PHPJoy: A Novel Extended Graph-based PHP Code Analysis Framework

Youkun Shi, Yuan Zhang, Tianhan Luo, Guangliang Yang, Shengke Ye, Chengyu Yang, Fengyu Liu, Xiapu Luo, Min Yang

***Abstract*—Nowadays, the PHP language is widely used in web development. Owing to PHP's inherent flexibility and dynamic language features (e.g., cross-module dependencies and runtime polymorphism), PHP applications are prone to various security vulnerabilities, such as XSS and SQL injection. As an effective PHP semantic understanding and security vetting technique, static program analysis has been widely applied. However, prior work faced difficulties in dealing with diverse and dynamic PHP features, which caused serious false negatives (e.g., call target missing).**

In this paper, we propose a novel extended graph-based program analysis approach, called `PHPJoy`, that can effectively and universally learn the semantic landscape of the target PHP program and conduct security validations. Specifically, `PHPJoy` first performs fine-grained program analysis (i.e., cross-module analysis and field-level analysis) for the purpose of learning the extended semantic graphs. Then, based on the graph-based semantic information, `PHPJoy` universally models various security issues by efficiently utilizing a new security-oriented graph query framework, which provides rich and easy-to-use graph query APIs and a high-performance cache-and-prefetch strategy.

We evaluate `PHPJoy` on 333 popular PHP programs. The results show that `PHPJoy` can effectively discover 269,901,982 semantic graph edges, improving by 23.76% when compared to the existing analysis tools. Our further analysis also shows that the runtime analysis overhead is reduced by 76.54%. Furthermore, `PHPJoy` successfully hunts 53 zero-day security vulnerabilities in the wild, which verifies the practicality of `PHPJoy`.

***Index Terms*—PHP Program Analysis, Web Security, Vulnerability Detection.**

I. Introduction

In recent years, PHP has become widely used in web development, particularly for server-side programming. According to a recent report [1], PHP powers 74.7% of websites worldwide, including numerous popular websites such as Facebook and WordPress. Given PHP's widespread adoption, its security has become a critical concern, as vulnerabilities could lead to severe security threats such as Cross-Site Scripting (XSS) [2], [3], SQL Injection (SQLi) [4], [5], and Remote Code Execution (RCE) [6], [7].

Recently, graph-based program analysis [8]–[15], particularly the Code Property Graph (CPG) which synthesizes multi-layer code semantics, has shown great promise. This approach was first adapted for PHP analysis in `PHPJoern` [16], which successfully demonstrated the potential of using CPG traversal to identify severe web vulnerabilities.

However, effectively applying this graph-based approach to modern PHP applications still faces significant challenges. On one hand, the intrinsic flexibility and extensive use of dynamic features in PHP—such as dynamic includes that create cross-module dependencies, complex class hierarchies, and fine-grained field-level array access—significantly impede the accuracy of static analyzers. Specifically, the challenge of resolving "include" statements with variable-based paths cascades into subsequent analyses, preventing the construction of a complete class hierarchy and ultimately leaving cross-module call targets unresolved. Similarly, tracking data flows through field-level array access (e.g., `$info[U_TOKEN]`) is challenging, as field-insensitive approaches may overlook the precise data dependency. As a result of these obstacles, critical edges in the constructed semantic graphs may be missed, inevitably causing false negatives during vulnerability detection.

On the other hand, current graph-based security methods also struggle with the efficiency of large-scale analysis. A significant factor contributing to this is the poor performance of graph traversal and queries used by existing tools, often resulting in unacceptable time overhead, and even timeouts [16]–[18]. This inefficiency stems from the fact that current graph query techniques are designed for whole-graph level operations, which are resource-consuming, whereas security-oriented analysis typically focuses on smaller graph segments. For example, taint analysis for detecting vulnerabilities usually focuses on analyzing security-relevant code blocks along the source-to-sink path. Spending excessive analysis time on other redundant and vulnerability-irrelevant code blocks on the path will hinder efficiency.

Motivated by those, we propose `PHPJoy`, a novel graph-based PHP code analysis framework designed for fine-grained and high-performance general program analysis and security validations. To address the above analysis limitations, we design an extensive semantic graph data structure, namely Extended Code Property Graph (E-CPG), with the combination of several helpful semantics, e.g., module dependency graph (MDG), class hierarchy graph (CHG), and field-level

This work was supported in part by the National Natural Science Foundation of China (U2436207, 62172105, 62202106), the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012), Hong Kong RGC Projects (PolyU15224121, PolyU15231223) and HKPolyU Project (No. ZGGG).(Corresponding authors: Yuan Zhang and Min Yang.)

Youkun Shi, Yuan Zhang, Tianhan Luo, Guangliang Yang, Shengke Ye, Chengyu Yang, Fengyu Liu and Min Yang are with the School of Computer Science and Technology, Fudan University, Shanghai, China (e-mail: ykshi21@m.fudan.edu.cn; yuanxzhang@fudan.edu.cn; thluo20@fudan.edu.cn; yanggl@fudan.edu.cn; skye21@m.fudan.edu.cn; cyyang24@m.fudan.edu.cn; fengyuliu23@m.fudan.edu.cn; m_yang@fudan.edu.cn).

Xiapu Luo is with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China (e-mail: csxluo@comp.polyu.edu.hk).

semantics. Specifically, given a PHP program, PHPJoy can improve the effectiveness and efficiency of program analysis tasks in two main ways. On one hand, PHPJoy learns multi-layer semantics to handle the dynamic features of PHP. For cross-module programming, PHPJoy generates a module dependency graph and a class hierarchy graph to comprehend semantics across different modules. Additionally, PHPJoy utilizes field-level program analysis and backward value analysis to compute variable dependencies, facilitating the generation of E-CPG through the synthesis of the various semantic graphs. By offering a more comprehensive code semantic graph, E-CPG enhances the effectiveness of graph-based analysis techniques in practice. On the other hand, we introduce a security-oriented graph query framework that incorporates a performance-enhancing algorithm based on a graph cache-and-prefetch strategy. This approach addresses the unacceptable inefficiencies of existing graph-based techniques when applied to large-scale program analysis tasks.

In our evaluation, we applied PHPJoy on 333 popular web applications collected from GitHub, each having over 500 stars. Regarding the effectiveness of code semantic graph construction, PHPJoy can successfully discover 269,901,982 semantic graph edges. Compared to the previous technique PHPJoern, PHPJoy identified 23.76% more graph edges, including 22.81% more abstract syntax tree nodes, 24.93% more control flow edges, 37.84% more program dependency edges, and 167.52% more function call edges. In terms of analysis efficiency, further analysis indicates that the runtime analysis overhead was reduced by 76.54%, demonstrating the superiority of our cache-and-prefetch strategy. Leveraging effective code semantic graph construction and our efficient security-oriented graph query framework, PHPJoy successfully hunts 53 zero-day security vulnerabilities in the wild, verifying its practicality. To date, 17 of these vulnerabilities have been patched.

In all, this paper makes the following major contributions:

- We present PHPJoy, a novel graph-based program analysis framework capable of performing both general and fine-grained analysis through the design of a new cache-and-prefetch-based graph query framework.
- We evaluate the effectiveness and efficiency of PHPJoy by applying it to a set of popular PHP programs. PHPJoy is capable of learning significantly more graph edges than existing work while considerably reducing runtime overhead.
- We conducted a vulnerability detection task on a large scale of PHP programs using PHPJoy, and the evaluation results demonstrated its practicality, as it successfully identified 53 zero-day security vulnerabilities in the wild.

Data Availability Statement: we have open-sourced the PHPJoy artifacts[1][2], which include its prototype, usage instructions, a tutorial-style demo case, and evaluation dataset.

[1]https://github.com/seclab-fudan/PHPJoy
[2]https://zenodo.org/records/17015233 (v3 on Zenodo)

II. Background and Motivation

A. CPG and CPG-based Analysis

In recent years, graph-based analysis has gradually become a mainstream approach in program analysis [8]–[15], [17]–[20]. Yamaguchi *et al.* [8] first proposed a Code Property Graph (CPG) by combining three semantic graphs, i.e., Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependency Graph (PDG). Since then, CPG has been recently applied to various tasks such as automated vulnerability detection [10], [21], code clone detection [14], [17], [22], and code authorship determination [23], [24]. Given the significant progress made by CPG, it has been extended to several fields. For example, Khodayari *et al.* [13] extended CPG for client-side JavaScript programs by incorporating event traces and page environment to detect CSRF vulnerabilities. Similarly, Li *et al.* [11], [12], [25] extended CPG for the NodeJS program, using complex class attribute dependency to detect prototype pollution vulnerabilities.

B. Extended CPG to PHP Programs

Recognizing the effectiveness of graph-based approaches, Backes *et al.* [16] recently adapted CPG and extended it with a semantic graph, specifically a Call Graph (CG), for PHP program analysis. Their design of PHPJoern successfully demonstrated the promising efficacy of leveraging CPG to identify XSS, SQL injection, and other security vulnerabilities in PHP projects. However, analyzing PHP remains challenging due to its intrinsic flexibility and dynamic nature, which can lead to issues such as missing edges in graphs and potential false negatives. In light of this, TChecker [26] highlighted the inaccuracies in call graph construction within this work and attempted to address the problem by handling PHP's object-oriented features. Despite these efforts, TChecker adopts PHPJoern's approach for constructing control flow graphs and data flow graphs, thereby inheriting PHPJoern's inaccuracies in these areas, leaving effective and efficient PHP program analysis as an unresolved challenge.

C. Motivating Example and Challenges

To further illustrate how the inherent flexibility and dynamic nature of the PHP language make program analysis particularly challenging, let's consider a real-world example.

