applied sciences

MDPI

Article

VulPathsFinder: A Static Method for Finding Vulnerable Paths in PHP Applications Based on CPG

Chunhui Zhao, Tengfei Tu *, Cheng Wang and Sujuan Qin

State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China; zhaochunhui@bupt.edu.cn (C.Z.); wang1998@bupt.edu.cn (C.W.); qsujuan@bupt.edu.cn (S.Q.)

* Correspondence: tutengfei.kevin@bupt.edu.cn

Abstract: Today, as PHP application technology is becoming increasingly mature, the functions of modern multi-layer web applications are becoming more and more complete, and the complexity is also gradually increasing. While providing developers with various business functions and interfaces, multi-tier Web applications also successfully cover the details of application development. This type of web application adopts a unified entrance, many object-oriented codes are used, and features such as encapsulation, inheritance, and polymorphism bring challenges to vulnerability mining from the perspective of static analysis. A large amount of object-oriented code makes it impossible for a simple function name-matching method to build a complete call graph (CG), resulting in the inability to perform a comprehensive interprocedural analysis. At the same time, the encapsulation feature of the class makes the data hidden in the object attribute, and the vulnerability path cannot be obtained through the general data-flow analysis. In response to the above issues, we propose a vulnerability detection method that supports vulnerability detection for multi-layer web applications based on MVC (Model-View-Control) architecture. First, we improve the construction of the call graph and Code Property Graph (CPG). Then, based on the enhanced Code Property Graph, we propose a technique to support vulnerability detection for multi-layer web applications. Based on this approach, we implemented a prototype of VulPathsFinder, a security analysis tool extended from the PHP security analyzer Joern-PHP. Then, we select ten MVC based and ten non-MVC-based applications to form a test dataset and validate the effectiveness of VulPathsFinder based on this dataset. Experimental results show that, compared with currently available tools, VulPathsFinder can handle framework applications more effectively, build a complete code property map, and detect vulnerabilities in framework applications that existing tools cannot detect.

Keywords: call graph; Code Property Graph; taint analysis; alias analysis; PHPtoTAC; object oriented; graph traversal

Citation: Zhao, C.; Tu, T.; Wang, C.; Qin, S. VulPathsFinder: A Static Method for Finding Vulnerable Paths in PHP Applications Based on CPG. *Appl. Sci.* **2023**, *13*, 9240. https://doi.org/10.3390/app13169240

Academic Editor: Vincent A. Cicirello

Received: 14 July 2023
Revised: 10 August 2023
Accepted: 12 August 2023
Published: 14 August 2023

1. Introduction

We frequently use static analysis to detect security vulnerabilities in programs, which has the advantage of analyzing code without executing programs. A series of security analysis tools, such as Pixy [1], RIPS [2], ooPixy [3], PHPSAFE [4], Weverca [5], and Joern-PHP [6], are commonly used to detect the loophole. However, with the increasing complexity of PHP applications, current tools and methods face additional challenges. Multi-layer web applications (such as MVC architecture) adopt a unified entry point and extensively apply object-oriented code, making it impossible for simple function name-matching methods to construct a complete call graph, resulting in the inability to perform comprehensive interprocedural analysis. At the same time, programming techniques such as encapsulation, routing, and global configuration are applied to hide vulnerabilities in complex data flows that cannot be accessed by universal data flow analysis [7].

Existing methods cannot effectively perform static analysis on modern multi-layer web applications, and the false positive rate is elevated [8,9]. Although some methods attempt to address this issue through dynamic analysis and automated vulnerability verification [7,10–14], these methods rely on static analysis. Existing static analysis techniques cannot effectively handle multi-layer web applications, leading to false positives and limiting dynamic analysis. To alleviate the above issues, we have incorporated the alias analysis procedure into the construction of the Code Property Graph. We use alias analysis techniques to infer object types, construct method call edges, and improve the call graph to perform complete context-sensitive inter-process data flow analysis. In addition, we propose a taint analysis approach to generate vulnerability data flow paths via variable propagation paths to improve the efficiency of graph traversal in constructing vulnerability paths.

Based on the proposed method, we have implemented a new static analysis tool, VulPathsFinder. Based on the Joern-PHP analyzer, it adds capabilities for alias analysis and the construction of variable access paths, enhancing the analysis capabilities of modern multi-layer PHP applications. We divide the workflow of VulPathsFinder into three phases. First, we convert the PHP source code into an intermediate representation, P-TAC [1], similar to the traditional three-address code [15]. Then, we construct a code attribute graph based on the abstract syntax tree and perform alias analysis [16] and context-sensitive data flow analysis [17] on this basis to construct a complete call graph and variable access paths. We mainly improve the original Code Property Graph in the following ways: (1) Collect variable alias information and track variable access paths; (2) Utilize alias information and type inference to improve the call graph; (3) Mark user input (sources) and security-sensitive calls (sinks) nodes based on the attack dictionary. Finally, we construct vulnerable data flow paths based on node labels and variable access paths.

The main contributions of this paper are summarized as follows:

- Design and Implementation of the VulPathsFinder tool: We developed a static analysis tool named VulPathsFinder, specifically designed to address the security analysis challenges in modern multi-layer PHP applications. Based on Joern-PHP, this tool extends the capabilities of alias and taint analysis to enhance its analytical power for complex applications.
- Improved Construction of Code Property Graphs: We redesigned the construction process of Code Property Graphs. We can generate a more accurate and comprehensive call graph by collecting variable alias information, tracking variable data propagation paths, and inferring type information for class instance variables.
- New Methods for Alias Analysis and Taint Analysis: We designed an alias analysis process based on comment parsing after constructing the control flow graph. We also marked user input and security-sensitive function nodes with taints. These improvements increased the accuracy of the tool's detection and reduced false positives.
- A Novel Method for Security Vulnerability Path Finding: We proposed a data flow analysis method based on graph traversal techniques that can effectively excavate application vulnerability paths. This method allows for an efficient backtracking analysis of security-sensitive functions in procedural and object-oriented programming.

Through experiments, we confirmed that these improvements significantly enhance the effectiveness of static analysis in complex applications. We believe that VulPathsFinder, along with the other contributions of this research, can provide new perspectives and methods for future security analysis of PHP applications.

The organization of this research is as follows. Section 2 discusses related work. Section 3 introduces the technical background, discusses the challenges that MVC architecture-based applications pose for security analysis using a running example, and briefly outlines the solutions implemented by VulPathsFinder to address these challenges. Section 4 describes the details of the improved CPG and vulnerability path construction algorithm. Section 5 discusses the experimental evaluation of VulPathsFinder. Finally, Section 6 includes the conclusion.

2. Related Works

Over the past decade, detecting security vulnerabilities in PHP code has been a focal point of research.

2.1. Static Analysis

In 2006, Xie and Aiken [18] addressed the issue of statically identifying SQL injection vulnerabilities in PHP applications. Meanwhile, Jovanovic et al. introduced a PHP static taint analysis tool, PIXY [1]. Their focus was cross-site scripting errors in PHP applications. They analyzed a total of six different open-source PHP projects. In these tests, they rediscovered 36 known vulnerabilities (with 27 false positives) and an additional 15 previously unknown defects (with 16 false positives). Wasserman and Su introduced two works focusing on statically finding SQL injections and cross-site scripting [19,20]. Jovanovic et al. [21] extended their method to SQL injection as a follow-up to the Pixy work. Although all these tools are pioneers in the field of automatic discovery of vulnerabilities in PHP applications, they only focus on very specific types of defects, namely, cross-site scripting and SQL injection.

