DEPHP: A Source Code Recovery Method for PHP Bytecode with Improved Structural Analysis

Shiwu Zhao[1,2], Ningjun Zheng[3], Haoyu Li[1,2], Ruizhi Feng[1,2], Xingchen Chen[1], Ru Tan[1,2](✉), and Qixu Liu[1,2]
[1]*Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China*
[2]*School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China*
[3]*Tencent Technology (Shanghai) Co., Ltd, Shanghai, China*
[1,2]{zhaoshiwu, lihaoyu2023, fengruizhi, chenxingchen, tanru, liuqixu}@iie.ac.cn
[3]ningjzheng@tencent.com

***Abstract*—Over the past decade, PHP has consistently been one of the most popular server-side programming languages among developers for web development. To protect intellectual property, various PHP source code obfuscation and encryption methods have been developed, which has led to difficulties in performing security analysis on PHP source code. Previous work has demonstrated the feasibility of recovering source code by extracting bytecode from PHP during dynamic execution. However, there is still a lack of a universal decompilation method for this kind of bytecode, tailored to PHP's unique syntax.**

Thus, we propose a systematic decompilation framework for PHP bytecode. First, we design a unified intermediate representation that eliminates the differences between bytecodes from different PHP versions. Then, we introduce a structural analysis algorithm specifically for PHP syntax, improving upon existing methods to better accommodate PHP's unique syntax. We use over 3 million lines of PHP code as a dataset and compiled it into PHP bytecode. After decompiling it with our method, we successfully recovered 92% of the classes and 85% of the methods. Furthermore, from the encrypted dataset containing 37 SQL injection and 31 XSS vulnerability patterns, we fully restored the original vulnerability patterns and reconstructed the exploitation chains. Furthermore, we identified a series of vulnerabilities in real-world projects and were assigned 6 new CVE IDs[1], demonstrating the correctness of our method and its ability to assist in static analysis for vulnerability discovery.

***Index Terms*—decompilation, structural analysis, code protection, PHP bytecode, vulnerability detection**

I. Introduction

Besides C and Java, PHP is also an important programming language. According to statistics from W3Techs [1], PHP has become the most popular web server programming language over the past decade. Servers using PHP as a programming language have consistently accounted for over 70% of the market share in the past ten years, far surpassing other programming languages such as Java [2], ASP.NET [3], and Ruby [4].

Unlike compiled languages, PHP is an interpreted language, where its interpreter directly executes the source code. During execution, the source code is first compiled into PHP bytecode, but this compilation process is transparent to the user, and users typically do not need to access PHP bytecode [5].

(✉) : Corresponding Author
[1]CVE-2025-45046, CVE-2025-45047, CVE-2025-45048, CVE-2025-45049, CVE-2025-45050, CVE-2025-45052

In the process of analyzing PHP source code security issues, we sometimes encounter protected PHP code, where the protection mechanisms are typically based on PHP bytecode principles. These mechanisms make PHP security analysis challenging. For example, when static analysis algorithms encounter encrypted PHP code, data flow analysis and control flow analysis become infeasible. Even when bytecode is obtained during PHP execution according to Weißer et al. [6], directly analyzing the obtained PHP bytecode remains difficult. However, there is still a lack of effective decompilation solutions for PHP bytecode. Decompilation technology for PHP bytecode is still in a very early stage. This is mainly due to the unique syntactical features of PHP, which are significantly different from those of other programming languages [7] [8].

Challenge. We summarize challenges for decompilation below PHP's unique syntax.

• ***C1:*** *The lack of a unified linear intermediate representation (IR) for different versions of PHP bytecode.* This is similar to how different processor architectures have different instruction sets. In PHP, each version of the interpreter can only execute bytecode for that specific version and cannot fully be compatible with bytecode from other versions, either upwards or downwards [9]–[11]

• ***C2:*** *The lack of structural analysis algorithms specifically tailored for PHP's unique syntax.* Existing decompilation methods for PHP bytecode rely on direct bytecode pattern matching, while control flow analysis algorithms widely used in decompilation of C [12]–[15] and Java [16] [17] cannot handle PHP's special syntax, such as jumping out of nested loops.

Solution. To address the above issues, we design an intermediate representation for PHP bytecode through in-depth analysis of different versions of PHP bytecode. Additionally, by improving the structural analysis algorithms used in the decompilation of other languages, we develop a complete PHP decompilation framework.

Finally, we systematically evaluate the effectiveness of our decompilation method. In addition, we verify that our method can restore the original vulnerability patterns and exploit chains from encrypted code, demonstrating its ability to assist in the security analysis of encrypted PHP code. At the same

time, we applied our proposed method to real-world scenarios and identified a series of vulnerabilities.

In this paper, we make the following **contributions**.

- **Decompilation Framework:** We propose a method for decompiling PHP bytecode. By studying a total of 226 bytecodes across different PHP versions, we develop an intermediate representation that can be applied to PHP bytecode from various PHP versions. Additionally, we improve the original SESS analysis to support the structural analysis tailored to PHP's unique syntax.
- **Accuracy Validation.** Similar to the decompilation research for C and Java, we design a series of experiments to evaluate the correctness and structuredness of the recovered code. Using eight large open-source PHP projects as a dataset, we conduct experiments on over 3 million lines of PHP code. On average, we are able to recover 92% of the classes and 85% of the methods from the bytecode.
- **Vulnerability Pattern Recovery.** We perform code recovery on the encrypted dataset, which includes 37 SQL injection patterns and 31 XSS patterns. We successfully extract original vulnerability patterns from the encrypted data and reconstruct the vulnerability exploitation chains. In addition, we discover a series of vulnerabilities in real-world scenarios and were assigned 6 CVE identifiers, demonstrating that our method can aid in the analysis of security issues in encrypted PHP code.