Real-World Example: We use the code snippet shown in Figure 1 to illustrate the challenges in constructing complete semantic graphs. This example is based on a real-world SQL injection issue. In this case, there is a data flow from user input `$_POST["token"]` (line 6) to a database query `PDO->exec()` (line 24). Within the PHP file `index.php`, the user input `$_POST["token"]` is stored in an internal array `$info[U_TOKEN]` (line 6). This value is then retrieved and concatenated with other strings to construct an SQL query statement (line 10), which is subsequently passed as a parameter to `dbExecute()` in `application.php` (lines 28-31). Following this, the virtual function `exec()` (lines 23-25) is invoked, forwarding the parameter to the lightweight built-in interface `PDO::exec()` [27] for executing the SQL

```php
1  define("APP","application");
2  const U_TOKEN = "user_token";
3  const IS_DRAFT = "is_draft";
4  include dirname(__FILE__).DIRECTORY_SEPARATOR.APP.".php";
5  $info = array("detail"=>implode($_POST["details"]));
6  $info[U_TOKEN] = $_POST["token"];
7  $info[IS_DRAFT] = true;
8  $sid = intval($info["detail"]["sid"]);
9  $sql = "SELECT usesleft FROM {{t_}}$sid";
10 $sql .="WHERE t='".$info[U_TOKEN]. "'";
11 $result = dbExecute($sql);
```

index.php

```php
12 class BaseFramework{
13    public static DbConnection $app;
14    // init $app and self::app->pdo;
15    public static function init(){}
16 }
17 /**
18  * @property \PDO $pdo
19  */
20 class Connection{
21    protected $pdo;}
22 class DbConnection extends Connection{
23    public function exec($sql){
24       return $this->pdo->query($sql);}
25 }
```

class.php

```php
26 include "class.php";
27 use BaseFramework as Framework;
28 function dbExecute($sql){
29    $db = Framework::$app;
30    return $db->exec($sql);
31 }
```

application.php

Fig. 1: A Motivating Example of an SQL Injection Vulnerability

query statement. However, due to the lack of adequate security checks, this process can lead to an SQL injection vulnerability.

By applying existing tools [16], [26] to the real-world code snippet above, we identified several analysis challenges that lead to missing or redundant edges in the code property graph. These challenges are detailed below.

Challenge I: PHP programs frequently employ multi-module architectures. For example, in `application.php`, an inclusion statement (line 26) dynamically imports `class.php`, merging its classes and methods into the execution flow. This creates implicit dependencies: the variable `$db` (line 29) in `application.php` references the static variable `$app` (line 13) from `class.php`, which instantiates the `DbConnection` class (line 22). Consequently, the `exec()` call (line 30) ultimately executes code at line 24. While such cross-module interactions are ubiquitous in PHP ecosystems, existing analyses [16], [26] struggle to capture these relationships completely. For example, `PHPJoern` explicitly states that it does not analyze `include` or `require` statements [16]. This design choice treats a project as a collection of separate files rather than a cohesive application, inevitably resulting in the omission of many cross-file calls and data-flow edges. More recent tools like `TChecker` [26] attempt to work around this issue by handling inclusion statements as call sites and the corresponding included files as user-defined functions in the call graph. However, this model could not reflect PHP's scoping rules, where an included file's code executes in the current scope, unlike a function call which creates a new local scope. Furthermore, we observed that `TChecker` cannot correctly handle the file inclusion statement in `index.php` (line 4) because of its dynamic concatenation of multiple operations (e.g., magic constant and function calls). As a result, these limitations prevent existing approaches from accurately understanding the application's multi-module architecture, rendering them unable to detect the vulnerability.

Challenge II: PHP programs are often developed using flexible and dynamic PHP features, such as array and hashmap access. For instance, in `index.php`, the security-critical variable `$info[U_TOKEN]` stores a user-provided token, which is a common source for PHP program analysis. However, accurately modeling its data dependencies requires a field-sensitive analysis, a complexity that prior works often simplify. For example, a field-insensitive tool like `PHPJoern` simply tracks dependencies for the entire `$info` array rather than the specific tainted field [16]. This causes it to incorrectly construct a data-flow dependency between `$info[U_TOKEN]` (line 10) and the unrelated array assignment `$info[IS_DRAFT]` (line 7), while ignoring the crucial source at line 6. To mitigate such issues for its taint analysis, `TChecker` [26] attempts to model arrays with concrete key values. However, in this case, the array key `U_TOKEN` falls outside its scope, preventing it from establishing the correct data dependency. This imprecision in handling dynamic features by both approaches can cause analyzers to misidentify data flows, leading to false negatives that propagate through def-use chains.

Challenge III: PHP programs usually involve complex function call relationships. For example, when resolving the virtual call `exec()` in `application.php` (line 30), an analyzer must infer the type of the `$db` object. `PHPJoern` resolves such dynamic method calls in a coarse-grained manner by constructing call edges that rely solely on matching function names [16]. However, its approach fails to construct a call edge if multiple methods with the same name exist in the project (e.g., our evaluation dataset includes applications with an average of 2,810 function definitions existing the same name), as it cannot disambiguate the target. This leads to an incomplete call graph and potential false negatives. Given this, `TChecker` [26] attempts to improve call graph precision by introducing data-flow analysis to infer object types. Unfortunately, its analysis is fundamentally limited because it is built upon the Program Dependence Graph (PDG) generated by `PHPJoern`. This PDG is incomplete as `PHPJoern` fails to understand cross-module semantics, thereby hindering the construction of cross-module data dependencies, such as `$app` in `application.php` (line 29) and `class.php` (line 13).

Challenge IV: The complexity of PHP programs makes efficient analysis a challenging task. Our analysis reveals that previous graph-based approaches [16]–[19], [26], [28] tend to be inefficient. This core issue stems from two main factors. First, end-users are required to develop analysis agents from scratch for specific tasks. This involves establishing numerous basic models (e.g., for source variables, sink functions, or common sanitizers) and implementing analysis tasks through low-level graph queries, demanding significant effort. Second, existing PHP program analysis frameworks [16], [26] lack efficiency optimizations specifically for security-oriented anal-

ysis (e.g., taint analysis for detecting injection vulnerabilities). These frameworks typically target and traverse the entire graph for queries, whereas security-oriented analysis often focuses on smaller fragments. Consequently, a substantial amount of program analysis time is spent on repetitive, security-irrelevant code, leading to poor runtime performance and even analysis failures, such as timeouts.

III. System Design and Implementation

A. Our Ideas

In this paper, we present PHPJoy, a novel extended graph-based framework for analyzing PHP programs. Our approach effectively understands the multi-layered semantics of PHP code and constructs comprehensive semantic graphs, thereby facilitating thorough security validations. Compared to prior work, PHPJoy offers two primary advantages.

First, PHPJoy applies fine-grained program analysis against the inherent flexibility and complexity of modern PHP code. Specifically, PHPJoy extends CPG by synthesizing a *Module Dependency Graph* (MDG), which depicts the file inclusion relationships among PHP files, and a *Class Hierarchy Graph* (CHG), which captures class inheritance relationships among PHP classes. These graphs facilitate the understanding of cross-module semantics and resolve call targets and data flows that span different modules. Furthermore, PHPJoy utilizes field-level program analysis techniques that are context-, field-, and array-sensitive. Relying on these techniques, PHPJoy not only expands with two new semantic graphs (MDG and CHG) to facilitate program analysis for end-users, but also enhances the understanding and handling of several dynamic features prevalent in modern PHP code. This results in the construction of more complete semantic graphs from existing ones while minimizing the risk of missing edges.

Second, prior CPG-based static analysis frameworks [16], [26] often only convert programs into Code Property Graphs (CPGs) and import them into graph databases (e.g., Neo4j). These frameworks require end-users to develop low-level queries from scratch against a graph database to detect vulnerabilities. These conventional querying methods are often inefficient for common security tasks like taint analysis. Such analysis typically targets specific paths from sources to sinks, focusing on security-relevant code blocks along the path. However, during analysis, conventional queries struggle to avoid repeatedly analyzing redundant and security-irrelevant code blocks (e.g., the functional code blocks along the path) throughout their entire graph traversal, which leads to performance bottlenecks. To address this, we have designed a novel, efficient graph query framework. By introducing a set of easy-to-use query APIs and a high-performance prefetch-and-cache strategy, our framework significantly enhances efficiency.

B. PHPJoy Architecture

The workflow of PHPJoy is illustrated in Figure 2. PHPJoy is divided into two main components:

- *Stage I: Effective E-CPG constructor:* PHPJoy constructs a more comprehensive code representation for the target program by understanding cross-module semantics and leveraging field-level program analysis techniques.

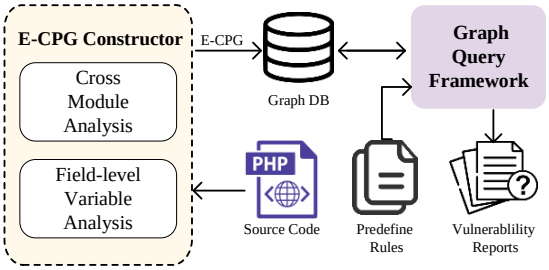


Fig. 2: The Architecture of PHPJoy.

- *Stage II: Efficient security-oriented framework:* PHPJoy provides a set of easy-to-use E-CPG query APIs and a built-in caching strategy to streamline the development of security analyzers and accelerate analysis tasks.

C. Stage I: Effective E-CPG Construction

1) E-CPG Definition: First, it is essential to define the Code Property Graph (CPG) that PHPJoy extends. As established in prior work like PHPJoern [16], a CPG for PHP programs is a data structure that merges multiple program representations into a single graph. Its primary components are as follows:

- *Abstract Syntax Tree (AST):* represents the syntactic structure of the code.
- *Control Flow Graph (CFG):* represents the possible order of statement execution.
- *Program Dependence Graph (PDG):* represents data and control dependencies between statements.
- *Call Graph (CG):* represents the calling relationships between functions to enable inter-procedural analysis.

Based on the CPG, PHPJoy further introduces the Extended Code Property Graph (E-CPG), improving two main aspects:

- *Synthesizing new graphs:* PHPJoy introduces two new graphs into the CPG (i.e., Module Dependency Graph (MDG) and Class Hierarchy Graph (CHG)). These graphs depict file inclusion and class inheritance relationships within complex PHP applications, thereby facilitating cross-module semantic analysis (in §III-C2).
- *Resolving dynamic features:* Based on the newly introduced graph, PHPJoy performs field-level program analysis to address the prevalent dynamic features in PHP applications, which further refines the completeness of the original graphs within the CPG (e.g., PDG and CG), leading to more precise subsequent analysis (in §III-C3).

2) Resolving Crossing-Module Semantics: Given the multi-module architectures of modern PHP applications, PHPJoy should understand the semantics across different modules. Therefore, we introduce the Module Dependency Graph (MDG) and the Class Hierarchy Graph (CHG) into CPG to aid in this process. More details are provided below.

Module Dependency Graph (MDG). We define the module dependency graph as $MDG = (N, E)$, where each node N represents a PHP module or file, and each edge E indicates a dependency between two distinct modules. However,

as demonstrated by the motivating example, analyzing the inclusion statement can be complicated by dynamic features, which require careful handling. Thus, we categorize module involvement into two main types: static and dynamic.