Dahse and Holz [2] proposed RIPS. RIPS builds a control flow graph and then creates block and function summaries by simulating the data flow of each basic block, thereby performing precise taint analysis. In the process, the authors discovered previously unknown defects in osCommerce, HotCRP, and PHPBB2. Dahse and Holz continued their work by detecting second-order vulnerabilities (e.g., persistent cross-site scripting), identifying over 150 vulnerabilities in six different applications [22]. In 2015, Olivo et al. [23] discussed the static analysis of second-order denial of service vulnerabilities. They analyzed six applications, some of which overlapped with those analyzed in previous work, and found 37 vulnerabilities, accompanied by 18 false positives. David Hauzar et al. present WeVerca [5], a framework that allows one to define static analyses for PHP applications. It supports a type system, method calls, data structures, etc. The framework addresses the problem of implementation is either imprecise or overly complex by defining the end-user analysis independently of the value and heap analysis necessary to resolve these features. Zhao Jingling et al. [8] propose a framework for detecting vulnerabilities in PHP web applications. The framework combines static and dynamic analysis to improve the efficiency of detection. Paulo A.L.D. Nunes et al. [4] propose phpSAFE, a static analyzer that identifies vulnerabilities in PHP web applications developed using Object-Oriented Programming (OOP). The paper evaluates phpSAFE against two well-known tools and 35 widely used CMS plugins. The results show that phpSAFE clearly outperforms other tools. Michael Backes et al. [6] were the first to apply Code Property Graphs to mining vulnerabilities in PHP applications and implemented a method for discovering vulnerabilities in PHP applications based on Code Property Graphs. Our work is based on their implementation, and we further improved the call graph and data flow graph by collecting summary information of functions, methods, and instantiation statements to complete the call edges. At the same time, using summary information, attack dictionaries were combined to mark user input and security-sensitive operation nodes. Eventually, we constructed vulnerability paths using graph traversal techniques. Other work in this field concerns the correctness of sanitization programs [4,11,24].

2.2. Exploit Generation

Abeer Alhuzali et al. [25] present Chainsaw, a tool that improves the state-of-the-art in injection vulnerability identification and exploit generation for web applications. Chainsaw was used to analyze nine open-source applications and generated over 199 first- and second-order exploits combined, outperforming several related approaches. The next year, Alhuzali et al. presented NAVEX [7], a tool combining dynamic and static analysis techniques to identify vulnerabilities and build working automatic exploits in large multi-tier web applications. They all apply static symbolic execution to model the inter-state dependencies, which can produce seeds to guide the dynamic scanner to find vulnerabilities

hidden deeply in code. Lee et al. [10] propose a penetration testing tool, FUSE, designed to discover UFU and UEFU vulnerabilities in server-side PHP web applications. FUSE aims to generate upload requests; each request becomes an exploit payload that triggers a UFU or UEFU vulnerability. FUSE discovered 30 previously unreported UEFU vulnerabilities, including 15 CVEs from 33 real-world web applications, thereby demonstrating its efficacy in finding code execution bugs via file uploads. Park et al. [12] present FUGIO, the first AEG tool for POI vulnerabilities. The authors proposed a series of static analysis, dynamic analysis, and fuzzing techniques to compute POP chains and generate exploits. FUGIO reported 68 exploit objects from known POI vulnerabilities in 30 real PHP applications. FUGIO also reports two previously unknown POI vulnerabilities and generates exploitable objects, demonstrating the effectiveness of FUGIO in significantly alleviating the burden of laborious property-oriented programming.

In summary, static taint vulnerability detection techniques have become increasingly complete concerning data structures, from checking security-sensitive functions to control flow graphs and then to Code Property Graphs. At its core, it checks user inputs and traces data streams to trigger sensitive operations by detecting whether user inputs reach security-sensitive functions. However, due to the extensive use of object-oriented code, encapsulation, and dynamic features, modern multi-layer web applications make it difficult for these methods to accurately obtain the path of taint propagation or perform a complete inter-procedural analysis. While dynamic analysis or a hybrid approach of dynamic and static analysis can avoid the above issues, it cannot achieve fully automated detection due to the need for code-running environments and the lengthy time required for dynamic testing. We build on the Code Property Graph prototype Michael Backes et al. implemented and add method call edges to the call graph through alias analysis. We propose constructing the taint propagation path via the variable propagation path to generate the vulnerability path. On the one hand, we have constructed a complete call graph, allowing security analysts to conduct a more accurate analysis of the inter-process data flow based on this code attribute graph. On the other hand, we have addressed the problem of not accurately obtaining the path of taint propagation due to encapsulation.

3. Background and Motivation

In this section, we first introduce the basic knowledge of Code Property Graphs (CPGs). Subsequently, using several examples, we demonstrate the deficiencies in the prototype implementation of the existing CPG for vulnerability detection in modern PHP applications, inspiring us to redesign our solution.

3.1. Overview of Joern-PHP

Our research is based on the concept of a Code Property Graph, an integrated representation of the syntax, control flow, and data flow of a program. This concept was first introduced by Yamaguchi et al. and applied to detecting vulnerabilities in C code [26]. The key idea of this method is to incorporate traditional program representations into a Code Property Graph, thus enabling code pattern mining through graph traversal techniques. Abstract Syntax Trees (AST) clearly exhibit the nested structure of a program, while Control Flow Graphs (CFG) allow for the reasoning of interactions between statements, especially their execution order, and Program Dependence Graphs (PDG) reveal dependencies between statements and predicates. These dependencies enable static data flow analysis within a program, particularly the propagation of attacker-controllable data. Although the combination of AST, CFG, and PDG in a Code Property Graph provides a powerful structure for analyzing program control and data flows, CFG and PDG are limited to the function level. Incorporating call graphs into the Code Property Graph allows us to perform control and data flow analysis at the inter-procedural level.

In 2017, Michael et al. introduced this into the detection of vulnerabilities in PHP applications and used call graphs to enrich the Code Property Graph [6]. They developed Joern-PHP, a prototype tool for constructing Code Property Graphs for PHP applications,

and demonstrated how to use graph traversal techniques for inter-procedural vulnerability analysis. Although Michael et al. introduced the call graph into the prototype implementation of the Code Property Graph, the edges for method calls were incomplete. For dynamic method calls, an edge from the call node to the corresponding method declaration was only constructed when the method's name was unique in the project. Without type inference, the mapping relationship between dynamic method calls and class methods could not be obtained, rendering the tool unsuitable for vulnerability detection in modern applications that extensively use encapsulation and method calls. Therefore, we have further enriched this structure by using alias analysis, adding method call edges to the call graph, and tracking data propagation between variables or fields to facilitate inter-procedural analysis better.

3.2. Alias Analysis

As the foundation of static analysis, pointer analysis primarily examines all possible objects a pointer might refer to [27–29]. Object-oriented languages are mainly used to ascertain the potential objects a variable or field might point to. Pointer analysis plays a crucial role in the program analysis of object-oriented programming, with results commonly represented as points-to relations between pointers and memory locations or as sets of possible target objects for each pointer. It provides essential data flow information within the program. Recognized as one of the most fundamental static analysis techniques, research on pointer analysis has a history of over forty years, primarily focusing on languages like Java and C/C++ [30–32]. Pointer analysis is a static code analysis technique determining the objects a pointer or reference variable in a program might point to. This analysis is indispensable in many advanced program analysis tasks such as program slicing, data flow analysis, program understanding, etc.

Alias analysis [33] is an important branch of pointer analysis that focuses on determining two or more pointer variables that might refer to the same memory location during program execution, also known as aliases. This problem exists in many programming languages, especially those allowing explicit pointer operations like C and C++. The significance of alias analysis stems from its multiple application scenarios. For example, a compiler can use alias information to optimize a program, rearrange instructions more freely when no alias conflict is confirmed, and improve program execution efficiency. Furthermore, alias analysis is the foundation for many advanced program analyses and transformations, such as program slicing, data flow analysis, thread analysis, etc.

