `from_aa` receives a taint source, extracts the `title` element to instantiate the `Step_Part` class, and stores it in the `$this->parts` array. Then, through a for loop, it retrieves each `$part` from the `parts` array and calls its `render_title` method, accessing the `title` property (alias of line 5) and triggering the XSS vulnerability. ZIPPER detects this vulnerability by accurately inferring the type of `$part` from dynamic array using PHP features well-handled value set analysis, and tracks taint propagation through precise alias analysis. PHPJoern and TChecker fail to detect this vulnerability as they miss the method call edge due to failed type inference. Even if TChecker is able to connect the call edge, its object-insensitive taint analysis results in false positives at other echo points of the `title` property.

Responsible Disclosure. Considering the potential risks of the vulnerabilities identified in our research, we responsibly

Table 3: **RQ3** Zero-Day Vulnerabilities Detected by ZIPPER. SQLi/XSS/Cmdi/A.I. stands for SQL injection/Cross-Site Scripting/Command injection/Active installations respectively.

No.	Application	Stars/A.I.	Vuln. Type	Status
#1			SQLi	Fixed
#2	PicUploader	1.2k	SQLi	Fixed
#3			SQLi	Fixed
#4	onethink	381	SQLi	Confirmed & CVE-Assigned
#5			SQLi	Fixed & CVE-Assigned
#6			XSS	Fixed & CVE-Assigned
#7	Cacti	1.7k	XSS	Fixed & CVE-Assigned
#8			Cmdi	Fixed
#9			Cmdi	Fixed & CVE-Assigned
#10	ProfileGrid	7k	XSS	Fixed & CVE-Assigned
#11	Multi-Step-Form	10k	XSS	Fixed

disclosed the findings to the project maintainers. Specifically, we reported these vulnerabilities to the relevant developers via email or designated channels and provided recommended strategies for fixing them. We also refrained from making our findings public for at least 90 days after disclosure. We collaborated with the maintainers of the respective projects and MITRE to responsibly disclose these vulnerabilities, resulting in the assignment of 6 CVE IDs to date.

Summary to RQ3: ZIPPER effectively detects vulnerabilities in real-world projects. Its precise value set analysis and taint analysis enable it to identify more vulnerabilities.

5 Discussion

Dynamic Features in Other Programming Languages While our work primarily focuses on PHP's dynamic features, it is important to note that similar dynamic characteristics exist in other dynamic programming languages such as Python and JavaScript. For instance, JavaScript supports dynamic method invocation through runtime-determined function names. Consider the following example:

```
const str = "hello";
const methodName = "toUpperCase";
str[methodName]();  // Outputs: "HELLO"
```

This code demonstrates dynamic method resolution where the `toUpperCase` method is invoked through a variable containing the method name, rather than through direct invocation. Such dynamic features are not unique to PHP but are common across various dynamic programming languages.

***Limitation of* ZIPPER** ⓐ **Selective Implementation of PHP Dynamic Syntaxes** Given the engineering effort required to support PHP's dynamic features, ZIPPER prioritizes those used in over 75% of PHP projects [7], as well as the top four features listed in Table 5. ZIPPER currently supports a subset of dynamic syntax, including dynamic array, file inclusion, dynamic functions, and variable variables. In contrast, features such as `__get`/`__set` magic methods, references, closures, and `match` expression are not supported due to relatively low frequency. Section 4.2 confirms that these features introduce no FNs/FPs under the known ground truth in our dataset, which supports our prioritization. We acknowledge that such issues may exist in the wild and are not captured due to dataset limitations. ⓑ **Imprecise Inference of Dynamic Values** Although VSA infers dynamic values through static analysis, it struggles with complex string operations outside the framework. Its heuristic rules may also not be applicable to all scenarios, leading to false positives or false negatives. Moreover, VSA adopts a branch-insensitive approach by merging states at control-flow joins for scalability, but this approximation may lead to imprecise call graph and alias analysis. ⓒ **Limitations in Web Framework Functions Modeling** ZIPPER currently focuses on modeling PHP built-in functions to cover core language features, but does not yet explicitly support framework-specific APIs, which may result in false positives or false negatives in projects that rely on such framework APIs. ⓓ **Lack of Support for Second-order Vulnerabilities** Detecting second-order vulnerabilities requires tracking taint propagation through databases, which involves modeling schemas, analyzing SQL queries, and identifying taint write/read points. However, the wide variety of database access methods, such as native SQL and ORM libraries, makes this task complex; therefore, support for second-order vulnerabilities is deferred to future work. ⓔ **Constraints of Analysis Scope** ZIPPER is unable to identify code snippets with sanitization semantics. Additionally, it does not analyze third-party code and currently supports only three types of taint vulnerabilities, which limits its detection capabilities. ⓕ **Semantic Gaps Between Code and Symbolic Representation** Static analysis tools inherently use program abstractions. ZIPPER employs semantic abstraction and over-approximation for feasibility, which may introduce a semantic gap with the original code (e.g., currently missing support of closure) and lead to false positives or false negatives.