For static expressions, where the operand is often a constant string, we can directly parse the absolute path of the target file and establish an edge between the related modules. For instance, the inclusion statement on line 26 (Figure 1) uses a static string, allowing PHPJoy to locate the included file (i.e., class.php) and link it to the corresponding module.

In contrast, for dynamic cases, we focus on understanding the dynamic value of the file path expression, such as include dirname(__FILE__). DIR... on line 4 of Figure 1. Considering the complexity of dynamic expressions, we reconstruct the concrete paths of dynamically included files based on heuristic rule 1.

Heuristic Rule 1: We assume that dynamic paths are constructed via string concatenation involving variables (often through constant propagation), string manipulation operators (e.g., dot operator), specific magic constants (e.g., __FILE__), and a limited set of path-manipulation functions (e.g., dirname()). This heuristic is grounded in established PHP programming practices, as it is designed to cover the prevalent coding patterns used by developers.

Specifically, PHPJoy conducts data-flow value analysis to track variables within dynamic expressions. Since these expressions frequently involve constant propagation, PHPJoy incorporates two key mechanisms to enhance its effectiveness. First, PHPJoy maintains a map of constants to their resolved values, which facilitates value propagation. Second, it precisely models three types of operations commonly used to construct file paths: (1) built-in functions (e.g., getcwd(), dirname()), (2) magic constants (e.g. __FILE__, __DIR__), and (3) string manipulation operators (e.g., dot operator). This combined approach enables PHPJoy to effectively reconstruct dynamic inclusion paths, even when they appear ambiguous.

Taking the inclusion statement (line 4) from the index.php in Figure 1 as an example. The file path expression includes a dynamic string. By following the aforementioned design, PHPJoy can determine that dirname(__FILE__) yields the absolute path to the current directory. The specific value of the user-defined constant APP is application, and the built-in constant DIRECTORY_SEPARATOR is "\". By concatenating these strings, PHPJoy identifies that the included file in the statement is application.php, which allows it to establish the module dependency edge.

Class Hierarchy Graph (CHG). From PHP 5 onwards, developers can implement PHP programs using an object-oriented style [29]. Recently, with the advent of Composer [30], a widely used PHP package manager, developers can easily import third-party packages and their classes with various functionalities, mainly done via the USE statement. Given the popularity of class references, it is necessary to learn semantics to avoid edge missing in graph construction. For this purpose, we introduce the Class Hierarchy Graph (CHG), which aids in modeling class inheritance and reuse behaviors.

We define the class hierarchy graph as $CHG = (N, E)$, where N represents PHP classes, and E denotes class inheritance. Specifically, E can be categorized into three types according to PHP's class inheritance mechanisms: *EXTENDS*, *IMPLEMENTS*, and *TRAITS*. The first two types involve standard methods of class inheritance, while traits provide PHP's unique method of code reuse, representing the horizontal composition of PHP class behaviors [31].

To effectively construct the CHG, PHPJoy performs the following two steps. First, PHPJoy stores all class declarations, including their namespaces and corresponding file names, in a global hashmap. During this process, it handles class alias statements by converting aliased namespaces back to their original names. Additionally, PHPJoy uses the PHP built-in function get_declared_classes() to retrieve and store all PHP built-in classes. Second, PHPJoy identifies inherited classes in all class declaration expressions that contain class inheritance keywords [32]. Specifically, it searches for inherited classes or traits using the namespace declaration statements [33] within the current file. If the parent class cannot be found, PHPJoy expands the search by considering file inclusion relationships to explore a broader scope.

3) Conducting Field-Level Program Analysis: PHPJoy performs fine-grained field-level program analysis to handle several dynamic features, e.g., array access and virtual function invocations. As demonstrated in the motivating example (§II-C), context-, array-, and field-sensitive program analysis is essential for constructing semantic graphs and avoiding false negatives (i.e., graph edge missing). However, PHP's syntax for field and array access is highly flexible, accommodating several types of access operations like $a[0][1], $a[BAR_INDEX], $a[$bar], and $a->bar. These different formats are often semantically equivalent. Given that soundly resolving these dynamic features with static analysis is exceedingly difficult, PHPJoy employs two practical heuristic rules designed to balance effectiveness and performance. Our high-level idea is to conduct backward program analysis to infer variable dependency relationships, which aids in resolving field-level access. We provide more details on variable inference based on the category of access operations.

For array-style access operations, such as $a[0][1], $a[BAR_INDEX], and $a[$bar], PHPJoy employs a fine-grained structural approach to model the target array, tailored to the type of array indexing used. Initially, when PHPJoy encounters an array variable, it saves the variable and records its initialization position along with any explicitly initialized indices. If the index in an array-access operation is a constant value, PHPJoy can directly model the array using that index. For instance, as illustrated in Figure 1, PHPJoy notes that the array $info is initialized at line 5 and identifies the index as "detail". As the analysis progresses, PHPJoy continuously updates the array model. Specifically, if the index is a string or integer, PHPJoy creates or updates that index within the array model. For constant variables (e.g., BAR_INDEX), PHPJoy replaces the recorded symbol with its concrete value, if available, thereby mitigating dynamic array-indexing issues. When the index is a variable, PHPJoy attempts to compute its value, which is vital yet challenging for static analysis. To

mitigate this, PHPJoy applies a straightforward value analysis technique based on heuristic rule 2. Specifically, PHPJoy records statements that directly assign numeric or string values to variables and replaces these variables with constant values during later stages of analysis. For example, PHPJoy can correctly resolve the dependency for `$info[$key]` if `$key` was previously assigned a literal value (e.g., line 2 and line 6 in Figure 1). This heuristic covers many common programming patterns and significantly improves the completeness of the constructed graphs.

Heuristic Rule 2: We assume that the array-index variable is often assigned a simple string or numeric literal in the preceding code, rather than being the result of complex computations. Note that this rule aligns with common practices and serves as a practical workaround, as statically determining the precise value of a variable derived from complex computations is a challenging task, particularly in a highly dynamic language like PHP.

For regular field-style access (e.g., `$a->bar->func()`), PHPJoy employs a lightweight, heuristic-driven approach to determine the target instance's potential type. It then efficiently searches for relevant declarations within a limited scope. This process involves a backward data flow analysis to pinpoint where the target instance was initialized, which is crucial for type determination. Specifically, PHPJoy operates in a multi-step process. In the first step, PHPJoy begins by performing an intra-procedural backward data flow analysis on all variables associated with the dynamic call. This helps ascertain the potential type corresponding to the variable. When tracing back to a variable's initialization point, PHPJoy considers four possible sources. Note that inferring variable types in a weakly-typed PHP language is highly challenging. As a pragmatic trade-off, PHPJoy employs heuristic rule 3.

Heuristic Rule 3: We utilize annotations and type-hint information to infer variable types. For cases where this information is missing, we perform a backward analysis on the variable. However, for cases involving complex call chains, our approach makes a pragmatic design to ensure scalability on large applications: we trace only the immediate caller and analyze the type of the corresponding parameter at that call site.

- **Direct Assignment:** In these cases, PHPJoy records the type of the variable `$a` as class A (e.g., `$a = new A()`).
- **Function Pass-through:** When a variable like `$a` is derived from a function's input parameter. PHPJoy leverages annotations or type-hint information within the current function to handle it. If type hints are unavailable, PHPJoy examines the current function's call site and analyzes the type of the corresponding incoming parameter.
- **Function Return Value:** For scenarios like `$a = foo()`, PHPJoy initially inspects the return value information (i.e., annotations or type-hint) of the function `foo()`. For functions that lack this information, PHPJoy statically traces backward from its return statements to identify the types of the returned variables.
- **Class Field:** When dealing with class fields, such as `$this->bar`, PHPJoy first considers the current class. If the type remains undetermined, PHPJoy searches for direct assignment statements to `$this->bar` within the current class or its parent class constructor. It then traces these statements backward to ascertain the type of `$this->bar`.

In the second step, PHPJoy matches the method definition of the same name within the corresponding class and its parent classes, leveraging MDG and CHG information. This significantly optimizes method call chains. For example, consider an execution flow where `$db->exec` is called. PHPJoy traces backward along the current data flow to the function's initialization point (e.g., line 29 in `application.php`). PHPJoy then determines that the variable `Framework::$app` can be inferred as `DBConnection` based on the attribute declaration at line 13 in `class.php`.

In the final step, PHPJoy analyzes the field access information. It inspects the class definition to identify potential types and examines any explicit or implicit type hints. If the class attribute is not found within the defined class scope, PHPJoy looks up the parent class using the CHG. By understanding field-level semantics, PHPJoy can enhance the CPG by adding edges related to control flow and data flow.

4) Extending CPG: By leveraging the proposed techniques, i.e., module dependency, class hierarchy analysis, and field-level analysis, PHPJoy can construct E-CPG. These techniques are helpful for understanding multi-layer code semantics and improving the completeness of the constructed CPG.

First, by understanding cross-module semantics, PHPJoy can resolve method call targets that span different modules, enabling it to add more edges to the CPG and reduce false negatives. More specifically, PHPJoy resolves call targets based on their types: (1) For the `self` keyword, PHP uses the expression `self::foo()` to call a static method in the current class. PHPJoy connects the node of method call `self::foo()` with the method definition node of `foo` in the current class (2) For the `parent` keyword, PHP uses the expression `parent::foo()` to call a static method in the parent class. PHPJoy searches the CHG and connects the method call `parent::foo()` with the method definition node of `foo` in the parent class. (3) For constructor calls, we observed that if a class object is initialized without an overloaded constructor function, the default class constructor method is invoked. Therefore, when a constructor is absent in a given class, PHPJoy creates a default constructor function instead. (4) Given PHP's extensive library of built-in classes with rich functionality, PHPJoy opts to model several popular built-in classes to prevent false negatives.

Second, our proposed field-level analysis enhances CPG construction by improving both control and data flow perspectives. For array-style access, PHPJoy utilizes the modeling information of the target array to construct related data flows accurately. For instance, in Figure 1 of our motivating example, when constructing the data-dependent edge at line 10, the creation of the array index `"U_TOKEN"` at line 6 is already captured within the array modeling. This allows PHPJoy to accurately establish the data-dependent edge from

line 6 to line 10. Regarding dynamic function calls, which are prevalent in PHP, many functions are resolved dynamically, posing challenges for static analysis. This can lead to false negatives if call targets are missed. The field-level information provided by `PHPJoy` is crucial for improving the resolution of these call targets. By obtaining the class definition for a variable or examining a class field, `PHPJoy` can determine the possible types of the target function and efficiently search for potential declarations within a limited scope.