The rest of this paper is organized as follows. Section 2 and 3 introduce the research background and the motivation. Section 4 outlines our proposed method. Section 5 provides a detailed explanation of the experimental design used to evaluate our decompilation method, including the accuracy and its ability to recover vulnerability patterns. Section 6 reviews related work on decompilation in other languages. Section 7 concludes with a summary of the overall approach and results.

II. Background

A. PHP Bytecode

PHP is an interpreted language, meaning its source code does not need to be fully compiled into machine code beforehand. It is dynamically parsed and executed by the PHP interpreter at runtime [18]. However, PHP does not rely entirely on directly interpreting the source code during execution. Instead, the source code is first converted into PHP bytecode before execution. When a PHP script is executed, the source code is first processed by lexical analysis and syntax analysis, generating an abstract syntax tree (AST). This AST is then translated into a set of bytecode instructions. These bytecodes are not machine-specific instructions but rather special instructions designed for the Zend Engine [19] [20]. PHP bytecode is a simplified, platform-independent instruction set that is passed to the Zend Engine for interpretation during execution. Each bytecode instruction consists of an opcode, and each opcode is handled by a corresponding handler function in the Zend Engine [21].

B. PHP Source Code Protection Mechanism

Through previous research and our own investigation, there are generally two types of PHP source code protection mechanisms: extension-based code encryption and extension-free code obfuscation [22]. Extension-based code encryption tools typically provide an encoder-decoder pair. The encoder compiles PHP source code into encrypted bytecode files, which cannot be directly read or executed. The decoder is loaded as an extension of the PHP interpreter at runtime, allowing it to decrypt the PHP files and execute them correctly.

Extension-free code obfuscation tools are usually more independent, mainly transforming PHP source code through syntax-level or structure-level changes to make it hard to understand, thus achieving the purpose of protection. These tools do not require installing any additional extension, and the obfuscated code can run in any standard PHP environment.

For example, Fig. 1(a) shows code that has been obfuscated using YAK Pro Obfuscator [23]. Fig. 1(b), on the other hand, shows code encoded by Sourceguardian [24] and represents a common form of encrypted code.

```
<?php
goto qcI_F; hg1pU: XLdKx: goto ZozjN;
ZozjN: if (!($PVF18 <= 7)) { goto C1KO0;
} goto mfUlN; sq6FB: goto XLdKx;
goto n8EY3; z23hU: $saYwN =
"\142\145\141\x75\x74\151\x66\165\x6c\x20";
goto dvSkC; nCx3k: $PVF18 = 0; goto hg1pU;
mfUlN: echo $Hgf5C; goto bMjVZ; qcI_F: $Hgf5C
= "\150\x65\x6c\154\157\x20"; goto T_FPu;
g_9jr: echo PHP_EOL; goto Othpo; Othpo:
```

(a) Obfuscated code

```
+C57qWsPvRE8602LP4JglUi7DWbTvqML5oXpovKBAd/wO/
Osuvij0QQIetpjPJ6cUcWqzIs11bUG0BGHhTTqjRb2S0FO
v2CkZoVZZCljmHiYRpaz
+KCD7wJk5HWgiJMA4RV7zV5pdeij0quqnOBzLEsfxJzXcl
TRv9Wc0Qm2LdRFYUA1JaoVAE1AAAABAEAALdLvHbgRmbSI
S+N793wPpdEQw3sJVK65kKJHhwBF2aE
+99zoBPeHMEbytHzySL9zPhf92
+qOZYOOhrEYknktelzbVBn8Fi8
+mpsIhOEjSUyrv5coWruzrcn9p97VRGMP/
bTzCSz1FkgFMs8v0hT8pZDVNNH/
```

(b) Encrypted code

Fig. 1. Example of PHP protected code

Based on the research by Weißer et al. [6], during the execution of protected PHP code, if the bytecode can be dynamically retrieved, it can be decompiled to recover the source code, demonstrating that protected PHP code also exhibits potential security risks. To better analyze the security of PHP code at the bytecode level, our work proposes a comprehensive decompilation framework based on PHP bytecode.

C. Decompilation Process

Decompilation refers to the process of analyzing and transforming low-level code, such as binary code or assembly code [25], into equivalent high-level source code. This process encompasses a range of methodologies, including disassembly techniques [26], data type analysis [27], and control flow analysis [28], *etc*.

A complete decompilation system needs to include three key modules: the frontend, the universal decompilation engine, and the backend [29]. Due to the different instructions across platforms, the frontend of the decompiler is responsible for converting the binary code from different platforms into a unified intermediate representation (IR).

Intermediate representation (IR) is a data structure or code used by compilers to represent source code. It serves as an intermediate layer between high-level languages and target code during the compilation process [30] [31].

Based on this IR, a control flow graph is constructed. The next step is to reduce this control flow graph. The goal of reduction is to progressively transform the complex structure of the control flow graph into a single basic block, converting the complex graph structure into a linear structure that can be output as source code, while preserving the semantic information. Fig. 2 shows a simple example of such a reduction process.