One of the challenges faced by alias analysis is that the problems it needs to solve are undecidable, meaning no algorithm can precisely determine all alias relationships in all circumstances. Therefore, existing alias analysis techniques often need to balance precision and efficiency. Common strategies [28,31] in practice include:

- Flow-sensitive analysis: This type of alias analysis considers the execution order of the program, providing more precise results but with higher computational complexity.
- Flow-insensitive analysis: Contrary to flow-sensitive analysis, this type of alias analysis disregards the execution order of the program, yielding less precise results but with lower computational complexity.
- Context-sensitive analysis: This type of alias analysis considers the context of function calls, capable of handling alias relationships between functions but with higher computational complexity.
- Context-insensitive analysis: In contrast to context-sensitive analysis, this type of alias analysis ignores the context of function calls, resulting in less precise outcomes but with lower computational complexity.

Despite the numerous challenges faced by alias analysis, it can still be a highly useful tool by selecting strategies suitable for specific application requirements.

3.3. Taint Analysis

Taint analysis [34] is a technique that tracks and analyzes the flow of tainted information within a program. In vulnerability analysis, taint analysis is used to mark data of interest (*sources*, e.g., untrusted user-supplied data) as tainted data. Vulnerabilities can be discovered by tracking the flow of tainted data and examining whether it influences critical program operations (*sinks*). This transforms the problem of identifying program vulnerabilities into determining whether tainted information reaches sink points where critical operations are performed.

Taint analysis typically involves the following components:

- Identification of sources where tainted information is generated and marking the tainted data.
- Tracing and analyzing the propagation of tainted information through specific rules.
- Detection at sink points whether critical operations are affected by tainted information.

Tainted information can propagate not only through data dependencies but also through control dependencies. We refer to information flow via data dependencies as explicit information flow and information flow via control dependencies as implicit information flow.

- Data flow-based taint analysis treats taint analysis as data flow analysis focused on tainted data without considering implicit information flow. It involves tracking tainted information according to taint propagation rules or marking variables along the path of contamination and checking whether tainted information affects sensitive operations.
- Dependency-based taint analysis considers implicit information flow. The analysis process checks whether sensitive operations at sink points depend on operations that receive tainted information at source points based on dependencies between statements or instructions in the program.

Static taint analysis systems [2,4,6,7,35] first parse the program code to obtain an intermediate representation. They perform auxiliary analyses such as control flow analysis to obtain control flow graphs, call graphs, and other necessary information. During the auxiliary analysis, the system can employ taint analysis rules to identify source points and sink points in the intermediate representation. Finally, the detection system utilizes static taint analysis based on the taint analysis rules to check for vulnerabilities of taint-related types in the program.

3.4. Running Example

For statements that create class instances using new ones, obtaining the class type of the instantiated variables is relatively easy. Nowadays, an increasing number of PHP applications are developed based on frameworks. These frameworks encapsulate common business functionalities using classes and extensively utilize dynamic arrays and reflection features. This makes it challenging to obtain type information through static analysis. Static analysis cannot retrieve the class methods corresponding to method calls, resulting in incomplete call graphs and difficulty in conducting interprocedural data flow analysis.

To explain the research questions and motivation, we extracted relevant code from two real-world applications and combined them into sample code snippets [36,37], as shown in Listings 1 and 2:

Listing 1. The first section of the simple code snippets from a MVC application.

```
    keywordskeywords
1  <?php
2    $DB = new DB();
3    class Push extends Controller{ // This is a Controller of the MVC-based
       Application
4        function index(){
5            $cf_name = $this->request.post('cf_name'); // Handle user input,
       and mark $cf_name as tainted source
6            $SQL = new SQL();
7            $_t1='( blog_name LIKE \'%' . $cf_name . '%\'';
8            $SQL.WHERE_and($_t1); // $SQL->where is tainted
9            $_t2=$SQL.get(); // $_t2 is tainted
10           $blogs_Results = new Results($_t2); //$Result->sql is tainted
11           $blogs_Results.run_query();
12       }
13   }
14   class Controller{
15       /**
16        * @var \think\Request
17        */
18       var $request;
19       function __construct($app = null){
20           $this->app = Container::get('app');
21           $this->request = $this->app['request'];
22       }
23   }
24   class Results{
25       function __construct($sql = null){
26           $this->sql = $sql;
27       }
28       function run_query(){
29           global $DB;
30           $this->rows = $DB.query($this->sql);
31       }
32   }
```

Listing 1 defines a controller class named Push, which extends the Controller class (lines 3–13). In the index() method of the Push class, the class property request is invoked with the post() method, and its return value is assigned to the variable *$cf_name*. The request property originates from the parent class Controller. Listing 2 shows that the post() method retrieves user input from the global variable *$_POST* array.

In Listing 1, the property *$this->request* in the Push class is initialized in its parent class Controller. In the Controller class, *$this->request* is initialized as an instance of *\think\Request* through dynamic array assignment in the constructor. We cannot obtain the class type of *$this->request* through static or alias analysis. However, we notice that we can infer from the annotation of the requested property (lines 15–17) that it is an instance of the *\think\Request* class. Apart from instance creation statements, standardized development practices provide us with additional static information. We can obtain type information for corresponding parameters or class fields through annotation parsing.

Listing 2. The second section of the simple code snippets from a MVC application.

```
    keywordskeywords
1  class Request{
2      function post($name = ''){
3          return $_POST[$name];
4      }
5  }
6  class DB{
7      var $dbhandle;
8      function __construct(){
9          $this->dbhandle = new mysqli();
10     }
11     function query($query_SQL){
```

```
12          $this->result = $this->dbhandle.query($query_SQL); // native sink
13          return $this->result;
14      }
15  }
16  class SQL{
17      var $where = '';
18      function get(){
19          return $this->where;
20      }
21      function WHERE_and($where_and){
22          $this->where .= '(' . $where_and . ')';
23      }
24  }
25  ?>
```

In Listing 1 (lines 6–8), *$cf_name* is concatenated directly into the SQL statement and utilized as a parameter in the non-static method invocation, *$SQL->WHERE_and()*. On line 10, the instantiation of *$blogs_Results* is performed by calling the constructor of the Results class. Subsequently, on line 11, the *run_query* method of the Results class is called *$blogs_Results*. An examination of the Results class definition (Listing 1, lines 24–32) reveals that, within the class method *run_query()* (Listing 1, lines 28–31), the class attribute *$this->sql* is transferred as a parameter to the non-static method, *$DB->query()*. According to Listing 1, line 2, *$DB*, a global variable, is an instance of the DB class (Listing 2, lines 6–15).

In Listing 2, the *query* method (lines 11–14) conveys the *$query_SQL* parameter to the non-static method invocation *$this->dbhandle.query()* (Listing 2, line 12). As inferred from the DB class's constructor (*__construct()*), *$this->dbhandle* is an instance of the mysqli class, making *$this->dbhandle.query()* a raw, security-sensitive method call.

Finally, *$this->request.post()* extracts user input data from global variables. The resultant value is assigned to the variable *$cf_name*, which is then incorporated into the SQL statement and transferred via *$SQL->WHERE_and()* to the SQL class attribute *$this->where*. Then, *$SQL->get()* directs the value of *$this->where* as a parameter to the *$blogs_Results* instance attribute *$sql*. *$blogs_Results* executes the SQL statement containing user input by invoking the security-sensitive method, *Mysqli->query()*, through the *run_query()* method.

3.5. Motivation

To summarize the provided instance, the dynamic characteristics present in object-encapsulated applications make the direct retrieval of an instance object's class challenging. However, these applications, characterized by a unified architecture and relatively standard language conventions, enable us to indirectly ascertain their corresponding classes by parsing the variable or class field declaration type from annotations, thereby constructing method call edges. Moreover, such applications encapsulate data through instance attributes, obscuring potential vulnerabilities from detection through data flow graph traversal.

Given the example, we have synthesized the existing challenges and proposed solutions for static analysis in detecting vulnerabilities in contemporary web applications.