D. Stage II: Efficient Security-oriented Framework

Relying on the constructed E-CPGs, `PHPJoy` can further perform security analysis, primarily through E-CPG-based graph traversal and queries. However, when applied to real-world PHP programs, we found that existing graph query algorithms are difficult to use and often perform poorly. The core reasons are as follows: First, their design is based on incomplete program representations (e.g., CPGs), preventing them from leveraging the full advantages of E-CPGs. Second, their implementations are task-specific and tightly coupled, making secondary development challenging. Third, security-oriented analysis (e.g., taint analysis for detecting injection vulnerabilities) typically focuses on small code fragments within a program. In contrast, conventional approaches (e.g., regular cypher queries) target and query entire program representations. This leads to significant query and analysis time being spent on repetitive, security-irrelevant code handling. For example, identical functional code blocks on different source-to-sink paths do not impact security, but their repetitive analysis impedes efficiency.

Therefore, we aim to address the aforementioned limitations with a new security-oriented, E-CPG-based, and efficient query framework. This new framework not only leverages the full advantages of E-CPGs but is also general and easily extensible to various security analysis scenarios. Specifically, the framework includes graph query APIs and a cache-and-prefetch strategy. Further details are provided below.

1) Graph Query API: Our framework offers a set of efficient and user-friendly query APIs that facilitate the development of security analysis tasks. These APIs can be categorized into the following five types:

- **AST-Related APIs:** 17 APIs are designed to facilitate operations on AST nodes. For example, the `match_ast_nodes()` method is designed to locate nodes that meet specific conditions. This can be used to identify specified nodes, such as user-controllable variables, thereby defining the origin or endpoint for graph traversal.
- **Cross-Module APIs:** 9 APIs are designed to support cross-module and class inheritance relationship analysis. For instance, the `get_parent_class()` method helps locate the parent class for a given class definition node, aiding in class type-related analyses [34], [35].
- **Control-Flow APIs:** 5 APIs are designed to handle control flow operations. For instance, the `get_predecessors()` method identifies the control flow predecessors of a given node, while the `get_reaching_conditions()` method returns a set of reaching conditions for a given node within an intra-procedural scope.
- **Call Graph APIs:** 4 APIs are designed to provide inter-procedural analysis interfaces.
- **Data-Flow APIs:** 7 APIs are designed to provide the capability of def/use analysis and taint analysis.

Moreover, these APIs can be further divided into two levels: (1) Basic APIs for searching for nodes or their neighbors, and (2) advanced APIs for traversing the graph and fetching multi-layer relationships (e.g., related nodes and edges) based on input nodes. Note that an advanced API is often implemented by combining and using several basic APIs. For example, `get_predecessors()` is a basic API that retrieves the predecessor of a given node. In contrast, `get_reaching_conditions()` is an advanced API whose actual implementation is split into several calls of `get_predecessors()` to get the reaching conditions. This stems from a bottom-up design principle, which effectively avoids repetitive queries and better serves the subsequent cache-and-prefetch strategy (as detailed in §III-D2).

2) Cache-and-Prefetch Strategy: Regarding our cache-and-prefetch acceleration strategy, we observed that top-level queries are rarely identical, but they are typically composed of multiple basic API calls. Therefore, our cache-and-prefetch strategy also follows a bottom-up design principle. For example, in taint-style vulnerability detection, the same source might flow to different sinks, resulting in multiple flow paths (e.g., $Source$-A-B-C-$Sink_1$, $Source$-A-B-D-$Sink_2$, $Source$-A-D-E-$Sink_3$). Our bottom-up strategy caches each basic operation (e.g., A and B) instead of directly caching the entire path. Specifically, we developed a query optimization approach centered around this subgraph analysis target, which includes: (1) *Subgraph Caching*: our framework caches recently used nodes and edges for future use; and (2) *Query Prefetching*: our framework processes graph queries in advance independently, without going through the low-level graph database query and waiting for its results.

Subgraph Caching. For subgraph caching, a subgraph is stored in memory and consists of two components: (1) *Node Cache*, the queried graph nodes; and (2) *Edge Cache*, the queried edges between nodes, including abstract syntax tree edges, control flow edges, data flow edges, call edges, and module dependency edges. As the analysis progresses, the cached subgraph expands by recording the queried nodes and edges, eventually covering code representations pertinent to security analysis (e.g., source, sink, and their data flow). During runtime, when a graph query API (as defined in §III-D1) is invoked, the system checks the cached subgraph to verify whether the requested node is already included. If a cache hit occurs, the API operates directly on the cached subgraph to process the requested node. Otherwise, the framework queries the low-level graph database to retrieve the necessary data, which is then updated in the cache system.

Query Prefetching. However, the subgraph caching strategy still has performance limitations: every time a node is not found in the cache, the entire graph database must be accessed, resulting in additional costs. To mitigate this issue, we have designed a parallel graph query scheme utilizing two independent threads. One thread manages security analysis using a series of query APIs (§III-D1), while the other communicates

with the graph database to update caches and construct a complete subgraph. These threads operate in parallel, saving time, especially when numerous query requests are queued.

3) Vulnerability Detector: We model various security vulnerabilities by examining sensitive data flows that start with user input (i.e., source) and end at a potentially dangerous function (i.e., sink), lacking data flow protection (i.e., sanitization) throughout the process. The details are as follows.

Source. In PHP web applications, we focus on the following six types of super-global variables for user inputs:

1) `$_GET` variables are usually employed by developers to retrieve form data provided by user requests via the HTTP GET method, These variables encapsulate the parameters transmitted in the URL, formatted as key/value pairs. Considering that attackers can manipulate their values, we model this type of variable as a source in our analysis.
2) `$_POST` variables are similar to `$_GET` variables, but originate from user requests via the HTTP POST method. Given that attackers can manipulate these variables through the body of a POST request, our analysis also includes it.
3) `$_COOKIE` variables are utilized to access cookies sent by the user's browser. Cookies, small pieces of data stored on the user's device, are frequently used for session tracking and preserving user preferences. Due to their client-side location, they can be easily modified or tampered with by attackers. Therefore, we included them in our analysis.
4) `$_REQUEST` variables contains data from `$_GET`, `$_POST`, and `$_COOKIE`, and thus we included them.
5) `$_SERVER["HTTP_*"]` variables contain information about the headers of the HTTP request made to the server, such as `HTTP_USER_AGENT` for the user's browser and `HTTP_REFERER` for the referring URL. Since such headers can be modified by clients, these types of variables are also treated as sources in our analysis.
6) `$_FILES` variables are used to access file upload data from forms. It contains details about files uploaded via the HTTP POST method, such as the file's name, type, size, and a temporary storage path on the server. Given that they are as controllable as `$_POST`, these types of variables are also incorporated as sources in our analysis.

Sink. In the context of dangerous functions in PHP web applications, our analysis considers various common vulnerability types (as shown in Table I) and models corresponding PHP built-in sinks for them, drawing on insights from relevant works [16], [19]. When parameters of these functions are received from user input without any data flow protections, they can present significant security risks.

Sanitization. Based on notable existing works in PHP static analysis [6], [16], [19], [26], we have collected three types of PHP built-in functions that are commonly utilized by developers for data flow sanitization and vulnerability prevention: crypto/hash functions (e.g., `md5()`, `openssl_encrypt()`), numeric functions (e.g., `interval()`, `is_numberic()`), and vulnerability-specific sanitized functions (e.g., `mysqli_real_escape()` for SQLi).

TABLE I: Vulnerability Types and Sink Functions Mapping

Vulnerability Type	Potential Sink Functions
Cross-Site Scripting	echo, print, var_dump
SQL Injection	pg_query, pg_send_query, mysql_query, mysqli_query, mysqli_real_query
Local File Inclusion	include, include_once, require, require_once
Arbitrary File Read/Write	file_put_contents, fopen, fwrite, file, readfile, unlink, rmdir
Command Injection	exec, passthru, proc_open, system, shell_exec, popen, pcntl_exec
Executable File Upload	copy, fopen, move_uploaded_file, rename
Open Redirect	header
Directory Traversal	dir, dirname, opendir, scandir
PHP Object Injection	unserialize

E. Prototype Implementation

We have developed a prototype of `PHPJoy` based on the aforementioned design, integrating several practical features:
New Language Features. Each new PHP version introduces various features to the language, which are supported by the latest AST parser. Research indicates that using outdated AST parsers can lead to file omissions, negatively impacting the performance of static analysis tools [36], [37]. However, the AST parser used by `PHPJoern` supports only PHP 7.0, resulting in missed files, especially when processing recent PHP applications. To address this, `PHPJoy` employs the latest AST parser [38], which supports 16 additional node types and various new features beyond those provided by `PHPJoern`.
Fake Built-in Nodes. We store empty declarations of built-in classes and functions in a specific file named `_predefined.php`, allowing users to determine if a given function or class is built-in. During the construction of the E-CPGs, `PHPJoy` creates an empty declaration node in this file whenever it detects a built-in class, method, or function. Existing tools typically do not construct function call edges for built-in class methods or functions. Consequently, `PHPJoy` can establish these edges, a capability that `PHPJoern` lacks.
Data-flow and Control-Flow Enhancement. We have customized control flow and data flow adaptations for some new node types (e.g., match statement [39]). Additionally, we also address the ternary operator, which is a common syntax for primary conditional expressions in several programming languages. We enhance the internal control flow for the ternary operator to handle it similarly to an IF-ELSE statement. For instance, in the expression `a ? b : c`, we add a control flow from `a` to `b` and another from `a` to `c`.

IV. Evaluation

We conducted three experiments to evaluate the effectiveness and efficiency of `PHPJoy`. Our evaluation is organized by answering the following research questions:

- RQ1: Can `PHPJoy` effectively generate E-CPGs? Compared to CPGs, can E-CPGs improve graph-completeness?

- RQ2: Can PHPJoy and E-CPGs facilitate security analysis?
- RQ3: Can PHPJoy be applied in real-world vulnerability detection?

A. Experimental Setup

1) Experiments: In the following, we give an overview of the experiment that is designed to answer each research question. More experimental details will be given in the subsequent sections (see §IV-C, §IV-D and §IV-E).