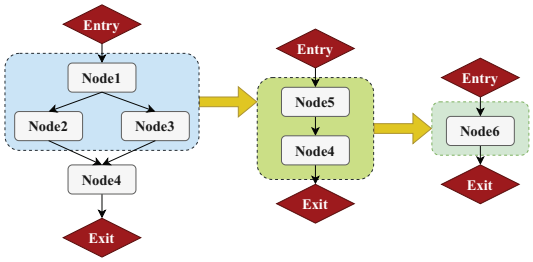

Fig. 2. Example of reduction

There are well-established decompilation algorithms and frameworks for C [12]–[15] and Java [16] [17]. However, there is currently a lack of a universal decompilation framework for PHP bytecode. Therefore, we aim to design a complete PHP decompilation framework to fill this gap in the research field.

D. SESS Analysis

The SESS (Single Entry Single Successor) analysis was first introduced by Engel et al. [32] as an improvement to traditional structural analysis. A region is considered a SESS region when it has only one entry point and one exit point. Through SESS analysis, loops in high-level languages can be better reduced, especially in cases where the loop contains control flow statements like `break` or `continue`. For example, Fig. 3 shows a control flow graph with a loop. After applying SESS analysis, the SESS region can be quickly identified and then reduced to a new node.

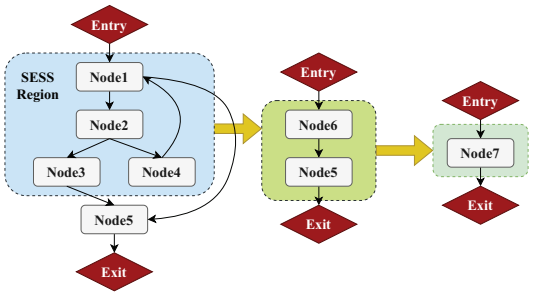

Fig. 3. Example of SESS analysis

SESS analysis addresses the limitations of previous structural analysis in decompilation, which could not handle control flows involving `return` and `break` statements. In designing the decompilation algorithm for PHP syntax, we also referred to the SESS analysis algorithm. We analyzed its limitations when applied to PHP's specific syntax and made several improvements based on it.

III. Motivation

In our team's analysis of closed-source PHP projects in real-world environments, we frequently encounter core PHP code that has been encrypted or protected. However, directly performing security analysis on encrypted PHP code is challenging, and the presence of encrypted code significantly impacts the effectiveness of static analysis on the entire system.

For example, we uncovered a vulnerability in the `remove` method of the `Certificate` class, as shown in Fig. 4(a). However, in the real-world system, `LDAP.php` is an encrypted file, due to the limitations of current source code-based static analysis methods, encrypted PHP code cannot be analyzed, leading to a disruption in the call chain of the remove method. By using the method proposed by Weißer et al. [6], we can obtain the PHP bytecode from encrypted files at runtime. However, there is currently no comprehensive method or related work specifically designed for decompiling PHP bytecode. Therefore, we conduct a systematic review and analysis of the PHP engine and existing decompilation algorithms, and design an entire source code recovery method for PHP.

IV. Proposed Framework: DEPHP

A. Overview

This section introduces the overall framework of DEPHP, a decompilation method based on PHP bytecode. The framework consists of three main parts, as shown in Fig. 5.

The first step is to elevate the original PHP bytecode, regardless of its version, into an intermediate representation. The second step generates a control flow graph from the intermediate representation and uses structural analysis algorithm in decompilation to reduce the control flow graph to a single node. The third step generates an abstract syntax tree from the reduced control flow graph. Finally, the source code is output from the abstract syntax tree to complete the decompilation process.

Due to the complexity of analyzing loop structures, we dedicate a separate chapter to this topic and introduce our improved SESS algorithm designed specifically for PHP's unique syntax. The following sections will provide a detailed analysis and explanation of these three main components.

B. Lifting PHP Bytecode Using Intermediate Representation

The first part of our DEPHP is the PHP bytecode lifting method based on an intermediate representation. First, we analyzed the commonalities and differences across PHP bytecode versions and designed an intermediate representation accordingly. Based on this representation, PHP bytecode is transformed into a unified intermediate format. Subsequently, these intermediate representations are segmented into basic blocks using related `jump` instructions and `try-catch` descriptors, constructing the control flow graph.

```php
<?php
class Certificate{
    // ...
    function __construct()
    {
        // ...
    }
    public function add()
    {
        // ...
    }
    public function remove($id, $cert_dir, $cacert_dir)
    {
        $pem_file = "$cert_dir/$id.pem";
        $cert_file = "$cert_dir/$id.crt";
        shell_exec("rm -rf $cacert_dir/
                `openssl x509 -noout -hash -in $pem_file`.0");
        shell_exec("rm -rf $pem_file $cert_file");
    }
}
```

(a) Certificate.php

```
...
8F7492F264381D49AAQAAAAXAAAABHAAAACABAAAAAAAAAD/
+mQOimSYB0/Hb9TXHUCW+PX75O1u91JP06EVW5mvJdXboi/
FeY/g/4ZP+zDbrYMwWin8xna28QjWNKL5uRHnea0HqRoGbf
dpEudesOLq5AjHCyQN+rvxSsa5YaU/RMFK++AbGiKxuYXLi
TzDQnU+jDUAAABIEgAAyGfWGcgeS+OjZq8uTDro6oqXAWNK
w3z83EHB5dlTLZ8XpQG0OPa/rkFFJ0LY8nmNG3zEAZ32hf0
BXgt9CqKM3gpXmzWsL8lvF/Tc+I4MYaTd651YRh9BA2cdTF
h3FQ1DTiETJt/
...
```