To uncover the aforementioned SQL injection vulnerability, we must:

- Implement inter-procedural data flow tracking as many critical method calls transpire across multiple call points, creating an accurate call graph.
- Ensure the analysis maps field variables and context temporary variables within the control flow graph to the same object or value (e.g., *$SQL->sql==$_t2*).
- Incorporate program dependence graphs with inter-procedural taint analysis to trace the variable access paths.

In summary, modern multilayer web applications heavily use object-oriented code and its encapsulated, dynamic, and additional features, leaving object types unclear, resulting in the need for existing tools to help build complete call graphs based on method calls and perform comprehensive inter-procedural analysis. The tainted information is encapsulated in a large amount of class code, resulting in false positives due to the inability of traditional data flow analysis to discover hidden vulnerable data flows. The existing implementation

of a Code Property Graph aims to construct a complete graph structure that represents code syntax and semantics and performs inter-procedural data flow analysis through graph traversal techniques. However, constructing the Code Property Graph does not infer the object type, making it impossible to perform a complete inter-procedural analysis. Graph traversal techniques use the Code Property Graph to construct vulnerability paths based on a generic taint tracking method that fails to discover the taint information encapsulated in object attributes and can lead to false positives. Moreover, such graph traversal methods based on security-sensitive node backtracking data flow graphs are prone to path explosion and low efficiency. Therefore, it is necessary to improve the existing Code Property Graph construction process and optimize the graph traversal algorithm to alleviate the problems of false positives and low vulnerability detection efficiency in existing static analysis tools. Specifically, we identify the following challenges:

3.6. Challenges

Object Oriented. Multi-layer Web applications extensively adopt object-oriented programming (OOP), executing features via method invocations. Particularly, non-static methods are invoked through variables instantiated from corresponding classes. We must identify the specific types of these instantiated variables to construct a comprehensive call graph within the Code Property Graph. However, many applications rely heavily on object-oriented programming and dynamic features, revealing specific types only at runtime. As illustrated in Listing 1 (lines 20–21), it is clearly challenging to directly obtain these types through static analysis. On the other hand, the encapsulation feature of object-oriented programming obscures data dependencies, making it difficult for us to discover defects by retracing the data flow.

Performance. In many contemporary applications, numerous features necessitate a series of interdependent steps, typically completed using distinct modules. If a security-sensitive function resides deep within these modules, the crossing of intermediate data flow nodes can generate many data flow paths. Additionally, to accurately identify the data flow path from user input to a security-sensitive function/method, it is essential to recognize the name of the security-sensitive method and analyze the instantiation of the variable calling it. Directly integrating these tasks into the vulnerability path discovery algorithm would inevitably amplify its complexity and diminish its efficiency.

Scalability. Besides utilizing security-sensitive functions, modern applications also encapsulate security-sensitive methods, as exemplified by the *$this->dbhandle* attribute of the DB class in Listing 2. Designing specific data flow path-finding algorithms for all security-sensitive functions/methods presents a challenge. To maintain scalability, we categorize security-sensitive operations according to their existing forms into security-sensitive functions and methods and design a unified vulnerability path traversal algorithm for them. By introducing an attack dictionary, users can configure corresponding security-sensitive functions or methods according to the type of vulnerability without changing the algorithm. For instance, we can set the *query()* method of the DB class in Listing 2 as a security-sensitive method and perform traversal.

4. Implementation Overview

We aim to design and implement a unified, scalable, and efficient vulnerability path discovery method based on an enhanced Code Property Graph for modern multilayered web applications, considering their object-oriented encapsulation characteristics.

Based on this method, we have developed a tool—VulPathsFinder. To address the issue of scalability, our tool is divided into three steps: (1) conversion from source code to three-address code; (2) construction of an enhanced Code Property Graph; (3) identification of vulnerability paths. The architecture of VulPathsFinder is shown in Figure 1.

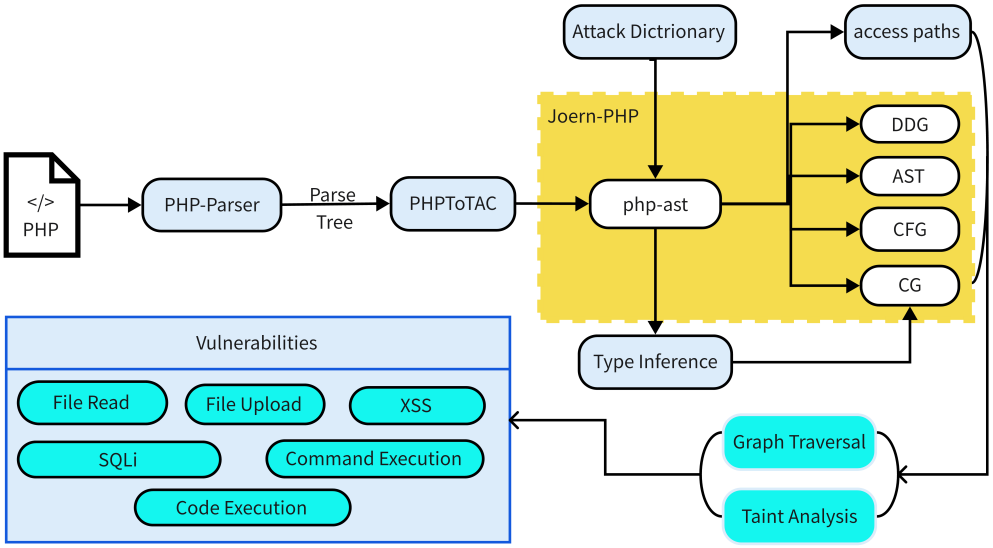

Figure 1. The architecture of VulPathsFinder. The components with gray represent modifications or additions to the original architecture of Joern-PHP.

The first step takes the source code of the given application as input, parses it into an Abstract Syntax Tree (AST), and then uses the PHPToTAC tool to transform the AST into source code in three-address code form while maintaining the directory structure of the original project files.

The second step involves constructing an enhanced Code Property Graph. This stage processes the code files in three-address code form, parsing the statements within the files to construct control flow graphs and program dependence graphs. To tackle the challenges of object orientation, we perform type inference based on the control flow graph to obtain type information of instance objects and variable access paths. We then add the call relationships from call points to method definitions to the call graph, facilitating subsequent inter-procedural analysis. Additionally, during the type inference process, we establish data dependency edges (USE-DEF relationships) between function or method return values and call points, thus refining the program dependence graph. We introduce an attack dictionary that includes information about security-sensitive functions and mapping from recognized security-sensitive methods to classes to address performance issues and scalability. During the type inference process, we use the attack dictionary to mark security-sensitive functions and method call nodes and user input nodes, thereby avoiding querying the Code Property Graph for security-sensitive operations and user input nodes.

The third step involves constructing the vulnerability path. By taking the security-sensitive function/method nodes as starting points for data flow backtracking and performing taint analysis, we can trace the propagation paths of taints through marked user input nodes and variable access paths, thereby identifying vulnerability paths from security-sensitive nodes to user inputs.

4.1. PHP Parser

To analyze the latest version of PHP, we use PHP-Parser [38] as a library to generate parse trees from PHP source code. This library updates continuously with each PHP version, making subsequent updates and maintenance convenient. The parser can be used for static analysis, code manipulation, and any other application that programmatically handles code. The parser constructs an Abstract Syntax Tree (AST) of the code, enabling it to be handled abstractly and robustly.

There are other methods for handling source code. One natively supported by PHP is using tokens generated by 'token_get_all'. The token stream is much lower-level than the AST and therefore has different applications: it also allows for the analysis of the exact

format of a file. On the other hand, token streams are more difficult to handle for more complex analyses. For instance, the AST abstracts out the fact that in PHP, variables can be written as *$foo*, *$$bar*, *${'foobar'}*, or even *${!${''}=barfoo()}*. You do not have to worry about identifying all the different syntaxes from the token stream.