Experiment-1: To answer RQ1, we employed both PHPJoern and PHPJoy to generate graphs across our dataset. More specifically, we compare the number of nodes and the edges that have been constructed in the graphs. Moreover, we break down the contribution of each design of PHPJoy to the E-CPG nodes and edges.

Experiment-2: To answer RQ2, we select the vulnerability dataset (as shown in §IV-D) and use PHPJoy and PHPJoern to detect them for comparison. Specifically, we evaluate the development difficulty and the efficiency in different settings.

Experiment-3: To answer RQ3, we select the PHP applications dataset and focus on the capability of PHPJoy in detecting zero-day vulnerabilities on real-world web applications.

2) Dataset: To answer the above questions, we constructed an experiment dataset by crawling popular PHP programs from GitHub. This effort resulted in a collection of 333 web applications, each with more than 500 stars. For each application, we downloaded its dependent packages from the third-party PHP code management platforms, e.g., composer [30]. We then installed these additional dependencies to ensure the integrity of the web applications. Finally, our dataset contains over 1.1 million PHP files, with over 141.2 million lines of code, with an average of 424,000 lines per application. Furthermore, to validate PHPJoy's security analysis capabilities, we collect 29 real-world CVEs (vulnerabilities) from three popular projects (Piwigo, MantisBT, and LimeSurvey). These vulnerabilities cover 13 different tasks, categorized by vulnerability type and application version for testing, demonstrating that our dataset is representative. The details of the real-world vulnerability dataset are shown in Table II.

3) Evaluation Settings: We evaluated the efficiency and capability of PHPJoern and PHPJoy in detecting security vulnerabilities. For PHPJoy, we configured three settings: (1) *no Cache&Prefetch* indicates that all acceleration strategies are disabled. (2) *only Cache* implies PHPJoy is used with a caching strategy. (3) *w/ Cache&Prefetch* signifies that both caching and prefetching strategies are enabled. We configured the number of supporting threads to be 8. All experiments were performed on a Linux machine equipped with two Intel Xeon E7-4820 processors and 378 GB of RAM.

B. Result Overview

We applied PHPJoy to 333 popular web applications collected from GitHub. On average, each application required 8.5 minutes for graph construction.

Graph Construction. First, PHPJoy effectively identified a total of 269,901,982 semantic graph edges. Compared to the existing tool PHPJoern, PHPJoy reveals 23.76% more graph edges, including 22.81% more abstract syntax tree nodes, 24.93% more control flow edges, 37.84% more program dependency edges, and 167.52% more function call edges.

Comparison. Then, we compare the efficiency and effectiveness of PHPJoern and PHPJoy in detecting security vulnerabilities, i.e., 29 CVEs. The results show that PHPJoy not only identifies more known vulnerabilities than PHPJoern but also, thanks to our novel cache-and-prefetch strategy, improves the efficiency of static analysis by 70.82% with minimal additional memory usage.

Vulnerability Detection. Finally, we conducted vulnerability detection on real-world popular PHP applications and successfully identified 53 zero-day security vulnerabilities in the wild, confirming the effectiveness of PHPJoy.

TABLE II: The Details about 29 CVEs for 13 Tasks

Application	Id	Version	Type	CVEs
MantisBT	1	1.2.6	XSS	CVE-2011-3578
	2	1.2.15	XSS	CVE-2014-8987, CVE-2014-9281, CVE-2014-9571, CVE-2016-5364, CVE-2016-6837, CVE-2014-9701, CVE-2017-12061
	3	1.2.15	SQLI	CVE-2014-1608, CVE-2014-9573, CVE-2014-9089
	4	1.2.15	POI	CVE-2014-9280
	5	1.2.15	OR	CVE-2014-6316
	6	1.2.15	CMDI	CVE-2019-15715
	7	2.1.0	XSS	CVE-2017-12062, CVE-2017-6797, CVE-2017-6799, CVE-2017-7897
	8	2.3.0	XSS	CVE-2017-7897
Piwigo	9	2.8.3	XSS	CVE-2016-10083
	10	2.8.3	LFI	CVE-2016-10084, CVE-2016-10085
	11	2.8.3	SQLI	CVE-2017-10682, CVE-2017-17822, CVE-2017-9463, CVE-2017-17823, CVE-2017-17824
	12	2.8.3	EFU	CVE-2017-5608
Limesurvey	13	2.06	SQLI	CVE-2015-4628, CVE-2015-5078

C. RQ1: Effectiveness of E-CPGs Construction

To answer RQ1, we used both PHPJoern and PHPJoy to generate graphs for applications in our dataset. Specifically, we compared the number of nodes and edges constructed in the graphs, and analyzed how each design aspect of PHPJoy contributed to the graph nodes and edges.

TABLE III: Basic Information of the Graphs Generated by PHPJoern and PHPJoy for 333 PHP Projects

Type	PHPJoern	PHPJoy	Improv.
# PHP files	1,017,452	1,108,678	8.97%
# AST nodes	533,389,125	655,068,624	22.81%
# AST edges	532,371,673	653,959,946	22.84%
# CFG edges	42,674,208	53,310,857	24.93%
# PDG edges	22,967,988	31,659,585	37.84%
# Call edges	4,361,277	11,667,241	167.52%

1) Overview: Table III provides basic information about the projects and the resulting code property graphs constructed by PHPJoern and PHPJoy. Overall, PHPJoy is capable of analyzing an additional 91,226 files compared to PHPJoern. Beyond file analysis, our enhancements have resulted in E-CPGs containing over 22.83% more AST nodes and edges, 24.93% more CFG edges, 37.84% more PDG edges, and 167.52% more call edges than those of PHPJoern.

TABLE IV: Results of MDG Constructed by PHPJoy

Type	File Inclusion Statements	MDG Edges
Static	20,423	20,423
Dynamic	144,971	96,468
All	165,394	117,251

2) MDG Construction: As detailed in Table IV, PHPJoy identified 165,394 file inclusion statements across the 333 projects. Among these, 144,971 (87.65%) involve dynamic file inclusions, where file paths are determined at runtime, while only 20,423 are static. This high proportion of dynamic cases underscores the critical importance of supporting dynamic features for modern PHP applications.

Resolving inclusion statements is fundamental for creating a complete view of an application, as it connects disparate source code files into a single, coherent program representation. In all, PHPJoy successfully constructed MDG edges for 117,251 (70.89%) of all inclusion statements, including 96,468 dynamic cases. By building the Module Dependency Graph, PHPJoy can effectively trace function calls and data flows across different files, which is a prerequisite for discovering complex vulnerabilities that span multiple modules.

TABLE V: Results of CHG Constructed by PHPJoy

Type	Statements	Constructed Edges	Ratio
Implement	193,820	163,920	84.57%
Extend	577,734	519,820	89.98%
Trait	61,984	58,912	95.04%
Total	833,538	742,652	89.10%

3) CHG Construction: The extensive use of object-oriented programming in modern PHP is evident in our dataset, where we identified a total of 833,538 statements related to class inheritance (extend), interface implementation (implement), and code reuse (trait). The sheer volume of these statements highlights that constructing a Class Hierarchy Graph (CHG) is not a niche problem but a fundamental requirement for analyzing PHP applications. A complete CHG allows for the precise resolution of inherited methods and properties, which is crucial for building an accurate call graph and tracing data flows through objects.

PHPJoy demonstrated high effectiveness by successfully resolving the parent class or interface for 742,652 of these statements, achieving a success rate of 89.10%. This high resolution rate significantly reduces the risk of missing edges in the call graph and data dependency graph, which is vital for analyzing vulnerabilities like PHP Object Injection (POI) that depend on complex class interactions. The remaining challenges stem from two main issues. First, some PHP projects conditionally define multiple classes to maintain compatibility across different configurations. Second, certain class definitions are generated dynamically, which hinders PHPJoy from identifying the parent class that inherits from them.

4) CG Construction: The details of the CG construction are presented in Table VI. Additionally, Table VII summarizes the results of call graph construction produced by PHPJoern and outlines the optimization steps implemented by PHPJoy. Overall, PHPJoy improved the success rate of call edge construction to 56.48%, compared to 26.01% achieved by PHPJoern. Note that the completeness of the call graph is fundamental to improving vulnerability detection.

Function Call Edges. PHPJoy constructs 80.95% more function call edges than PHPJoern. A primary reason for this improvement is the accurate identification of built-in functions. PHPJoy identified 5,115,462 calls to built-in functions, which constitute 86.93% of the newly constructed edges and PHPJoern could not manage.

Static Method Call Edges. PHPJoy resolves 15.36% more static method call edges compared to PHPJoern. Notably, the proper handling of inheritance-related keywords (e.g., `self::`, `parent::`) alone contributed to 26.5% of the static method call constructions, which enables more precise inter-procedural analysis in object-oriented code. However, a considerable number of static function call edges remain unresolved. Unlike strictly typed languages like C++, where method calls are strictly invoked using `object::method()`, PHP allows for more dynamic constructs, such as `$a->object::method()`, which require fine-grained pointer analysis or hybrid analysis for effective resolution. Consequently, PHPJoy currently struggles to address these types of issues.

Constructor Call Edges. PHPJoy creates 26.50% more constructor call edges than PHPJoern. This is achieved through two key enhancements. First, PHPJoy identifies 106,797 built-in classes, which account for 14.04% of the constructed constructor call edges. Second, PHPJoy creates default class constructors for 134,074 classes that lack an

TABLE VI: Results of CG Constructed by PHPJoy

Mode	Constructed Edges
PHPJoern	4,361,277
PHPJoern +AST	5,689,115
PHPJoern +AST+Built-in	10,911,374
PHPJoern +AST+Built-in+CHG&MDG	11,051,358
PHPJoy	11,667,241

TABLE VII: Comparison of Call Graph (CG) Construction for PHPJoern and PHPJoy across 333 Popular Projects.

Types	PHPJoern		PHPJoy	
	Statements	Edges	Statements	Edges
# of function call	5,005,857	731,940	6,164,224	5,891,421
# of static method call	1,554,697	359,029	1,930,707	751,877
# of constructor call	1,453,498	236,767	1,777,048	760,328
# of dynamic method call	8,752,976	3,033,541	10,785,257	4,263,615
# Total	16,767,028	4,361,277	20,657,236	11,667,241

explicit one, representing another 17.6% of constructed edges. Nevertheless, dynamic features for class initialization, such as `new $a->clazz()` or `new $a::clazz()`, remain outside PHPJoy's current capabilities.