(b) LDAP.php

```php
<?php
class LDAP {
    // ...
    public function __construct() {
        // ...
    }
    public function add_certificate() {
        // ...
    }
    public function get_certificates() {
        // ...
    }
    public function remove_certificate() {
        $id = $this->input->post("id");
        $this->Certificate->remove($id,
                $this->_certs_dir, $this->_cacerts_dir);
        // ...
    }
}
```

(c) LDAP_decrypted.php

Fig. 4. Motivation Example

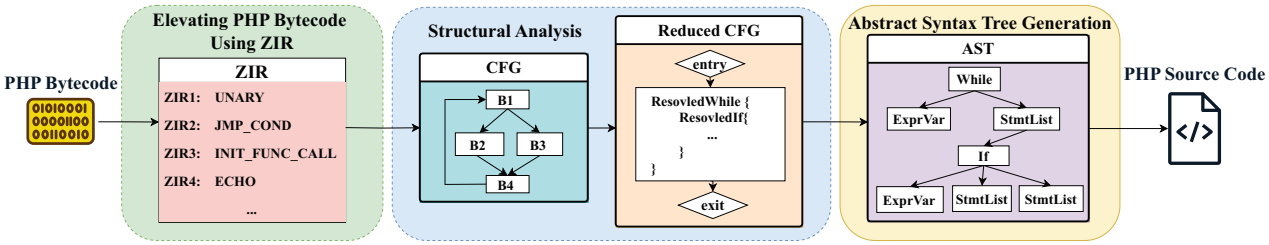

Fig. 5. DEPHP Framework (consists of three modules)

1) Design of Intermediate Representation for PHP: From PHP 7.0 to PHP 8.1, there have been a total of 226 different PHP bytecodes. Each version of PHP includes a subset of these 226 bytecodes. After analyzing the semantics of the bytecodes across different versions, we designs 65 unique intermediate representations[2]. For convenience, these representations are collectively referred to as ZIR, signifying an intermediate representation designed for the Zend Virtual Machine that runs PHP bytecode.

Among these bytecodes, many are unnecessary for decompilation purposes. For example, debugging bytecodes such as *USER_OPCODE* and *VERIFY_RETURN_TYPE*, as well as the inherently meaningless *NOP*, do not require corresponding intermediate representations in the design. For more common cases, many similar bytecodes can be merged. Examples include the *ASSIGN_** series for assignment-related calculations and the *SEND_** series for parameter passing.

We categorize our ZIR into nine major groups, as summarized in Table I. For categories with numerous corresponding bytecodes, only a subset is displayed.

The *ASSIGN* group represents common assignment operations. *UNARY* and *BINARY* include mathematical and bitwise operations, with all unary and binary operations consolidated into their respective groups.

CLASS, *FETCH*, and *FUNCTION* groups address PHP-specific features. The *CLASS* group handles dynamic class binding, allowing different class definitions at runtime within conditional blocks. *FETCH* supports PHP's variable index-

[2]https://sites.google.com/view/zirofdephp

TABLE I
CLASSIFICATION OF ZIR

Category	Included IR	Corresponding Bytecode
ASSIGN	ASSIGN ASSIGN_OP QM_ASSIGN	ASSIGN ASSIGN_OBJ_REF ASSIGN_REF
UNARY	UNARY CAST	PRE_INC BOOL_NOT
BINARY	BINARY	ADD SUB
CLASS	DECLARE_CLASS ADD_INTERFACE ADD_TRAIT	DECLARE_CLASS DECLARE_CLASS_DELAYED DECLARE_INHERITED_CLASS
FETCH	FETCH_LOCAL FETCH_GLOBAL FETCH_STATIC	FETCH FETCH_DIM FETCH_OBJ
FUNCTION	INIT_FUNC_CALL INIT_METHOD_CALL NEW	INIT_FCALL INIT_FCALL_BY_NAME INIT_NS_FCALL_BY_NAME
JUMP	JMP JMP_COND JMP_SET	JMP JMPNZ JMPZ
KEYWORD	ECHO RETURN CLONE	RETURN RETURN_BY_REF GENERATOR_RETURN
MISC	ROPE_INIT ADD_ROPE CATCH	ROPE_INIT ROPE_END ROPE_ADD

ing mechanism, where variables can be accessed by name. This also reflects the dynamic characteristics of the PHP language. *FUNCTION* manages global, class, static, and anonymous functions, accommodating PHP's dynamic function indexing. Bytecodes like *FCALL*, *FCALL_BY_NAME*, and *NS_FCALL_BY_NAME* are merged due to their semantic similarity.

The *JUMP* group encompasses conditional and unconditional jumps like *JMPNZ*, *JMPZ*, and *JMP*, representing branching logic in constructs like `if-else` and loops. *KEYWORD* group corresponds to PHP-specific keywords such as `clone`, `echo`, and `return`, each with dedicated bytecodes.

The MISC group includes remaining unclassified representations, such as exception handling, array initialization, and template strings, which do not fit the primary categories. To simplify the categorization, we have grouped them all under the MISC category without further subdivision.