In addition to the parser itself, this package also bundles support for several other related functionalities:

- Pretty-printing support, which is the action of transforming the AST into PHP code;
- Support for serializing and deserializing node trees to JSON;
- Support for dumping node trees in a human-readable form;
- Basic infrastructure for traversing and modifying the AST (node traversers and node visitors);
- Node visitors for parsing namespace names.

4.2. PHPtoTAC

In addition to introducing a new PHP parser, we have modified the intermediate model's construction to accommodate object-oriented features and alias analysis. As we have introduced a new PHP parser to construct the Abstract Syntax Tree (AST), we leverage the node traversal functionality of PHP-Parser based on this AST to inspect each node. This module [39] tracks various types of expressions (assignments, function calls, method calls, actual parameters, formal parameters, etc.) and monitors function definitions, class definitions, object references, method definitions, variables, namespaces, and interfaces. PHP-Parser transforms these expressions and statement nodes into a three-address code format through traversal tracking, which allows us to perform alias analysis and variable data tracking on the transformed code. Consequently, we can perform inter-procedural data flow analysis when conducting taint analysis.

4.3. Code Property Graph

After transforming the PHP source code into a three-address code format based on PHP-Parser, we use it as the source code input for Joern-PHP to construct the Code Property Graph (CPG). First, we parse the transformed three-address code into an AST, and then each file in the AST is traversed to construct the control flow graph. This control flow graph is then traversed to analyze alias to obtain variable alias information. Based on comments and instance creation statements, we infer instance types of variables, parse method call points, and create method call edges to improve the call graph.

On top of Joern-PHP, we add alias analysis and variable instance type inference to create method call edges. The pseudocode of this process is shown in Algorithms 1 and 2:

Algorithms 1 and 2 show the process of type inference and call graph construction that we added based on Joern-PHP.

In Algorithm 1, the process of traversing is called the Abstract Syntax Tree (AST) (line 4). Lines 5–22 in Algorithm 1 show the process of traversing the control flow graph to perform context-sensitive alias analysis. This procedure traverses the control flow graph and parses the statement's assignment, store, and load operations. By this parsing procedure, on the one hand, we can infer the directional information of the variables, which is the type information. On the other hand, we can construct access paths for variables. Line 23 in Algorithm 1 shows the process of building the call graph based on the results of the first two steps.

Algorithm 1 Type inference and call graph construction (1)

Input:
1: AST: Abstract Syntax Tree; CFG: Control Flow Graph
Output:
2: pts: Point-to information; CG: Call Graph
3: $methodCalls = [], instanceVars = []$
4: $parseAST(AST)$
5: **for** $function\ in\ CFG.functions$ **do**
6: $Statements = function.getVertices()$
7: **for** $stmt\ in\ statements$ **do**
8: $context = dispatch(stmt)$
9: $Var = parseVar(stmt)$
10: **if** $context : var\ in\ instanceVars$ **then**
11: **if** $stmt\ is\ var.f\ =\ y$ **then**
12: $AddEdge(context : y, context : var.f)$
13: **end if**
14: **if** $stmt\ is\ y\ =\ var.f$ **then**
15: $AddEdge(context : var.f, context : y)$
16: **end if**
17: **end if**
18: **if** $stmt\ is\ x\ =\ y$ **then**
19: $AddEdge(context : y, context : x)$
20: **end if**
21: **end for**
22: **end for**
23: $ProcessCalls(methodCalls)$

The detail of *parseAST* is shown in Algorithm 2 (lines 3–21). It mainly parses instance creation statements (Algorithm 2, lines 5–9), method calls (Algorithm 2, lines 10–15), and Global statements (Algorithm 2, lines 16–19). Global statements reflect the scope of variables, which affects our type inference for the corresponding variables. The process *ProcessCalls* (Algorithm 2, lines 28–39) first parses the instance variable and method in the method call, then obtains the class corresponding to the instance variable from the variable direction information through the instance variable name and then connects the call statement with the corresponding method definition node with a call edge. In addition, we establish a data dependency edge between the actual argument variable of the call point and the formal parameter of the corresponding method (Algorithm 2, lines 34–36). If the method has a return value, a data dependency edge is also established between the return value node and the left value of the call statement (Algorithm 2, line 37).

It is essential to note that Algorithms 1–3 have some functions for which the specific code is not shown. These functions are functional and represent a process. We describe their functionality by adding comments at the corresponding locations of the algorithm.

After applying the above process to the example code, we obtain the improved Code Property Graph and the variable access paths, as shown in Figure 2:

Algorithm 2 Type inference and call graph construction (2)

Input:
AST: Abstract Syntax Tree; CFG: Control Flow Graph
Output:
pts: Point-to information; CG: Call Graph
function PARSEAST(AST)
  for *node in AST.nodes* **do**
    if *node is Assign and node.right is New* **then**
      $context = dispatch(node)$ // Get the context information of the node
      $var, class = parseNew(node)$ // Get the variable name and the class
      $Add < context : var, [class] > to\ instanceVars$
    end if
    if *node is Global* **then**
      $context = dispatch(node)$
      $Var = parseGlobal(node)$ // Get the variable name of the global declaration
      $Classes = Get\ objects\ from\ instanceVars$
      $Add < context : var, classes > to\ instanceVars$
    end if
    if *node is MethodCall* **then**
      $context = dispatch(node)$
      $Add < context : var, [methodCall] > to\ methodCalls$
    end if
  end for
end function
function ADDEDGE(s, t)
  if $s \rightarrow t$ *not in pts* **then**
    $Add\ s \rightarrow t\ to\ pts$
    $instanceVars(s) \cup = \ instanceVars(t)$
  end if
end function
function PROCESSCALLS($methodCalls$)
  for *methodCall in methodCalls* **do**
    $Context, method \ = \ dispatch(methodCall)$
    $objectVar \ = \ parseVar(methodCall)$ // Parse the object from the method call
    $class \ \leftarrow \ get\ class\ from\ instanceVars\ based\ on\ the\ objectVar)$
    $add < methodCall \rightarrow class : method > \ to\ CG$
    for *parameter* $p_i\ of\ m$ **do**
      $AddEdge(context : a_i, p_i)$
    end for
    $AddEdge(context : m_{ret}, context : r)$
  end for
end function

As shown in Figure 2, we have improved the call graph by constructing call edges between the call sites and methods through type inference and perfected the Program Dependency Graph (PDG) [6] by establishing data dependencies between the method return value and the left value of the call point. Meanwhile, we construct the right-hand side variable access path graph by obtaining the type information of the instance and the data propagation path between variables during type inference. It includes the following elements: user input tag nodes, USE-DEF relations between variables, inter-procedural data propagation, and security-sensitive call tags. Furthermore, we use dotted lines to connect data field nodes representing the same instance encapsulation. Although they are in different contexts or processes, they represent the same data and there is truly no USE-DEF-defined program dependence between them. Hence, we use the dotted line to connect them. In this way, we obtain a complete variable access path, which can guide us to construct a complete vulnerability path.