Dynamic Method Call Edges. Dynamic method calls represent a significant challenge in PHP analysis due to their prevalence. PHPJoy identified 10,785,257 dynamic method call statements in our dataset, accounting for 52.21% of all function and method calls. To prevent a massively incomplete call graph, PHPJoy resolves 40.55% more dynamic method call edges than PHPJoern.

5) PDG Construction: A cornerstone of precise taint analysis is accurate data-flow tracking, which is frequently complicated by PHP's flexible field and array access. Compared to PHPJoern, PHPJoy constructs 37.84% more Program Dependency Graph (PDG) edges by employing a field- and array-sensitive analysis strategy. However, verifying these edges is challenging given the scale—both tools generate over 50 million total PDG edges. Therefore, to qualitatively assess the improvement, we randomly selected and manually analyzed 100 functions. In this sample, we found that PHPJoern failed to generate proper data dependency edges in functions with array-related issues (34 cases), object-related problems (13 cases), and other implementation issues (23 cases). By successfully constructing these missing edges, PHPJoy creates a more complete PDG, allowing the vulnerability detector to trace taint from source to sink more reliably and thereby reduce false negatives.

D. RQ2: Further Analysis on E-CPGs

To answer RQ2, we select the known vulnerability dataset and use PHPJoy and PHPJoern to detect them. We evaluated the efficiency of these two tools under different settings.

Development Difficulty Comparison. First, we used PHPJoern and PHPJoy to construct graph databases for the target applications and their different versions, as shown in Table II. Given that PHPJoern only releases its program representation framework without any vulnerability detection code, we opted to implement separate vulnerability detection modules for both PHPJoern and PHPJoy. To ensure a fair comparison, we adhered to the details in §III-D3, using identical modeling for sources, sinks, and sanitizers. We also followed traditional taint propagation analysis: starting from a sink, we performed a backward data-flow analysis to locate the source. Any path reachable without a sanitizer was then reported as a potential vulnerability.

However, because PHPJoern lacks any analysis APIs, we were limited to using conventional graph query statements provided by Neo4j's library. This involved performing low-level operations, such as traversing from an AST node to its corresponding statement node, querying for data dependency relationships, or identifying the function node containing a given AST node, and subsequently analyzing its call edges. Consequently, this detection module comprised 609 lines of code and required 13 manual hours to develop.

In contrast, PHPJoy provides various APIs to easily solve these problems, such as AST-related APIs (e.g. use `get_root_node()` to traverse from an AST node to its statement node) and Data-Flow APIs (e.g., use `find_def_nodes()` to traverse back along the data-flow from a given statement node). This significantly reduces development complexity, allowing end-users to focus on designing detection logic rather than expending effort on low-level operations. As a result, PHPJoy's security detector was built with only 128 lines of code and took just 4 hours to implement.

Efficiency Comparison. Table VIII presents a comparison of analysis efficiency between PHPJoern and PHPJoy across 13 scanning tasks. We observed that, without any acceleration strategy, PHPJoy takes 24.39% more time than PHPJoern. This is because PHPJoy employs a more detailed construction of edges, requiring approximately 26.98% more graph query operations on average. However, the cache strategy improves PHPJoy's efficiency by 27.02%, and the prefetch strategy further boosts efficiency by 66.63%.

To further analyze the impact of these acceleration strategies on efficiency, Table IX provides a detailed breakdown of the contributions from cache storage and prefetch strategies. With the cache strategy, PHPJoy achieves an average cache hit ratio of 39.06%, indicating that at least 39.06% of graph queries are repeated. The overhead from external memory usage is minimal, requiring only an additional 10.65 MB. To further improve graph traversal efficiency, we introduced two prefetch tasks to accelerate the traversal speed. The results demonstrate that the cache strategy alone enhances traversal efficiency by 27.02%. When employing both prefetch and cache strategies, PHPJoy achieves an average cache hit ratio of 64.14%, marking a 25.08% improvement in the cache hit ratio with an additional 0.79MB of external memory usage. This demonstrates the effectiveness of our prefetch strategy.

Effectiveness Comparison. Furthermore, we evaluated PHPJoy's ability to detect known security vulnerabilities using our dataset. Our findings reveal that PHPJoy identified 27 CVEs, whereas PHPJoern detected 25 CVEs. This suggests that PHPJoern overlooked two SQL injection vulnerabilities, specifically CVE-2014-9089 and CVE-2015-5078. A detailed analysis of the reported vulnerabilities showed that most of these CVEs were located in straightforward code logic. However, the two CVEs missed by PHPJoern specifically involved dynamic features, which are challenging for existing solutions to effectively address. The details are as follows.

The simplified code related to CVE-2014-9089 is presented in Listing 1. The user-input variable is assigned to `$f_sort` and then flows into `$t_setting_arr['sort']`. Existing tools struggle to accurately track this data flow due to the field-level access involved (i.e., array access of `$t_setting_arr`). However, with the support of E-CPG, PHPJoy successfully identified the data-flow path from line 1 to line 4, ultimately reaching the sink function at line 7.

```
1 $f_sort = source();
2 ...
3 $t_setting_arr['_query_id'] = '';
4 $t_setting_arr['sort']=  $f_sort;
5 $t_setting_arr['dir'] =  $f_dir;
6 ...
7 $tc_setting_arr = sink( $t_setting_arr );
```

TABLE VIII: The Comparison of Analysis Efficiency (Millisecond) between PHPJoern and PHPJoy on 13 Scanning Tasks. Note that the times in the table only include query execution and do not include graph generation or import.

Id	PHPJoern	PHPJoy (No Cache-Prefetch)	PHPJoy (Only Cache)	Improv.	PHPJoy (Cache-Prefetch)	Improv.
1	35,099	36,419	26,419	27.46%	9,571	73.72%
2	32,129	35,124	30,844	12.19%	11,212	68.08%
3	25,713	41,840	31,848	23.88%	4,694	88.78%
4	22,762	39,002	22,124	43.27%	4,543	88.35%
5	23,111	32,028	21,067	34.22%	7,980	75.08%
6	22,635	31,783	20,589	35.22%	7,366	76.82%
7	31,211	36,341	26,419	27.3%	12,312	66.12%
8	35,135	39,431	30,844	21.78%	11,221	71.54%
9	17,233	17,532	1,5529	11.42%	3,426	80.46%
10	16,733	19,779	1,4065	28.89%	3,544	82.08%
11	25,135	31,428	14,775	52.99%	4,069	87.05%
12	755	781	771	1.28%	300	61.59%
13	27,483	31,313	21,489	31.37%	11,931	61.9%
Avg.	24,241	30,215	21,291	29.54%	7,090	76.54%

TABLE IX: The Details of Cache and Prefetch Information about PHPJoy in 13 Scanning Tasks.

Id	PHPJoy (w/ Cache)		PHPJoy (w/ Cache & Prefetch)			
	Cache Hit	Cache Size (MB)	Cache Hit	Cache Size (MB)	Prefetch Ratio[1]	Prefetch Size (MB)
1	33.40%	12.75	55.02%	13.64	56.80%	1.44
2	34.25%	12.89	55.94%	13.20	57.28%	1.54
3	43.73%	8.83	82.27%	8.95	58.26%	1.54
4	44.28%	8.35	82.02%	8.73	58.51%	1.59
5	44.25%	8.44	81.62%	9.70	58.41%	1.16
6	44.38%	8.31	82.27%	9.46	58.69%	1.16
7	34.32%	13.43	57.22%	14.13	58.80%	1.44
8	33.43%	13.55	58.14%	14.32	57.28%	1.83
9	38.97%	10.56	53.34%	11.12	29.22%	0.90
10	45.23%	8.12	63.62%	9.03	29.99%	0.92
11	44.34%	8.82	61.11%	9.23	29.04%	0.89
12	28.63%	9.01	42.39%	9.81	26.57%	1.00
13	38.59%	9.32	58.83%	11.32	36.74%	2.12
Avg.	39.06%	10.18	64.13%	10.97	47.35%	1.34

[1] Prefetch Ratio = (When Cache hit, the content is written by supportive threads) / (Cache hit time)

Listing 1: CVE-2014-9089 code snippet in MantisBT-1.2.15

In the detection of CVE-2015-5078, we observed that certain module references were difficult to parse, which hindered the regular security analysis of PHPJoern. Specifically, the file dataentry.php (as shown in Listing 2) contains the dangerous function dbExecuteAssoc(), which leads to CVE-2015-5078. However, the module and its embedding modules are imported with dynamic expressions. By utilizing cross-module references, PHPJoy was able to successfully identify this sensitive module.

```
$oresult=dbExecuteAssoc($oquery) or throw new
    HttpException(500);
```

Listing 2: CVE-2015-5078 code snippet in LimeSurvey-2.06

E. RQ3: Zero-day Vulnerability Detection in the Wild

To answer RQ3, we evaluated the practicality of PHPJoy in detecting security vulnerabilities within real-world applications. We applied PHPJoy to a dataset comprising 333 popular PHP web applications, to identify six common types of vulnerabilities: Server-Side XSS, SQL Injection, Directory Traversal, Arbitrary Executable File Upload, Arbitrary File Inclusion,

and Arbitrary File Read/Write vulnerabilities. Considering the challenges of large-scale scanning, we imposed a time limit of two hours for each application.

Report Verification. For potential vulnerabilities reported by PHPJoy, we manually investigated the identified vulnerabilities to confirm their exploitability. Given that this process requires significant efforts, three authors have participated, each with a minimum of 3 years of expertise in web security. For each vulnerability report, the analyst will inspect and confirm its exploitability by writing a PoC.

Vulnerability Disclosure. Overall, PHPJoy successfully identified 53 critical security vulnerabilities, consisting of 40 server-side XSS, 7 arbitrary file read/write, 2 SQL injection, 2 arbitrary file inclusion, 1 directory traversal, and 1 arbitrary executable file upload vulnerability. To date, 17 of these vulnerabilities have been patched, as detailed in Table X. The remaining vulnerabilities are still being discussed or are awaiting developer responses. For ethical considerations, we are not disclosing their specific details in this paper.