2) Transforming ZIR to Control Flow Graph: After lifting the original PHP bytecode to ZIR, the next step in the decompilation process is to generate the control flow graph. In a control flow graph, a basic block is a linear segment of code with a single entry and exit point [33]. First, we divide these ZIRs into basic blocks based on their jump types and jump targets. After splitting the basic blocks, we connect them and determine the type of each node based on the jump type. The purpose of the categorization is to identify relevant regions during the subsequent structural analysis and reduction of the nodes. We categorize the nodes into seven major types, defined as follows:

- **BasicBlockNode**: This is the simplest node type, representing a block with a single successor. The last ZIR in the block is an unconditional `Jump` ZIR.
- **IfNode**: This node type represents a basic block where the last ZIR is a conditional jump. It has two successors corresponding to the two possible targets of the conditional jump.
- **ForEachNode**: This node is designed specifically for PHP's `ForEach` syntax. The last ZIR in the block is a `ForEach` ZIR.
- **MatchNode**: This node is designed for PHP's `Match` syntax. The last ZIR in the block is a `Match` ZIR, and each branch in the `Match` statement corresponds to one successor.
- **SpecialNode**: Represents the entry and exit points of the CFG.
- **TryCatchNode**: Designed for PHP's exception handling. This node is used to mark its successors as being part of an exception handling structure.
- **SESSRegionNode**: It represents an identified SESS region, with two successors: one pointing to the entry of the current loop and the other to the exit after the loop completes.

The *SESSRegionNode* is constructed dynamically during structural analysis and is not created during the current phase of transforming ZIR to the CFG. The specific mechanism for constructing the *SESSRegionNode* is discussed in detail in Section IV-D.

At this point, we have successfully unified PHP bytecode from different versions into a consistent intermediate representation and generated control flow graph based on it.

C. Structural Analysis Algorithm for PHP Language Features

The second part of our DEPHP describes how the nodes in the control flow graph are analyzed and reduced. After constructing the CFG, a structural analysis algorithm is needed to gradually reduce the complex graph into a single node. This section introduces the structural analysis algorithm tailored for the characteristics of the PHP language and its specific approach to analyzing control flow graphs.

First, we extend the ZIR designed in Section IV-B1 and explain how to ensure that no semantic information is lost during the reduction process. Then, we provide a detailed explanation of all the structural analysis methods for PHP syntax, excluding loop structures.

1) Replacing Reduced Nodes with Hybrid Intermediate Representations: When performing structural analysis to reduce multiple nodes, new nodes replace old ones, and it is essential to ensure that information from the reduced nodes is not lost in the process. In our DEPHP, reduced nodes are transformed into hybrid ZIR, which is incorporated as part of the basic block in the new node to retain the information from the reduced nodes. The main difference between Hybrid ZIR and regular ZIR is that it contains nested structures to store control flow information and semantic information.

Fig. 6 illustrates an example of reduction achieved through hybrid ZIR. The nodes within the identified region generate a new node, which transforms the original conditional jump statement and each of its branch statements into a single *ResolvedIfElse*. The *ResolvedIfElse* still belongs to a type of ZIR, but it preserves the control flow structure before the reduction through complex nested structures, ensuring that the original semantics are not lost.

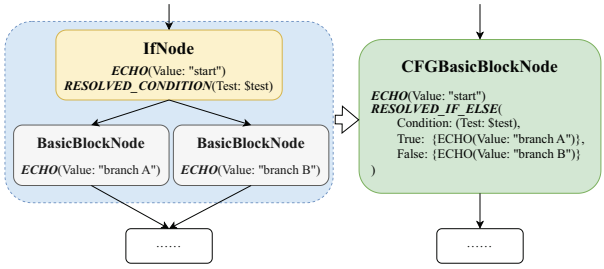

Fig. 6. Reduction through hybrid ZIR

The special hybrid ZIRs are shown in Table II. Among these, simple jump constructs like `break` and `continue`, do not nest other statements. All other representations contain one or more fields for storing basic blocks, representing the basic blocks within the reduced structures.

TABLE II
HYBRID ZIR

Name	Meaning
ResolvedIfElse	if-else syntax block
ResolvedCondition	composite condition syntax block
ResolvedMatch	match syntax block
ResolvedFor	for syntax block
ResolvedWhile	while syntax block
ResolvedDoWhile	do-while syntax block
ResolvedTryCatch	try-catch syntax block
ResolvedForEach	foreach syntax block
ResolvedGoto	goto statement
ResolvedBreak	break statement
ResolvedContinue	continue statement

2) Reduction of Consecutive Nodes: When a *BasicBlockNode* is directly followed by another *BasicBlockNode* or *IfNode*, the ZIR sequence of the latter node can be appended to the former. The subsequent node is then removed from the control flow graph, and the new node is connected to the successor of the removed node. This achieves the merging of consecutive nodes.

3) Reduction of Compound Conditions: When two *IfNode*s are connected sequentially and share a common successor, it represents a compound condition, as shown in Fig. 7. Depending on the direction of the true and false branches of the *IfNode*s, there are four possible cases: $a \,\&\&\, b$, $a \,\&\&\, !b$, $a \,||\, b$, $a \,||\, !b$. More complex compound conditions can also be transformed into a combination of these four cases.

4) Reduction of If-Else Structures: When structural analysis detects that both successor nodes of an *IfNode* are *BasicBlockNode*, each with exactly one predecessor and sharing the same successor, as shown in Fig. 7, the *IfNode* and its two successors are replaced with a single *BasicBlockNode*. The new node inherits the predecessors of the original *IfNode*, and its successor node is the common successor node of the two original *BasicBlockNode*s.

5) Reduction of Hash Jump Table Match Structures: When the `match` expression targets are constants (e.g., strings or numbers), PHP compiles it into a *Match* bytecode and uses a hash table for jump targets. In the control flow graph, this appears as a *MatchNode*, with each successor node corresponding to a match target. If all successors point to the same node, the *MatchNode* can be simplified into a *ResolvedMatch*, as shown in Fig. 7.