***Potential Improvements of* ZIPPER** ⓐ **More Language Support** While adapting to more programming languages requires specific handling of their dynamic features, many of our core methodologies—including the value-set algorithm, on-demand alias analysis, and taint analysis—can be effectively transferred to applications in other languages. ⓑ**More PHP Syntaxes**. The syntaxes not yet supported by ZIPPER can be addressed by extending the symbolic expressions and analysis domains in VSA. For example, closures can be naturally supported by incorporating functions into the analysis domain. ⓒ**More Vulnerability Types** The tool's detection capabilities can be enhanced in two ways. First, by configuring more types of sources and sinks, we can detect a broader range of taint-style vulnerabilities. Second, other logical vulnerabilities, such as permission check vulnerabilities, can be transformed into taint-style vulnerabilities for detection, leveraging existing research approaches. Third, by applying ex-

isting database/SQL statement analysis techniques [45], ZIPPER can be extended to identify second-order vulnerabilities. ⓓ**More Automated Source Identification** Currently, source points vary across different applications and manual specification is time-consuming. Given that large language models (LLMs) have been trained on extensive codebases and possess broad knowledge of various programming patterns, they could potentially be leveraged for automated source identification. ⓔ**Modeling Web Framework Functions** These APIs from framework can be supported similarly by enhancing VSA with the semantics of transformation functions and resolving callbacks through VSA. ⓕ **Trade-off Between False Positives and Negatives** ZIPPER currently lacks configurable parameters for tuning the false positive/negative trade-off. Its analysis precision relies on core techniques like context-sensitive VSA and alias analysis rather than adjustable settings. We aim to introduce such options in future work.

6 Related Work

In this section, we review prior work in PHP application vulnerability detection, including static and dynamic approaches, and value set analysis techniques for dynamic value inference. Our review emphasizes challenges in handling PHP's dynamic features.

6.1 PHP Application Vulnerability Detection

Static Approaches. Static methods can be categorized into classical taint analysis [16, 18, 23–26, 30, 51], recurring vulnerability detection [44, 52], machine learning-based code auditing [31, 39, 40]. WebSSARI [23] uses a lattice-based type system to detect taint vulnerabilities in PHP but is intra-procedural and lacks support for dynamic features. Xie [51] models program behavior with function summaries, but it's context-insensitive and doesn't handle built-in functions. RIPS [16, 18] extends Xie's approach to model dynamic features and built-in functions, though it's limited to intra-procedural analysis. Pixy [24–26] improves taint analysis with constant propagation and alias analysis but doesn't support many PHP features like object properties. TChecker [30] provides inter-procedural, context-sensitive taint analysis and supports taint propagation for objects and arrays. However, its value inference is partial inter-procedural, and its alias analysis is object-insensitive. These methods employ intra-procedural or context-insensitive analysis with regex matching for dynamic objects, leading to false positives/negatives. ZIPPER combines context-sensitive VSA with two inference rules to effectively narrow type candidates. Moreover, traditional taint analysis builds full def-use chains with coarse handling of object properties and array indices, incurring high overhead and reduced precision. ZIPPER instead constructs a sparse GDDG, performing local dependency and alias analysis only as needed, improving both scalability and accuracy.

Dynamic Approaches. The main dynamic methods for detecting vulnerabilities in PHP applications is fuzz testing, which can be classified into black-box, gray-box, and white-box based on available execution information. Black-box fuzzing [19, 20] detects vulnerabilities from application responses, while building a navigation graph, identifying data dependencies and collecting input constraints from frontend. Gray-box fuzzing [21, 47] instruments web apps to gather detailed state for deeper analysis. White-box fuzzing [55] uses static analysis to identify potential vulnerabilities and provides feedback to the fuzzing.