Vulnerability Impact. Among all identified vulnerabilities, Cross-Site Scripting (XSS) was the most prevalent, accounting for 40 potential security issues. These XSS vulnerabilities can have serious security consequences. For instance, through an XSS vulnerability, a remote attacker can inject malicious JavaScript code into a web page. When a regular user accesses this compromised page, the malicious code might execute harmful actions, such as installing malware on the user's device, stealing private data (like cookies), or redirecting the browser to a fraudulent web page (i.e., phishing). Beyond XSS, we identified several other types of vulnerabilities. Among them are seven arbitrary file read/write vulnerabilities, which an attacker can exploit to access and manipulate specific files, such as site configuration files or database information files. We also discovered two SQL injection vulnerabilities. An attacker could exploit these to directly manipulate the target database, such as adding an admin account or deleting data. Additionally, there is a local file inclusion vulnerability where an attacker can inject a malicious path into include or require functions, using directory traversal sequences like `".."` to navigate directories. This allows the inclusion of malicious scripts that enable arbitrary code execution. We also identified a directory traversal vulnerability, allowing an attacker to exploit functions like `scandir()` to traverse sensitive directories. Lastly, an arbitrary file upload vulnerability was found, which permits an attacker to upload files to a specified directory. Once the attacker knows the file's path, they can execute it to further exploit the vulnerability and achieve arbitrary code execution.

TABLE X: Breakdown of Identified Vulnerabilities (Fixed).

Applications	Version	Stars	Vulns (Fixed)
Dolibarr	20.0.0	6,179	1
Organizr	1.9	5,479	3
GLPI	10.0.16	4,845	9
DzzOffice	2.3.0	3,948	1
Pi-hole Dashboard	5.21	2,083	1
Live helper chat	4.50v	2,002	1
phpMyFAQ	3.1.18	599	1
Total	-	-	17

Comparison on Identified Vulnerabilities. We also evaluated the capabilities of PHPJoern against vulnerabilities identified by PHPJoy. Conducting static analysis on large-scale datasets, such as 333 popular projects, presents several challenges: it demands substantial storage for the graph databases constructed by the baseline, requires extensive analysis time, and generates a massive number of potential reports that necessitate manual review. Therefore, as a trade-off, we opted to conduct our comparison focusing solely on the 53 vulnerabilities identified by PHPJoy. As a result, PHPJoern could only identify 37 of these vulnerabilities. The reasons for the undetected vulnerabilities are as follows:

- *Unsupported PHP Features (8 FNs):* Eight vulnerabilities were missed in four applications. This occurred because they leverage PHP features introduced in newer PHP versions (7.1+). As a result, PHPJoern does not support these features, which prevented it from constructing the necessary code graphs for analysis. According to a prior study [40], these unsupported features can be classified into four types: *nullable type*[3], *symmetric array destructuring*[4], *arrow functions*[5], and *constructor property promotion*[6].
- *Incomplete Call Graph (3 FNs):* Three vulnerability were not detected due to PHPJoern's incomplete call graph. As mentioned in §II-C, PHPJoern fails to build call edges for function calls with multiple identically named targets, leading to missed vulnerabilities.
- *Array-insensitive Analysis (4 FNs):* Four vulnerability, similar in cause to the previously mentioned CVE-2014-9089, was missed because PHPJoern is insensitive to array indices. This led to incorrectly constructed data-flow edges, preventing the data-flow backward analysis from correctly identifying the source, and thus missing the vulnerability.
- *Timeout (1 FN):* One vulnerability remained undetected due to a timeout. We configured PHPJoern with the same two-hour analysis time limit as PHPJoy. However, the Dolibarr application (over 1 million LoCs) contains over 10,000 potential sinks, causing PHPJoern to miss the vulnerable sink within the specified time. In contrast, PHPJoy successfully detected this vulnerability as its Cache-Prefetch design enabled it to analyze more sinks (as demonstrated in Table VIII, this design reduces analysis overhead by 76.54%). However, we acknowledge that PHPJoy was also unable to analyze all sinks within the two-hour limit, a necessary trade-off for conducting large-scale scans.

V. Discussion

Limitations. Statically analyzing the source code of a highly dynamic language like PHP is inherently challenging. In a

[3]Features found in file "src/AuthLDAP.php::get_entries_clean()" and "src/Features/TreeBrowse.php::getCategoryItemType()" of GLPI, caused 3 FNs.

[4]Features found in file "includes/html/pages/search/packages.inc.php" of LibreNMS and in file "php/import.php" of OpenFlights, caused 3 FNs.

[5]Features found in file "src/Reservation.php::getReservableItemtypes()" of GLPI, caused 1 FN.

[6]Features found in file "src/Akeneo/Tool/Component/FileStorage/File/FileStorer.php" of PIM, caused 1 FN.

practical and scalable manner, PHPJoy relies on three key heuristic rules (see §III-B).

These heuristics may face three potential sources of incompleteness. First, the heuristic for MDG construction may miss dynamic file paths determined by user input (e.g., `$_POST['path']`), though it successfully handles over 70% of common patterns. Second, the array access heuristic assumes indices are simple literals and can fail to resolve values from external inputs or complex functions (e.g., `$index = complex($_GET['id']);`). We posit this is a minority case, and this design proved sufficient to find vulnerabilities that other tools miss. Third, type inference heuristic struggles to infer variable types determined by distant callers, which is a pragmatic decision to ensure scalability and avoid timeouts in analyzing large-scale applications.

Regarding unsoundness, our conservative heuristics target common programming patterns, thereby minimizing this risk. The primary exception is the type inference heuristic, which is challenged by PHP's weak typing. False positives may arise from imprecise type hints (e.g., a super-type used for a sub-type variable) or when our backward analysis incorrectly infer a type that depends on complex internal logic.

General Applicability. To demonstrate PHPJoy's practicality and general applicability, we investigated the usage of the PHP features it supports within the broader PHP ecosystem. Specifically, we collected an additional 1,000 PHP projects on GitHub, each with over 100 stars. Given that building an E-CPG for each of these projects would be time and storage-intensive, we opted for a heuristic pattern-matching approach to conduct a rapid, text-level preview scan. The results show that only 14 out of these 1,000 applications did not utilize any of the features supported by PHPJoy. More precisely, among the additional 1,000 GitHub projects, we found that 725 applications utilized dynamic file inclusion statements, 944 applications employed dynamic method calls, 548 applications used class-related statements, and 937 applications made use of dynamic field-level array access. This highlights the widespread prevalence of the PHP features targeted by PHPJoy across the PHP ecosystem.

More Vulnerability Detection Scenarios. PHPJoy is equipped to handle a variety of scenarios in vulnerability detection tasks. For instance, PHPJoy effectively characterizes the class inheritance mechanism and optimizes the concatenation of class methods, which are crucial for PHP vulnerability analysis. Existing POI detection often relies on hybrid or static analysis [35]. The E-CPG provided by PHPJoy can be used in the static analysis portion of these methods, thereby aiding researchers in vulnerability detection. During our evaluation, we identified 36 projects that did not include any user-controllable input or sink functions as modeled by our approach. Upon manual review, we discovered that many of these applications rely heavily on specific frameworks, such as Laravel. To address vulnerability detection within specific frameworks, PHPSAFE [36] has conducted detailed modeling of WordPress and a thorough analysis of vulnerabilities in WordPress plugins. Thus, future researchers can also focus on modeling specific scenarios to more effectively target their vulnerability detection efforts.

Application Scenarios. We illustrate the application of PHPJoy for vulnerability detection in §IV-E. However, as a general graph-based PHP code analysis framework, PHPJoy can be utilized in a wider range of application scenarios. Several static analysis efforts have been based on PHPJoern, including NAVEX [7], SKYPORT [19], and AFV [20]. While these researchers have identified issues in PHPJoern and implemented some improvements, their solutions remain isolated and rudimentary. Each of these tools could serve as a downstream task for PHPJoy. We believe that PHPJoy will simplify the process for PHP security researchers to develop static analysis code, minimizing the occurrence of false positives or negatives due to inaccurate graph constructions. Moreover, since PHPJoy maintains a graph-based data structure, the extended code property graph it generates can be utilized in graph neural networks or sub-graph matching algorithms. This capability is particularly beneficial for automated vulnerability mining or code clone detection tasks leveraging deep learning technologies, such as HiddenCPG [17] and Recurscan [18].

VI. Related Work

PHP security has garnered significant attention, resulting in the development of various methods designed to detect security vulnerabilities in PHP applications. Early efforts in this area primarily focused on specific analysis techniques for particular vulnerability classes. For instance, Pixy [2], SQLCIV [41] and SAFEELI [42] employed data-flow analysis to identify SQLi vulnerabilities. Other research used static analysis to find execution-after-redirect vulnerabilities [43], second-order SQL injection [4], and PHP object injection vulnerabilities [34]. Despite their effectiveness, the merits of existing static techniques, which provide different layers of code semantics, were not well leveraged in the past.

Recently, graph-based program analysis approaches [8]–[15] filled the gap and made significant progress. Specifically, the novel data structure '*CPG*' (Code Property Graph) [8], [9] was recently proposed and designed to fully synthesize and unitize multi-layer code semantics, such as AST (abstract syntax tree), CFG (control flow graph), CG (call graph), and PDG (program dependency graph). Using CPG-based graph traversal algorithms, static techniques can universally model several types of security issues, contributing to the discovery of many severe security vulnerabilities [8], [9], [11] in C/C++ and other areas. With the effectiveness of graph-based techniques, Backes *et al.* [16] introduced CPG into PHP program analysis and developed the static analysis tool, named PHPJoern. However, CPG often fails to effectively address the broader challenges posed by PHP's flexible and dynamic features.

More recently, Jahanshahi et al. proposed Minimalist [44], a semi-automated tool for debloating PHP applications. Minimalist focuses on function-level debloating, a workflow that involves utilizing web server access logs to identify user-accessible entries and then traversing the call graph to remove unused functions. While Minimalist shares the motivation of statically analyzing file inclusions and class hierarchies to construct and refine a call graph, its core design, due to different objectives, may be counterproductive in other

security analysis scenarios. Specifically, Minimalist is practical for debloating, where conservatively keeping extra code is preferable to incorrectly removing required functionality. Therefore, for variables whose values cannot be determined through analysis (e.g., in path inclusions) or functions whose call targets cannot be accurately inferred, it adopts a fuzzy resolve strategy. This strategy involves constructing edges with all possible candidates. While this is a conservative approach in debloating tasks, it is, conversely, an aggressive strategy in vulnerability detection. This can lead to numerous infeasible program paths, consequently causing false positives and wasted analysis efficiency. Besides, as a tool designed for function-level debloating, it does not output other critical semantic graphs, such as Control Flow Graphs (CFGs) and Program Dependence Graphs (PDGs), which are essential for precise taint analysis.