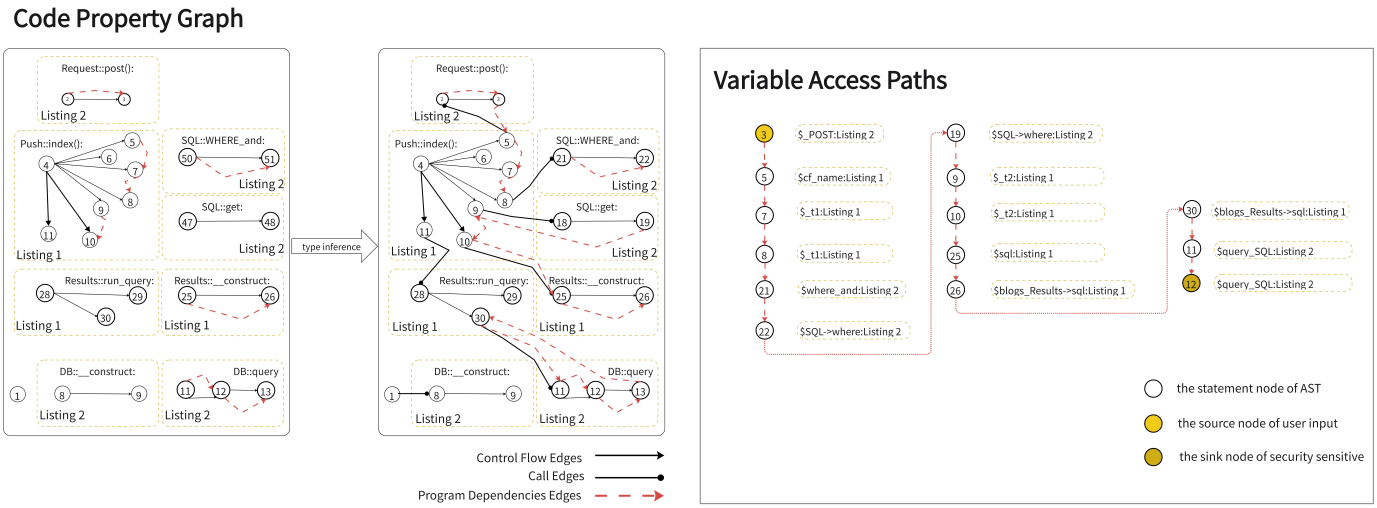

Figure 2. The Code Property Graph of the code example.

4.4. Graph Traversal

When constructing the Code Property Graph, we label user inputs and security-sensitive calls based on the attack dictionary. In this way, we avoid checking user inputs and querying safety-sensitive calls based on graph traversal during graph traversal. Since graph traversal techniques based on security-sensitive calls are prone to path explosion in recursive backtracking data flows, we modify the procedure for constructing vulnerable paths through recursive backtracking. The updated algorithm is shown in Algorithm 3:

Algorithm 3 Taint analysis based on propagation paths and graph traversal

1: $Sinks = GetSinks()$ // Get the sink nodes from CPG
2: AllPaths = []
3: **for** *sink in sinks* **do**
4: $paths = sink.loop().in("REACHES").loop(node.maker! = "source")$ // Traverse the variable access paths
5: $stmt\_paths \leftarrow retrieve(paths)$ //Retrieve statement node from the variable access path
6: $AllPaths.addAll(stmt\_paths)$
7: **end for**

Algorithm 3 is based on variable access paths and sink markers for vulnerability path construction, which is still a graph traversal process, but no longer based on data flow backtracking in the program dependency graph. First, we query the sink markers from the Code Property Graph to obtain the sink nodes (line 1). Based on the obtained sinks, we obtain the tainted variables, locate the variable nodes in the variable access paths, and then backtrack on the variable access paths with the sink as the starting node. If we encounter a user input marker, we stop backtracking (lines 3–4). At this point, we have extracted the variable access path from user input to safety-sensitive calls. Finally, we restore the variable nodes in these variable access paths to the statement nodes in the corresponding CPG, thus obtaining the vulnerability path (lines 5–6). Compared with the graph traversal algorithm of Joern-PHP's vulnerability path discovery, Algorithm 3 dramatically simplifies the graph traversal process. Still, it does not mean that our tool simplifies the entire process of vulnerability path discovery. As shown in Figure 2, we construct variable access paths, mark user input nodes, and mark security-sensitive operations while traversing the control flow graph. Since traversing the control flow graph is necessary, we perform some additional operations during this process to simplify the final graph traversal process.

5. Evaluation

5.1. Dataset

We evaluate VulPathsFinder on ten applications developed based on open-source frameworks or MVC structures, as shown in Table 1. Our selection criteria for the applications were as follows: (1) we evaluated popular, large-scale applications with unified framework structures, regardless of whether they were independently developed or complex applications developed based on open-source frameworks, and the most recent versions with reported vulnerabilities were included; (2) we compared the same test applications used by tools for static vulnerability detection.

Table 1. The list of MVC-Based and Non-MVC projects.

Categories	Projects/Version
MVC-Based	osTicket v1.11
	Open Web Analytics v1.73
	PrestaShop v8.0.4
	LimeSurvey v3.17.13
	WebSvn v2.6.0
	piwigo v11.3.0
	b2evolution v7.2.2
	lansuite v2.1.0
	qdPM v9.1
	impresscms v1.4.2
Non-MVC	pfBlockerNG v2.1.4
	SPIP v3.1.1
	Bus Pass Management System v1.0
	elFinder v2.1.47
	DomainMod v4.13
	60CycleCMS 2.5.2
	Webtareas v2.0
	SPIP v4.2.0
	SQLiteManager 1.2.0
	Webutler v3.2

5.2. Setup

We implemented PHPtoTAC [39] to convert the source code into three address code formats with PHP-Parser [38] and implemented the static analysis tool VulPathsFinder based on Joern-PHP [40]. We deployed VulPathsFinder on Ubuntu 22.04.2 LTS with 4-cores of 2.6 GHz each and 50 GB RAM. To evaluate VulPathsFinder, we compared its precision and recall with four PHP static analysis tools: Joern-PHP [6], ooPIXY [3], and RIPS [2]. We conducted experiments using real-world open-source PHP applications. A comparison between VulPathsFinder and Joern-PHP was performed to assess the improvement brought by the Joern-PHP-based enhancement. As RIPS and ooPixy support OOP features, the comparison with VulPathsFinder provides significant insights into OOP support. However, the latest version of RIPS is commercial, non-open source, and does not provide an academic license. Therefore, the most recent open-source version, 0.55, was employed in our evaluation.

Throughout the entire evaluation process, we used the following definitions: precision (P), recall (R), and $F_{measure}$ (F) [3]. Precision is the ratio of the number of true

positives (*TP*) to the number of reported errors, including both reported true positives and false positives (*FP*):

$$P = \frac{TP}{TP + FP}. \tag{1}$$

Recall is the ratio of the number of true positives to the actual number of errors, which includes both reported true positives and false negatives (*FN*, those we failed to detect).

$$R = \frac{TP}{TP + FN}. \tag{2}$$

$F_{measure}$ provides a comprehensive measurement standard combining precision and recall. The $F_{measure}$ ranges between 0 and 1 for a given tool. We will use P, R and $F_{measure}$ to provide accuracy rankings for the tools studied.

$$F_{measure} = \frac{(2 * P * R)}{P + R}. \tag{3}$$

5.3. Summary of Results

We selected 10 MVC architecture-based applications from the dataset as the test set to evaluate the differences between VulPathsFinder and Joern PHP regarding building Code Property Graphs, mainly in two respects: (1) the changes in the types of nodes and edges in the Code Property Graph before and after improvement; (2) the time efficiency comparison between the two in constructing the Code Property Graph.

As shown in Table 2, we made comparisons on the same dataset, which includes ten projects and a total of 1,336,649 lines of code. First, we can see that the number of AST nodes and edges and the number of CFG edges are consistent between the two, as we used php-ast to parse the same dataset in the same way. The main difference lies in the number of calls and program dependence edges. Then, the number of call edges is larger in VulPathsFinder than in Joern-PHP since we have added method call edges to the call graph. In addition, the number of program dependence edges also increases compared to Joern-PHP because, when processing method calls, we simultaneously processed the USE-DEF relations for the return values of functions and methods with left values at the call point, thus increasing the number of program dependence edges.

Table 2. Dataset and graph sizes.