6.2 Value Set Analysis

Value-set analysis is a foundational technique in static analysis used to construct data dependencies and analyzing advanced language features like reflection. [13] introduced a VSA method for binary programs, using abstract domains to approximate integer values at program points, mainly for security analysis. [32,56] infer value sets in Java through inter-procedural backward slicing and forward simulation to detect information leaks. [33,50,51] approximate string outputs with context-free grammars. However, these methods mainly focus on string type and do not account for PHP's dynamic features. To address this, ZIPPER (1) extends VSA to object and array domains using demand-driven analysis for scalability; (2) incorporates inference rules for external inputs and complex string operations to effectively infer string values and object types. To our knowledge, VSA has not been applied to other dynamic languages such as Python or JavaScript.

7 Conclusion

In this paper, we presents ZIPPER, a novel static analysis framework that addresses two key challenges: accurately inferring PHP's dynamic values and efficiently analyzing large-scale applications. ZIPPER achieves this through a context-sensitive, flow-sensitive value-set algorithm and an efficient on-demand taint analysis. Evaluation on 429 known vulnerabilities demonstrates ZIPPER's effectiveness, achieving the highest precision of 68.34% and an impressive recall of 98.14%, while analysis of 100 popular PHP applications revealed 11 new vulnerabilities, leading to 6 CVE assignments.

Acknowledgment

We appreciate the valuable feedback from the anonymous reviewers and shepherd. This work is supported by National Key R&D Program of China under Grant #2022YFB3103900, Strategic Priority Research Program of the CAS under Grant #XDCO2030200 and Chinese National Natural Science Foundation (Grants #62032010, #62202462, #62302500, #62202452).

Ethics Considerations

This research aims to enhance web application security by detecting taint-style vulnerabilities in PHP applications. While acknowledging that security analysis tools could potentially be misused, we have taken several precautions to ensure responsible research conduct. These include following responsible disclosure practices for all discovered vulnerabilities, coordinating with affected system maintainers, and providing appropriate usage guidelines with our open-source release. We believe the benefits of improving web application security for millions of users significantly outweigh the potential risks, which we have carefully mitigated through these measures.

Open Science

In alignment with USENIX Security's open science policy, we have made our tool ZIPPER publicly available. The complete source code, documentation, evaluation dataset, and experimental setup are fully accessible to enable reproducibility of our research findings. All materials are archived and accessible via Zenodo at doi:10.5281/zenodo.15582110. We believe this open approach not only validates our research claims but also facilitates future research in PHP application security analysis.

References

[1] Joomla! content management system. https://www.joomla.org.

[2] Owasp command injection. https://owasp.org/www-community/attacks/Command_Injection.

[3] Owasp cross site scripting (xss). https://owasp.org/www-community/attacks/xss/.

[4] Owasp sql injection. https://owasp.org/www-community/attacks/SQL_Injection.

[5] Website of zipper. https://sites.google.com/view/zipper-home/.

[6] Wordpress: Blog tool, publishing platform, and cms. https://wordpress.org/.

[7] Feras Al Kassar, Giulia Clerici, Luca Compagna, Davide Balzarotti, and Fabian Yamaguchi. Testability tarpits: the impact of code patterns on the security testing of web applications. In *NDSS*, 2022.

[8] Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 641–652, 2016.

[9] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. {NAVEX}: Precise and scalable exploit generation for dynamic web applications. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 377–392, 2018.

[10] Steven Arzt. Sustainable solving: Reducing the memory footprint of ifds-based data flow analyses using intelligent garbage collection. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1098–1110. IEEE, 2021.

[11] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269, June 2014.

[12] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE european symposium on security and privacy (EuroS&P)*, pages 334–349. IEEE, 2017.

[13] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *International conference on compiler construction*, pages 5–23. Springer, 2004.

[14] Eric Bodden. Heros: Ifds/ide solver for soot and other frameworks. https://github.com/soot-oss/heros.

[15] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 480–491, 2007.

[16] Johannes Dahse and Thorsten Holz. Simulation of built-in php features for precise static code analysis. In *NDSS*, volume 14, pages 23–26, 2014.

[17] Johannes Dahse and Thorsten Holz. Static detection of {Second-Order} vulnerabilities in web applications. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 989–1003, 2014.

[18] Johannes Dahse, Nikolai Krein, and Thorsten Holz. Code reuse attacks in php: Automated pop chain generation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 42–53, 2014.

[19] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. Black widow: Blackbox data-driven web scanning. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1125–1142. IEEE, 2021.