6) Reduction of Switch and Regular Match Structures: In a switch structure, interconnected *IfNode*s represent each `switch` expression, with each node's `False` branch linking to the next. Each *IfNode* performs a loose comparison between the matched variable and the current case expression, making it easy to identify and simplify switch regions. A match structure, a specialized form of switch, uses strict comparisons and ends each case with a `break`, allowing it to be distinguished and simplified.

7) Reduction Method for Try-Catch Structures: When the *TryCatchNode*'s *TrySuccessor*, *CatchSuccessors*, and *FinallySuccessor* all point to the *DoneSuccessor* via *BasicBlockNode*, it indicates that the current *TryCatchNode* can be reduced to a *ResolvedTryCatch*. For example, the try-catch control flow graph shown in Fig. 8 is already eligible for such reduction.

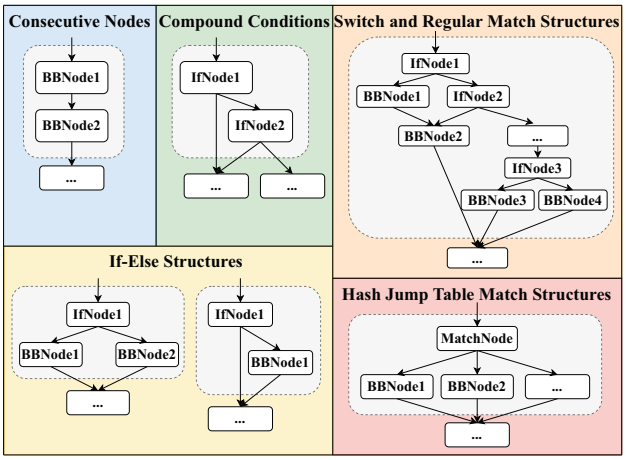

Fig. 7. Reduction method for regular structures

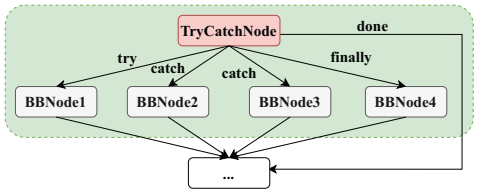

Fig. 8. Reduction method for try-catch

D. The Improved SESS Algorithm for PHP

1) Jumping out of nested loops in PHP: As mentioned earlier, PHP supports the use of `break` and `continue` statements to jump out of nested loops [34] [35]. We counted the number of `break` and `continue` statements used to jump out of nested loops in the dataset we selected. The details of the dataset selection and construction methods are described in Section V-A and the results are shown in Table III. It can be observed that in various major open-source PHP projects, there is a certain amount of syntax for jumping out of nested loops. This statistical finding also highlights the importance of addressing this special PHP syntax in decompilation. Otherwise, using `goto` statements to jump out of nested loops would significantly reduce code readability, which is not conducive to further code auditing and other types of analysis.

However, the traditional SESS algorithm for other languages cannot handle such special syntax in PHP. Specifically, as shown in the first control flow graph in Fig. 9, when identifying the inner loop region, the presence of two exit points prevents proper recognition of the inner loop as an SESS region. When analyzing the outer loop, although it forms an

TABLE III
STATEMENTS OF EXITING NESTED LOOPS IN LARGE-SCALE PHP PROJECT

Project	Version	Break 2	Break 3	Continue 2	Continue 3
Codeigniter	4.5.5	6	0	11	0
Drupal	11.0.9	17	0	53	3
Joomla	5.2.2	15	3	36	4
Laravel	10.47.0	26	3	35	1
Opencart	4.1	1	0	10	1
Symfony	6.4.15	6	0	31	2
Wordpress	6.7.1	16	2	14	2
Yii2	2.0.51	16	0	28	0

SESS region, the unresolved inner loop within it complicates the analysis, causing traditional SESS analysis methods to either introduce `goto` statements or make incorrect assumptions when dealing with nested loops [32].

To address these issues, we propose an improved method based on the original SESS algorithm. This method computes loop information in the control flow graph before analysis begins. Based on this loop information, *SESSRegionNode*s are dynamically generated to represent loops during analysis, and the entry and exit points of loops are transformed accordingly.

Fig. 9 shows the example where the `break 2` statement is used to exit two levels of loops and progressively simplify the control flow through flow reduction using the method we proposed. The method allows the reduction of exiting nested loops by first transforming the entry and exit edges of the outer loop. Then the inner loop is converted into an SESS region, and can be further reduced.

The following sections will introduce the detailed steps of each phase in the process.

2) Computing Loops in the Control Flow Graph: To compute loops in the control flow graph, the first step is to calculate the *Dominator* of each node. This determines the predecessor-successor relationships between nodes. Next, all nodes are traversed. If a node N has a successor node N_{succ}, and N_{succ} is a dominator of N, it indicates that N is part of a loop. The edge from N to N_{succ} is identified as a back edge of the loop.

The node N_{succ} is then marked as the head node of the loop. All nodes, including N and its direct and indirect predecessors, for which N_{succ} is a dominator, are identified as part of the loop body. These nodes are added to a set that represents the loop body. Using this method, all loops in the current control flow graph are identified, including each loop's head node and the corresponding set of body nodes.

3) Using SESSRegionNode to Transform Loop Edges: When performing structural analysis on the control flow graph after obtaining loop information, if the current node is marked as a loop head node, all loop body nodes within the current loop are traversed. If the node being traversed has edges that exit the loop and the target of those edges is not the exit node of the control flow graph, the target nodes of the outgoing edges are stored.