In contrast, `PHPJoy` is designed as a fully-automated framework that utilizes more fine-grained techniques, including context-, field-, and array-sensitive analysis to resolve dynamic features while avoiding aggressive graph construction strategies. This allows `PHPJoy` to construct a more comprehensive and precise Extended Code Property Graph (E-CPG), not only by introducing the Module Dependency Graph (MDG) and Class Hierarchy Graph (CHG), but also by refining the completeness of the original Call Graph (CG) and Program Dependence Graph (PDG).

In conclusion, we believe that the proposed `PHPJoy` can bridge this gap and better facilitate future work on program analysis for PHP applications.

VII. Conclusion

In this paper, we propose a novel graph-based PHP program analysis framework, called `PHPJoy`. The proposed framework is capable of automatically understanding the code semantics of PHP programs and constructing comprehensive semantic graphs effectively. By integrating several efficient security-oriented graph query APIs and employing a high-performance cache-and-prefetch strategy, `PHPJoy` can significantly facilitate security analysis. In our evaluation of 333 popular web applications and 29 CVEs, our experiment results demonstrate that `PHPJoy` can significantly improve the semantic graph construction and reduce false negatives. Additionally, `PHPJoy` successfully found 53 vulnerable real-world cases.

References

[1] "Statistics of Programming Languages," 2025. [Online]. Available: https://w3techs.com/technologies/overview/programming_language

[2] E. K. Nenad Jovanovic, Christopher Kruegel, "Pixy: A Static Analysis Tool for Detecting Web Application vulnerabilities," in *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P)*, 2006.

[3] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, "Precise Client-Side Protection against DOM-based Cross-Site Scripting," in *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[4] T. H. Johannes Dahse, "Static Detection of Second-Order Vulnerabilities in Web Applications," in *Proceedings of the 23th USENIX Security Symposium (USENIX Security)*, 2014.

[5] K. Zhang, "A Machine Learning Based Approach to Identify SQL Injection Vulnerabilities," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[6] J. Dahse and T. Holz, "Simulation of Built-in PHP Features for Precise Static Code Analysis," in *Proceedings of the 21st ISOC Network and Distributed System Security Symposium (NDSS)*, 2014, pp. 23–26.

[7] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan, "NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018, pp. 377–392.

[8] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2014.

[9] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic Inference of Search Patterns for Taint-Style Vulnerabilities," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2015.

[10] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proceeding of the 33rd Conference on Neural Information Processing Systems (NIPS)*, 2019, pp. 321–332.

[11] S. Li, M. Kang, J. Hou, and Y. Cao, "Mining Node.js Vulnerabilities via Object Dependence Graph and Query," in *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*, 2022.

[12] ——, "Detecting node. js prototype pollution vulnerabilities via object lookup analysis," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[13] S. Khodayari and G. Pellegrino, "JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals," in *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, 2021.

[14] B. Bowman and H. Huang, "VGRAPH: A Robust Vulnerable Code Clone Detection System Using Code Property Triplets," in *Proceedings of the 32nd Annual Computer Security Applications Conference (ACSAC)*, 2020, pp. 53–69.

[15] S. Wang, X. Wang, K. Sun, S. Jajodia, H. Wang, and Q. Li, "Graphspd: Graph-based security patch detection with enriched code semantics," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[16] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and Flexible Discovery of PHP Application Vulnerabilities," in *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017, pp. 334–349.

[17] S. Wi, S. Woo, J. J. Whang, and S. Son, "HiddenCPG: Large-Scale Vulnerable Clone Detection Using Subgraph Isomorphism of Code Property Graphs," in *Proceedings of the 31st International Conference on World Wide Web (WWW)*, 2022, pp. 755–766.

[18] Y. Shi, Y. Zhang, T. Bai, L. Zhang, X. Tan, and M. Yang, "Recurscan: Detecting recurring vulnerabilities in php web applications," in *Proceedings of the ACM on Web Conference 2024*, 2024, pp. 1746–1755.

[19] Y. Shi, Y. Zhang, T. Luo, X. Mao, Y. Cao, Z. Wang, Y. Zhao, Z. Huang, and M. Yang, "Backporting Security Patches of Web Applications: A Prototype Design and Implementation on Injection Vulnerability Patches," in *Proceeding of the 31st USENIX Security Symposium (USENIX Security)*, 2022.

[20] S. Youkun, Z. Yuan, L. Tianhan, M. Xiangyu, and Y. Min, "Precise (Un)Affected Version Analysis for Web Vulnerabilities," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[21] Y. Wu, J. Lu, Y. Zhang, and S. Jin, "Vulnerability Detection in C/C++ Source Code With Graph Representation Learning," in *Proceedings of the 11th Annual Computing and Communication Workshop and Conference (CCWC)*, 2021, pp. 1519–1524.

[22] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection," in *Proceedings of the 25th ISOC Network and Distributed System Security Symposium (NDSS)*, 2018, pp. 18–22.

[23] B. Alsulami, E. Dauber, R. Harang, S. Mancoridis, and R. Greenstadt, "Source Code Authorship Attribution Using Long Short-Term Memory Based Networks," in *Proceedings of the 22rd European Symposium on Research in Computer Security (ESORICS)*, 2017, pp. 65–82.

[24] E. Dauber, A. Caliskan-Islam, R. Harang, and R. Greenstadt, "Git Blame Who ?: Stylistic Authorship Attribution of Small, Incomplete Source Code Fragments," in *Proceedings of the 17th Annual Privacy Enhancing Technologies Symposium (PETS)*, 2017.

[25] Z. Kang, S. Li, and Y. Cao, "Probe the Proto: Measuring Client-Side Prototype Pollution Vulnerabilities of One Million Real-world Websites," in *Proceedings of the 29th ISOC Network and Distributed System Security Symposium (NDSS)*, 2022.

[26] C. Luo, P. Li, and W. 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*, 2022, pp. 2175–2188.

[27] "PHP Data Objects," 2025. [Online]. Available: https://www.php.net/manual/en/book.pdo.php

[28] Y. Shi, Y. Zhang, T. Luo, X. Mao, and M. Yang, "Precise (un) affected version analysis for web vulnerabilities," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[29] "Php object-oriented programming design basics," 2024. [Online]. Available: https://www.zend.com/blog/object-oriented-programming-php

[30] "A Dependency Manager for PHP," https://getcomposer.org/, 2022.

[31] "Traits," https://www.php.net/manual/en/language.oop5.traits.php, 2022.

[32] "Object Interfaces," https://www.php.net/manual/en/language.oop5.interfaces.php, 2022.

[33] "Namespaces Overview," https://www.php.net/manual/en/language.namespaces.rationale.php, 2022.

[34] J. Dahse, N. Krein, and T. Holz, "Code Reuse Attacks in PHP: Automated POP Chain Generation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[35] S. Park, D. Kim, S. Jana, and S. Son, "FUGIO: Automatic Exploit Generation for PHP Object Injection Vulnerabilities," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[36] M. V. Paulo Nunes, José Fonseca, "PHPSAFE: A Security Analysis Tool for OOP Web Application Plugins," in *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2015.

[37] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing Web Application Code by Static Analysis and Runtime protection," in *Proceedings of the 13th International Conference on World Wide Web (WWW)*, 2004.

[38] "PHP AST," https://github.com/nikic/php-ast, 2022.

[39] "The Macth Expression," 2025. [Online]. Available: https://www.php.net/manual/en/control-structures.match.php

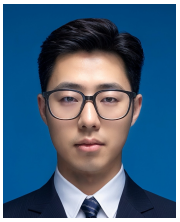
[40] L. Wang, Y. Zhang, X. Tan, S. Ye, and M. Yang, "New php language features make your static code analysis tools miss vulnerabilities," in *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2024, pp. 63–74.

[41] Z. S. Gary Wassermann, "Sound and Precise Analysis of Web Applications for Injection Vulnerabilities," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[42] X. Fu and K. Qian, "SAFELI: SQL Injection Scanner Using Symbolic Execution," in *Proceedings of the Workshop on Testing, Analysis, and Verification of Web Services and Applications (TAV-WEB)*, 2008.

[43] A. Doupé, B. Boe, C. Kruegel, and G. Vigna, "Fear the EAR: Discovering and Mitigating Execution after Redirect Vulnerabilities," in *Proceedings of the 18th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2011, pp. 251–262.

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



Youkun Shi received the Ph.D. degree in June 2024 from Fudan University, advised by Prof. Yuan Zhang and Prof. Min Yang. He is currently a postdoctoral fellow in the Department of Computing at The Hong Kong Polytechnic University, working under the supervision of Prof. Daniel Xiapu Luo. His research focuses on system security, especially web security.

Yuan Zhang received the BEng degree from Nanjing University, Nanjing, China, in 2009, and the PhD degree from Fudan University, Shanghai, China, in 2014, where he is currently a professor with the Software School. His research interests include vulnerability analysis, malware detection, and privacy preservation.

Tianhan Luo received the MEng degree from Fudan University, Shanghai, China, in 2023. His research interests include web application security and AI-driven software security.

Guangliang Yang received his PhD degree from Texas A&M University in 2019 and subsequently conducted postdoctoral research at Georgia Tech. He is currently an assistant professor in the School of Computer Science at Fudan University. His research is dedicated to constructing and advancing reliable and secure software by addressing foundational challenges in the design, implementation, and verification of evolving hardware-software coding.

Shengke Ye received the MEng degree from Fudan University, Shanghai, China, in 2024. His research interests are static analysis for software security.

Chengyu Yang received the BEng degree in Software Engineering from Dalian University of Technology, Dalian, China, in 2024. He is currently pursuing the MSc degree in Computer Science at Fudan University, Shanghai, China. His research interests include system security, program analysis, and large language model–driven software security.

Fengyu Liu is currently pursuing a Ph.D. degree at the School of Computer Science, Fudan University. His research focus is on Web Security and AI Security.

Xiapu Luo is currently a Professor with the Department of Computing, The Hong Kong Polytechnic University. His research focuses on blockchain and smart contracts security, mobile and IoT security, network security and privacy, and software engineering with papers published in top-tier security, software engineering, and networking conferences and journals.

Min Yang received the BSc and PhD degrees in computer science from Fudan University, Shanghai, China, in 2001 and 2006, respectively, where he is currently a professor with the School of Computer Science. His research interests include system security and AI security.