[20] Benjamin Eriksson, Amanda Stjerna, Riccardo De Masellis, Philipp Rüemmer, and Andrei Sabelfeld. Black ostrich: Web application scanning with string solvers. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 549–563, 2023.

[21] Emre Güler, Sergej Schumilo, Moritz Schloegel, Nils Bars, Philipp Görz, Xinyi Xu, Cemal Kaygusuz, and Thorsten Holz. Atropos: Effective fuzzing of web applications for server-side vulnerabilities. In *USENIX Security Symposium*, 2024.

[22] Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. Performance-boosting sparsification of the ifds algorithm with applications to taint analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 267–279. IEEE, 2019.

[23] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52, 2004.

[24] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 6–pp. IEEE, 2006.

[25] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 27–36, 2006.

[26] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security*, 18(5):861–907, 2010.

[27] Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 3–16, New York, NY, USA, 2011. Association for Computing Machinery.

[28] Haofeng Li, Haining Meng, Hengjie Zheng, Liqing Cao, Jie Lu, Lian Li, and Lin Gao. Scaling up the ifds algorithm with efficient disk-assisted computing. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 236–247. IEEE, 2021.

[29] Haofeng Li, Chenghang Shi, Jie Lu, Lian Li, and Jingling Xue. Boosting the performance of alias-aware ifds analysis with cfl-based environment transformers. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024.

[30] Changhua Luo, Penghui Li, and Wei Meng. Tchecker: Precise static inter-procedural analysis for detecting taint-style vulnerabilities in php applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2175–2188, 2022.

[31] Heloise Maurel, Santiago Vidal, and Tamara Rezk. Statically identifying xss using deep learning. *Science of Computer Programming*, 219:102810, 2022.

[32] Marc Miltenberger and Steven Arzt. Precisely extracting complex variable values from android apps. *ACM Transactions on Software Engineering and Methodology*, 2024.

[33] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th international conference on World Wide Web*, pages 432–441, 2005.

[34] Anders Møller and Michael I Schwartzbach. Static program analysis. *Notes. Feb*, 2012.

[35] Eric Olsson, Benjamin Eriksson, Adam Doupé, and Andrei Sabelfeld. {Spider-Scents}: Grey-box database-aware web scanning for stored {XSS}. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6741–6758, 2024.

[36] PHP-Parser. A php parser written in php. https://github.com/nikic/PHP-Parser.

[37] Ganesan Ramalingam. On sparse evaluation representations. *Theoretical Computer Science*, 277(1-2):119–147, 2002.

[38] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.

[39] Lwin Khin Shar, Lionel C Briand, and Hee Beng Kuan Tan. Web application vulnerability prediction using hybrid program analysis and machine learning. *IEEE Transactions on dependable and secure computing*, 12(6):688–707, 2014.

[40] Lwin Khin Shar and Hee Beng Kuan Tan. Predicting sql injection and cross site scripting vulnerabilities through mining input sanitization patterns. *Information and Software Technology*, 55(10):1767–1780, 2013.

[41] Chenghang Shi, Haofeng Li, Jie Lu, and Lian Li. Better not together: Staged solving for context-free language reachability. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 1112–1123, New York, NY, USA, 2024. Association for Computing Machinery.

[42] Chenghang Shi, Haofeng Li, Yulei Sui, Jie Lu, Lian Li, and Jingling Xue. Two birds with one stone: Multi-derivation for fast context-free language reachability analysis. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 624–636. IEEE, 2023.

[43] Chenghang Shi, Haofeng Li, Yulei Sui, Jie Lu, Lian Li, and Jingling Xue. Pearl: A multi-derivation approach to efficient cfl-reachability solving. *IEEE Transactions on Software Engineering*, pages 1–19, 2024.

[44] Youkun Shi, Yuan Zhang, Tianhao Bai, Lei Zhang, Xin Tan, and Min Yang. Recurscan: Detecting recurring vulnerabilities in php web applications. In *Proceedings of the ACM on Web Conference 2024*, pages 1746–1755, 2024.

[45] He Su, Feng Li, Lili Xu, Wenbo Hu, Yujie Sun, Qing Sun, Huina Chao, and Wei Huo. Splendor: Static detection of stored xss in modern web applications. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1043–1054, 2023.

[46] Yulei Sui, Ding Ye, and Jingling Xue. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 254–264, 2012.

[47] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities. In *2023 IEEE symposium on security and privacy (SP)*, pages 2658–2675. IEEE, 2023.