If all the edges exiting the loop point to the same target node, it indicates that the identified loop region is already an SESS region and can be reduced. If there are no edges exiting the loop, or if all outgoing edges point to the control flow graph's exit node, it indicates that the loop is an infinite loop. In this case, the exit node is treated as the loop region's exit.

Instead of following the original SESS structural analysis method of marking nodes that exit the loop as tail nodes and immediately reducing the loop, a new *SESSRegionNode* is dynamically generated. This node is inserted between the loop region's entry node and its predecessors. The entry and exit nodes of the loop region are designated as the new *SESSRegionNode*'s *Entry* and *Successor* nodes, respectively.

Additionally, the loop's incoming and outgoing edges are transformed. The successors of nodes that exit the loop are replaced with a new *BasicBlockNode* containing a *ResolvedGoto* ZIR. This *BasicBlockNode*'s successor is set to the exit node.

Each loop exit node is assigned a globally unique label. All new *ResolvedGoto* ZIRs for exiting the loop include a field that stores this label, indicating the corresponding exit node.

The same process is applied to all edges within the loop that point to the entry node. Its purpose is to recover the `continue` statement in the subsequent process. The detailed steps are described in Algorithm 1.

Algorithm 1 BuildSESSRegion Procedure

Input: loops identified in the control flow graph, entry node of the loop to be processed
Output: SESS region constructed with exit and break nodes

Procedure BUILDSESSREGION($allLoops$, $headNode$)
  $loop \leftarrow allLoops[headNode]$
  $ExitNode \leftarrow nil$
  $BreakNodes \leftarrow EmptyHashSet$
  if *loop* **then**
    for $Node$ in GETNODES(*loop*) **do**
      if CONTAINSEXITEDGE($Node$) **then**
        $currExitNode \leftarrow$ GETEXITNODE($Node$)
        $currBreakNode \leftarrow$ GETBREAKNODE($Node$)
        if $ExitNode = nil$ **or** $ExitNode = currExitNode$ **then**
          APPEND($BreakNodes$, $currBreakNode$)
          $ExitNode \leftarrow currExitNode$
        else
          return
        end if
      end if
    end for
    if $ExitNode = nil$ **then**
      $ExitNode \leftarrow$ GETCFGEXITNODE()
    end if
    $exitLabel \leftarrow$ ALLOCLABEL($ExitNode$)
    MAKEBREAKNODES($BreakNodes$, $exitLabel$)
    INSERTSESSREGIONNODE($headNode$)
    MAKECONTINUENODE($headNode$)
  end if

4) Reduction Method for SESSRegionNode: Next, we further reduce the control flow graph containing *SESSRegionNode*s. To reduce the *ResolvedGoto* nodes generated by `break` and `continue`, when one branch of an *IfNode* has a successor pointing to the *exit* node, it can be reduced to a *ResolvedIfElse*, which is contained within a *BasicBlockNode*, as shown in *BBNode4* and *BBNode5* in Fig. 9. When the *Entry* node pointed to by a *SESSRegionNode* is a *BasicBlockNode*,

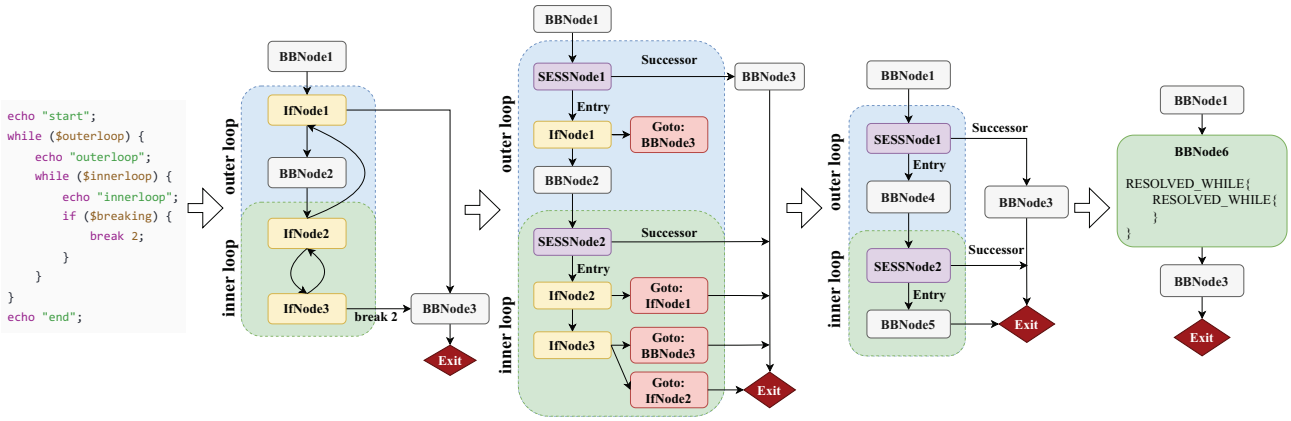

Fig. 9. Example of control flow graph reduction in jumping out of nested loops

and its successor is the *exit* node, it indicates that the current *SESSRegionNode* can be reduced, as demonstrated by *SESSNode2* and *BBNode5* in Fig. 9.

The next step is to recover `break` and `continue` statements from *ResolvedGoto*. During the process, all ZIRs within the loop body are recursively traversed. If the current ZIR is *ResolvedGoto* and its target label matches the *Entry* label of the *SESSRegionNode*, it indicates that the *ResolvedGoto* was dynamically generated by a `continue` statement. The ZIR is replaced by *ResolvedContinue*, and the loop depth of `continue` is determined based on the nesting level of the current basic block. For `break` statements, a similar process is followed.