Items	Joern-PHP	VulPathsFinder
# of projects	10	10
# of php files	6923	6923
# of LOC	1,336,649	1,336,649
# of AST nodes	3,936,057	3,936,057
# of AST edges	3,880,479	3,880,479
# of CFG edges	611,351	611,351
# of PDG edges	25,946	291,881
# of call edges	5328	6459
# of times	12	13

The last row in Table 2 shows the time consumed by the two tools to construct the Code Property Graph on the same dataset. It follows that the times are the same for both. This is mainly because the time consumed in the whole process is due to traversing the AST nodes and the control flow graph, neither of which is changed. Moreover, adding program dependence edges and call edges only increases the computational load by a constant amount with respect to the entire AST and CFG traversal process, so they do not significantly affect the final total runtime.

In addition to the dataset composed of modern applications using the frameworks described in Table 2, we have collected another dataset composed of popular applications that do not use a unified framework. This dataset is used to compare the impact of vulnerability detection by VulPathsFinder and Joern-PHP on non-framework and framework-based modern applications. The experimental results are as shown in Table 3.

In contrast to OOPixy and RIPS, PHPAudit and Joern-PHP can detect most vulnerabilities. OOPixy is a tool developed based on Pixy to adapt to the object-oriented nature of PHP. It only supports the detection of SQLi and XSS vulnerabilities, and the adapted PHP version is shallow (PHP 5). We did not obtain efficient output results when running this tool for vulnerability detection. When debugging, we commonly encounter syntax errors when parsing PHP code to generate intermediate code, such as access modifiers for classes, due to the inability to parse some code. In addition, this tool performs taint analysis by identifying user input and security-sensitive functions based on control flow graphs. Encapsulating user input and security-sensitive functions prevent a complete inter-procedural analysis. The above issues caused OOPixy not to output valid detection results. RIPS detects additional vulnerabilities to OOPixy and is currently maintained as a commercial project. Its open-source version is only 5.5. While it models numerous built-in functions in PHP and constructs vulnerability information flow paths based on control flow graphs, it does not support object-oriented code analysis and takes too long to process complex encapsulated applications, resulting in an inability to output effective results. We focus on the vulnerability detection performance of VulPathsFinder and Joern PHP on two datasets.

As shown in Table 3, there is a slight difference in the vulnerability detection results between Joern-PHP and VulPathsFinder for non-MVC applications. The vulnerability detection logic is slightly different between the two. The main difference is that we perform a three-address code conversion on the source code input, eliminating loop logic and dramatically reducing the number of paths traversed by the graph. Furthermore, we improve the vulnerability detection algorithm based on graph traversal. VulPathsFinder uses tags and access paths for vulnerability detection, which also plays a role in pruning graph traversal and alleviating the low-efficiency problem in graph traversal techniques. The column *Geometric Mean* displays the geometric mean of the application vulnerability detection results based on the MVC architecture. We can see from the values in this column that VulPathsFinder performs better than Joern-PHP regarding accuracy and time consumption in vulnerability detection for MVC-based applications. For MVC-based applications, VulPathsFinder adds variable type inference, which adds additional method call edges to the call graph by inferring object types. It also improves the program dependence graph and alleviates the problem of incomplete inter-procedural analysis in the Code Property Graph. At the same time, it also maintains the original scalability of Joern-PHP. Based on the above improvements, VulPathsFinder and Joern-PHP compare their vulnerability detection performance for MVC-based applications. Joern-PHP significantly improves accuracy and reduces the time spent on vulnerability detection.

Table 3. The experimental results of vulnerability. *T* denotes the running time (s); *TF*, *FP*, *P*, *R*, and *F* are the same as described in Section 5.2; a dash means the tool does not support the detection of the vulnerability.

Category	Subject	#Projects	VulPathsFinder						OOPixy						RIPS						Joern-PHP					
			TP	*FP*	*P*	*R*	*F*	*T*	*TP*	*FP*	*P*	*R*	*F*	*T*	*TP*	*FP*	*P*	*R*	*F*	*T*	*TP*	*FP*	*P*	*R*	*F*	*T*
Non-MVC	SQL Injection	4	4	3	0.56	1.00	0.72	138	0	0	0.00	0.00	-	3	0	0	0.00	0.00	-	30	4	3	0.57	1.00	0.73	201
	Command Injection	1	1	2	0.33	1.00	0.50	17	0	0	0.00	0.00	-	1	0	0	0.00	0.00	-	23	1	2	0.33	1.00	0.50	38
	Cross-Site Script	1	1	3	0.25	1.00	0.40	24	0	0	0.00	0.00	-	2	0	0	0.00	0.00	-	38	1	3	0.25	1.00	0.40	50
	File Read	1	1	0	1.00	1.00	1.00	51	-	-	-	-	-	1	0	0	0.00	0.00	-	67	1	0	1.00	1.00	1.00	97
	File Upload	1	1	0	1.00	1.00	1.00	50	-	-	-	-	-	1	0	0	0.00	0.00	-	65	1	0	1.00	1.00	1.00	85
	Code Execution	2	2	0	1.00	1.00	1.00	29	0	0	0.00	0.00	-	3	0	0	0.00	0.00	-	25	2	0	1.00	1.00	1.00	53
MVC-Based	SQL Injection	4	4	1	0.80	1.00	0.89	139	0	0	0.00	0.00	-	3	0	0	0.00	0.00	-	42	2	1	0.67	0.50	1.00	172
	Command Injection	1	1	1	0.50	1.00	0.67	19	0	0	0.00	0.00	-	1	0	0	0.00	0.00	-	36	0	0	0.00	0.00	-	34
	Cross-Site Script	1	1	1	0.50	1.00	0.67	23	0	0	0.00	0.00	-	2	0	0	0.00	0.00	-	47	1	1	0.50	1.00	0.67	39
	File Read	1	1	0	1.00	1.00	1.00	51	-	-	-	-	-	1	0	0	0.00	0.00	-	73	0	0	0.00	0.00	-	38
	File Upload	1	1	0	1.00	1.00	1.00	52	-	-	-	-	-	1	0	0	0.00	0.00	-	69	0	0	0.00	0.00	-	37
	Code Execution	2	2	0	1.00	1.00	1.00	30	0	0	0.00	0.00	-	2	0	0	0.00	0.00	-	39	0	0	0.00	0.00	-	27
	Geometric Mean				0.76	1.00	0.86	40.36	-	-	0.00	0.00	-	1.57	-	-	0.00	0.00	-	42.99	-	-	0.58	0.71	0.94	57.99

6. Conclusions

The extensive use of object-oriented code in multi-layer modern web applications poses a challenge for vulnerability mining from a static analysis perspective, with features such as encapsulation, inheritance, and polymorphism. A large amount of object-oriented code makes it impossible to construct a complete call graph using simple function name-matching methods, resulting in the inability to perform a comprehensive inter-procedural analysis. The encapsulated nature of the class makes the data hidden in the object properties, making it impossible to obtain the vulnerability path via universal data flow analysis. To alleviate the above issues, we improve the Code Property Graph by adding alias analysis capabilities to provide a foundation for full inter-procedural analysis based on the Code Property Graph. Moreover, to alleviate the efficiency issues of graph traversal techniques, we propose a vulnerability path construction method based on variable access paths to improve the efficiency of vulnerability discovery. However, we must admit that the deeply dynamic nature of modern PHP applications makes static analysis techniques unable to fully solve the problem of object type inference, and the results generated by VulPathsFinder still require manual verification. In future work, we will conduct dynamic testing studies based on the detection results of VulPathsFinder to avoid manual verification and achieve fully automated vulnerability detection and verification.

Author Contributions: Conceptualization, S.Q. and C.Z.; methodology, C.Z.; software, C.Z.; validation, C.Z., T.T. and C.W.; formal analysis, C.Z. and S.Q.; investigation, C.Z.; resources, C.W.; data curation, C.Z.; writing—original draft preparation, C.Z.; writing—review and editing, C.Z. and S.Q.; visualization, C.Z.; supervision, T.T.; project administration, C.Z.; funding acquisition, S.Q. All authors have read and agreed to the published version of the manuscript.