[48] W3Techs. Usage statistics of php for websites, 2024. https://w3techs.com/technologies/details/pl-php.

[49] Enze Wang, Jianjun Chen, Wei Xie, Chuhan Wang, Yifei Gao, Zhenhua Wang, Haixin Duan, Yang Liu, and Baosheng Wang. Where urls become weapons: Automated discovery of ssrf vulnerabilities in web applications. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 216–216. IEEE Computer Society, 2024.

[50] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–41, 2007.

[51] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th international conference on Software engineering*, pages 171–180, 2008.

[52] Seongil Wi, Sijae Woo, Joyce Jiyoung Whang, and Sooel Son. Hiddencpg: large-scale vulnerable clone detection using subgraph isomorphism of code property graphs. In *Proceedings of the ACM Web Conference 2022*, pages 755–766, 2022.

[53] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE symposium on security and privacy*, pages 590–604. IEEE, 2014.

[54] Fabian Yamaguchi, Niko Schmidt, and David Baker Effendi. Joern–the bug hunter's workbench, 2014. https://github.com/joernio/joern.

[55] Jiazhen Zhao, Yuliang Lu, Kailong Zhu, Zehan Chen, and Hui Huang. Cefuzz: An directed fuzzing framework for php rce vulnerability. *Electronics*, 11(5):758, 2022.

[56] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1296–1310. IEEE, 2019.

Appendix

A.1 Evaluation Benchmarks

To support our evaluation, we provide detailed statistics in Table 4 for the 15 PHP applications used as benchmarks in our study. These applications were selected based on their prior use in well-established research and cover a range of sizes and popularity levels.

Table 4: **Statistics of Evaluation Benchmarks.** Here, #LLoC means logical lines of codes for each project. #Stars refers to the number of stars a project has received on GitHub. #Vuln. refers to the number of vulnerabilities present in this project, as determined by the steps described in Appendix A.2. "NA" indicates that a project does not have a star count. The symbol ‡ denotes popular and well-known real-world applications with more than 1k stars.

Application	#LLoC	#Stars	#Vuln.	Source
Codiad (v2.8.4)‡	8,267	2.8k	34	[7,9,30]
Collabtive (v3.1)	171,968	215	2	[9,30]
cpg (v1.6.12)	64,324	68	2	[9,30]
Ecommerce-CodeIgniter-Bootstrap‡	61,078	1.3k	98	[30]
Joomla (v3.10.3)‡	273,754	4.8k	1	[7,9,18,30]
Mini-Inventory-and-Sales-Management	35,115	513	20	[7]
monstra (v3.0.4)	24,398	397	11	[30]
oscommerce2 (v2.3.4.1)	60,831	281	37	[9,16,30]
phpLiteAdmin (v1.9.8.2)‡	9,479	175	7	[30]
razor (v0.8.0)‡	91,531	1.1k	79	[7]
stock-management-system	42,002	181	10	[30]
webchess (v0.9)	3,924	NA	103	[8,9,30]
WeBid (v1.2.2)	34,614	115	14	[8,9,30]
WordPress (v5.4.8)‡	249,847	19.8k	1	[7,9,18,30]
zencart (v1.5.5)	110,163	382	10	[9,30]

A.2 Ground Truth Construction

We established the ground truth dataset through a systematic three-phase methodology to enable reliable precision/recall evaluation:

1. **Candidate Collection.** The candidate vulnerabilities originated from two complementary sources:
   - *Static Analysis Detection.* 2,478 potential taint vulnerabilities identified by ZIPPER and three baseline tools (TChecker, RIPS, and PHPJoern).
   - *CVE Database Mining.* Known taint vulnerabilities documented in the CVE database and related to the evaluated benchmarks. We employed specific keywords such as "XSS", "SQL injection", and "command injection", combined with project names from Table 4 (e.g., "XSS WordPress"), to search for relevant entries. Second-order vulnerabilities were excluded from our analysis, as they fall outside the detection scope of both ZIPPER and the baseline tools. For the remaining CVEs, we located their corresponding files or patches; those without available information were also filtered out. We then analyzed these files or patches to determine whether our target versions were affected by these vulnerabilities. Through this systematic process, we identified a total of 60 relevant CVEs, encompassing 413 individual vulnerabilities.

   After deduplication across the two sources—based on vulnerability type, file path, and the sink line number—we obtained 2,486 unique candidate vulnerabilities, including all 60 confirmed relevant CVEs.
2. **Manual Filtering.** After excluding the 413 CVE-validated vulnerabilities, we manually reviewed the remaining 2,073 static analysis reports to eliminate false positives. This process focused on two primary sources of inaccuracy:
   - *Over-approximation in complex data structures* — particularly imprecise taint propagation through arrays, objects, and nested containers.
   - *Properly sanitized input paths* — cases where user input was mitigated by built-in or application-specific sanitization routines, breaking the taint flow to sensitive sinks.

   This preliminary verification phase required approximately one man-month of expert effort. As a result of this manual assessment, 311 candidate vulnerabilities were retained for subsequent in-depth validation.
3. **Dynamic Validation.** For the remaining 311 candidate vulnerabilities, we created proof-of-concept (PoC) exploits for dynamic validation. In a controlled environment, these PoCs were used to simulate attacks and verify the exploitability of the vulnerabilities. If the PoC successfully triggered the vulnerability and caused the expected security event, the vulnerability was considered valid and added to the ground truth. This validation required two man-months of effort.

In total, we established a comprehensive ground truth dataset consisting of 429 vulnerabilities (413 from CVE database and 16 from static analysis tools). The distribution of these vulnerabilities across different applications is shown in the fifth column of Table 4.

A.3 UpSet Plot for ZIPPER and Beselines

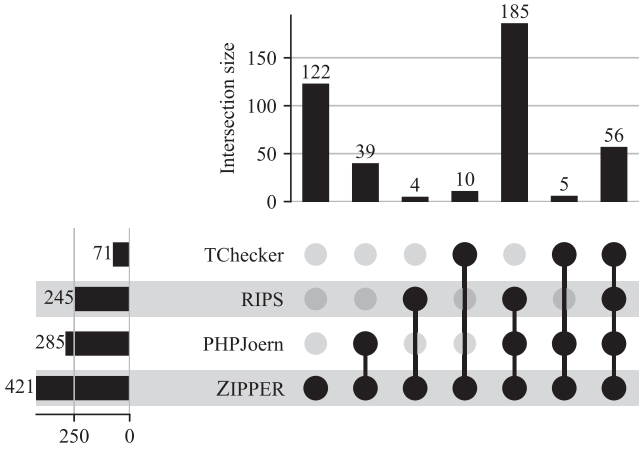

Figure 8: **RQ1** UpSet Plot for ZIPPER and Beselines. It illustrates the details of the intersections among the true positive sets found by ZIPPER and baselines.

A.4 Case Study for RQ3

```php
<?php
class ModelController extends AdminController {
  public function generate() {
    if (!IS_POST) {
      $tables = D('Model')->getTables();
      // ...
    } else {
      $table = I('post.table');
      $res = D('Model')->generate($table,
       ↪  I('post.name'), I('post.title'));
      // ...
    }
  }
}

class ModelModel extends Model {
  public function getTables() {
    // ...
  }
  public function generate($table,
   ↪  $name='',$title=''){
    $fields = M()->query('SHOW FULL COLUMNS FROM
     ↪  '.$table);
    // ...
  }
}
```

Figure 9: An SQLi vulnerability in Onethink

```php
<?php
class Step_Part {
  private $_title;
  public function __construct($title) {
    $this->_title = $title;
  }
  public function render_title() {
    echo $this->_title;
  }
  public static function from_aa($aa) {
    $title = isset($aa['title']) ? $aa['title'] :
     ↪  '';
    // ...
    return new Step_Part($title, ...);
  }
}
// ...
$this->parts[] =
 ↪  Step_Part::from_aa(taint_source());
// ...
for ($i = 0; $i < $cnt; $i++) {
  $part = $this->parts[$i];
  $part->render_title();
}
```

Figure 10: An XSS vulnerability in Multi-Step-Form

A.5 Usage Statistics and ZIPPER Support Status of PHP Syntaxes

Table 5: **Usage Statistics and ZIPPER Support Status of Partial PHP Syntaxes in 115 Projects Used for Evaluation.** "✓" indicates this feature is supported by ZIPPER, while "✗" indicates it is not. We annotate each PHP feature with its support status in ZIPPER. Features marked as unsupported are considered for future extensions.

No.	PHP Syntax	# of Projects	ZIPPER Support
1	Dynamic Array	112	✓
2	File Inclusion	112	✓
3	Dynamic Functions	89	✓
4	Variable Variables	74	✓
5	Reference	67	✗
6	Closure	38	✗
7	`__get`/`__set` Magic Methods	34	✗
8	`match` Expression	11	✗