Funding: This work is supported by National Key R&D Program of China under Grant 2021YFB2700400.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CPG	Code Property Graph
CG	Call Graph
PDG	Program Dependence Graph
CFG	Control Flow Graph

References

1. Jovanovic, N.; Kruegel, C.; Kirda, E. Pixy: A static analysis tool for detecting web application vulnerabilities. In Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06), Berkeley/Oakland, CA, USA, 21–24 May 2006; p. 6.
2. Dahse, J.; Holz, T. Simulation of Built-in PHP Features for Precise Static Code Analysis. In Proceedings of the NDSS, San Diego, CA, USA, 23–26 February 2014; Volume 14, pp. 23–26.
3. Nashaat, M.; Ali, K.; Miller, J. Detecting Security Vulnerabilities in Object-Oriented PHP Programs. In Proceedings of the 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM), Shanghai, China, 17–18 September 2017; pp. 159–164.
4. Nunes, P.J.C.; Fonseca, J.; Vieira, M. phpSAFE: A security analysis tool for OOP web application plugins. In Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Rio de Janeiro, Brazil, 22–25 June 2015; pp. 299–306.
5. Hauzar, D.; Kofroň, J. Weverca: Web applications verification for php. In Proceedings of the Software Engineering and Formal Methods: 12th International Conference, SEFM 2014, Grenoble, France, 1–5 September 2014; Proceedings 12; Springer: Berlin/Heidelberg, Germany, 2014; pp. 296–301.

6. Backes, M.; Rieck, K.; Skoruppa, M.; Stock, B.; Yamaguchi, F. Efficient and flexible discovery of php application vulnerabilities. In Proceedings of the 2017 IEEE European Symposium on Security and Privacy (EuroS&P), Paris, France, 26–28 April 2017; pp. 334–349.
7. Alhuzali, A.; Gjomemo, R.; Eshete, B.; Venkatakrishnan, V. NAVEX: Precise and scalable exploit generation for dynamic web applications. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, USA, 15–17 August 2018; pp. 377–392.
8. Zhao, J.; Gong, R. A new framework of security vulnerabilities detection in PHP web application. In Proceedings of the 2015 9th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, Santa Catarina, Brazil, 8–10 July 2015; pp. 271–276.
9. Livshits, B.; Nori, A.V.; Rajamani, S.K.; Banerjee, A. Merlin: Specification inference for explicit information flow problems. *ACM Sigplan Not.* **2009**, *44*, 75–86. [CrossRef]
10. Lee, T.; Wi, S.; Lee, S.; Son, S. FUSE: Finding File Upload Bugs via Penetration Testing. In Proceedings of the NDSS, San Diego, CA, USA, 23–26 February 2020.
11. Balzarotti, D.; Cova, M.; Felmetsger, V.; Jovanovic, N.; Kirda, E.; Kruegel, C.; Vigna, G. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP 2008), Oakland, CA, USA, 18–22 May 2008; pp. 387–401.
12. Park, S.; Kim, D.; Jana, S.; Son, S. FUGIO: Automatic Exploit Generation for PHP Object Injection Vulnerabilities. In Proceedings of the 31st USENIX Security Symposium (USENIX Security 22), Boston, MA, USA, 10–12 August 2022; pp. 197–214.
13. Eriksson, B.; Pellegrino, G.; Sabelfeld, A. Black widow: Blackbox data-driven web scanning. In Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 24–27 May 2021; pp. 1125–1142.
14. Li, P.; Meng, W.; Lu, K.; Luo, C. On the feasibility of automated built-in function modeling for php symbolic execution. In Proceedings of the Web Conference 2021, Ljubljana, Slovenia, 19–23 April 2021; pp. 58–69.
15. Aho, A.V.; Sethi, R.; Ullman, J.D. *Compilers: Principles, Techniques, and Tools*; Addison-Wesley Reading: Reading, MA, USA, 2007; Volume 2.
16. Sridharan, M.; Chandra, S.; Dolby, J.; Fink, S.J.; Yahav, E. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 196–232.
17. Khedker, U.; Sanyal, A.; Sathe, B. *Data Flow Analysis: Theory and Practice*; CRC Press: Boca Raton, FL, USA, 2017.
18. Xie, Y.; Aiken, A. Static Detection of Security Vulnerabilities in Scripting Languages. In Proceedings of the USENIX Security Symposium, Vancouver, BC, Canada, 31 July–4 August 2006; Volume 15, pp. 179–192.
19. Wassermann, G.; Su, Z. Sound and precise analysis of web applications for injection vulnerabilities. In Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, San Diego, CA, USA, 10–13 June 2007; pp. 32–41.
20. Wassermann, G.; Su, Z. Static detection of cross-site scripting vulnerabilities. In Proceedings of the 30th International Conference on Software Engineering, Leipzig, Germany, 10–18 May 2008; pp. 171–180.
21. Jovanovic, N.; Kruegel, C.; Kirda, E. Static analysis for detecting taint-style vulnerabilities in web applications. *J. Comput. Secur.* **2010**, *18*, 861–907. [CrossRef]
22. Dahse, J.; Holz, T. Static Detection of Second-Order Vulnerabilities in Web Applications. In Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14), San Diego, CA, USA, 20–22 August 2014; pp. 989–1003.
23. Olivo, O.; Dillig, I.; Lin, C. Detecting and exploiting second order denial-of-service vulnerabilities in web applications. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; pp. 616–628.
24. Yu, F.; Alkhalaf, M.; Bultan, T. Stranger: An automata-based string analysis tool for php. In Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems: 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, 20–28 March 2010; Proceedings 16; Springer: Berlin/Heidelberg, Germany, 2010; pp. 154–157.
25. Alhuzali, A.; Eshete, B.; Gjomemo, R.; Venkatakrishnan, V. Chainsaw: Chained automated workflow-based exploit generation. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 641–652.
26. Yamaguchi, F.; Golde, N.; Arp, D.; Rieck, K. Modeling and discovering vulnerabilities with Code Property Graphs. In Proceedings of the 2014 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 18–21 May 2014; pp. 590–604.
27. Weihl, W.E. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Las Vegas, NV, USA, 28–30 January 1980; pp. 83–94.
28. Hind, M. Pointer analysis: Haven't we solved this problem yet? In Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Snowbird, UT, USA, 18–19 June 2001 ; pp. 54–61.
29. Heintze, N.; Tardieu, O. Demand-driven pointer analysis. *ACM Sigplan Not.* **2001**, *36*, 24–34. [CrossRef]
30. Milanova, A.; Rountev, A.; Ryder, B.G. Parameterized object sensitivity for points-to and side-effect analyses for Java. In Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, Roma, Italy, 22–24 July 2002; pp. 1–11.

31. Sridharan, M.; Bodík, R. Refinement-based context-sensitive points-to analysis for Java. *ACM SIGPLAN Not.* **2006**, *41*, 387–400. [CrossRef]
32. Hardekopf, B.; Lin, C. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, San Diego, CA, USA, 10–13 June 2007; pp. 290–299.
33. Zhang, Q.; Lyu, M.R.; Yuan, H.; Su, Z. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, Seattle, WA, USA, 16–19 June 2013; pp. 435–446.
34. Lei, W.; Feng, L.; Lian, L.; Xiaobing, F. Principle and practice of taint analysis. *J. Softw.* **2017**, *28*, 860–882.
35. Li, P.; Meng, W. Lchecker: Detecting loose comparison bugs in php. In Proceedings of the Web Conference 2021, Ljubljana, Slovenia, 19–23 April 2021; pp. 2721–2732.
36. Skoruppa, M. b2evolution 7-2-2 - 'cf_name' SQL Injection. 2021.
37. Rookie, D. Remote Code Execution Vulnerability #238. 2020.
38. Popov, N. nikic/PHP-Parser: A PHP Parser Written in PHP. 2018.
39. Zhao, C. PHPCodeToTAC. 2023.
40. Skoruppa, M. PHPCodeToTAC. 2017.