After replacing all *ResolvedGoto* within the loop, the next step is to determine the type of the current loop and select the corresponding hybrid ZIR for reduction. The type of loop is inferred based on the characteristics of the current loop. If no specific characteristics are identified, the default type is set to `while(true)` or `for(;;)` to represent an infinite loop.

In our improved SESS analysis method, when an inner loop is reduced to a *ResolvedWhile*, it still uses *ResolvedGoto* to maintain the jump relationships. Only when the outer loop is reduced, the depth of the jump ZIRs is resolved to restore `break 2`, thus effectively handling nested loop exits without losing any information.

E. Abstract Syntax Tree Generation

The third part of our DEPHP is the abstract syntax tree generation algorithm, which illustrates how the reduced control flow graph is used to generate the corresponding AST.

After completing the control flow graph analysis, the CFG will contain only a single *BasicBlockNode* apart from the special entry and exit nodes. All control flow information and program semantic information is stored within this *BasicBlockNode* in the form of hybrid intermediate representation. The hybrid intermediate representation within the final *BasicBlockNode* serves as a container for other basic blocks. The next step is to recursively process these basic blocks and restore them to their corresponding abstract syntax tree based on their types.

During this step of decompilation, DEPHP leverages the open-source abstract syntax tree generation algorithm [36], eliminating the need to rebuild an abstract syntax tree library from scratch. By leveraging this well-established algorithm, DEPHP ensures both accuracy and performance in generating a reliable abstract syntax tree, which is crucial for producing high-quality decompiled source code. Finally, the constructed abstract syntax tree is used to directly generate the corresponding source code, thus completing the full decompilation process.

V. Evaluation

In this section, we design a systematic experiment to comprehensively evaluate DEPHP. *First, we assess the correctness and readability of the code decompiled by DEPHP. Then, we evaluate whether the improved structural analysis algorithm in DEPHP can restore special PHP syntax. Finally, we verify whether DEPHP could recover the original vulnerability patterns from encrypted PHP code.*

A. Evaluation Methodology

1) Datasets: To evaluate the proposed PHP bytecode-based decompilation algorithm, we select the top four open-source PHP CMSs based on market share statistics from W3Techs [37] and the top four open-source PHP frameworks with the highest usage rates according to the statistics of JetBrains [38], the selected datasets are shown in Table IV. We first compile the original code from these projects into PHP bytecode using the PHP interpreter. Then, we save the generated bytecode and decompile it using our DEPHP tool, producing output code and comparing against the original source code.

2) Experimental Setup: To simplify the experimental process and improve efficiency, we convert each project into locally stored OPCache files during the compilation process. OPCache [39] is a commonly used optimization caching mechanism in PHP that stores the bytecode of PHP scripts as static files on the server's memory or disk. This avoids the

TABLE IV
DATASETS

Category	Project	Version	PHP Files	Lines of Code
CMS	WordPress	6.7.1	1094	217356
	Drupal	11.0.9	11981	1032472
	Joomla	5.2.2	6091	557796
	OpenCart	4.1	5565	178554
Framework	Laravel	10.47.0	7314	574908
	Symfony	6.4.15	1464	124925
	CodeIgniter	4.5.5	2742	280458
	Yii2	2.0.51	4252	430809

need to recompile the script each time it is executed. Starting from PHP version 7, OPCache has been integrated as a core extension of PHP, allowing us to easily utilize the OPCache modules of different PHP versions for bytecode compilation. Using OPCache as the initial input for our DEPHP and then decompiling it, we evaluate the accuracy and the ability to restore the original structure of our decompilation algorithm by comparing the recovered code with the original code.

In addition to the static method for obtaining bytecode, we can also dynamically retrieve bytecode during PHP runtime, as shown in Fig. 10. Whether the PHP code is protected or not, it will call the underlying PHP engine during execution, specifically a series of core functions in the Zend Engine, such as `zend_execute` [40]. By writing a PHP extension [41] to hook these key functions, we can access the bytecode. For protected PHP code, as long as the protection mechanism does not modify the execution rules of PHP bytecode execution in the Zend Engine, we can get the normal bytecode and use DEPHP to recover the source code. We encrypt a series of PHP codes with vulnerability patterns, and then use this method to extract the bytecode for decompilation, observing whether the original vulnerability patterns are restored, so as to prove that DEPHP can assist in security analysis of encrypted PHP code.

3) Comparison with Zend-Decoder: Currently, there is limited work on decompilation for the PHP language. We compare DEPHP with the only existing open-source PHP decompilation tool, Zend-Decoder [42], which is a reproduction of the work by Weißer et al. [6]. The decompilation approach of Zend-Decoder is based on pattern matching of PHP bytecode, fundamentally differing from the structural analysis performed after intermediate representation transformation in our work. Moreover, Zend-Decoder only supports older versions of PHP and lacks continuous maintenance, which limits its applicability. Specifically, Zend-Decoder only supports PHP 5.6, according to the usage statistics of PHP in W3Techs [1], the usage rate of PHP 5 has dropped to less than 12%, additionally, PHP 5 reached its end of life in 2018 [43]. The current dominant PHP versions are PHP 7 and PHP 8, with a combined usage rate exceeding 88%. This highlights the urgent need for decompilation algorithms targeting higher versions of PHP to fill the gap in this research area. This situation also reflects that research on decompilation for PHP is still in a nascent stage, highlighting the need for a more universal PHP decompilation framework and methodology.

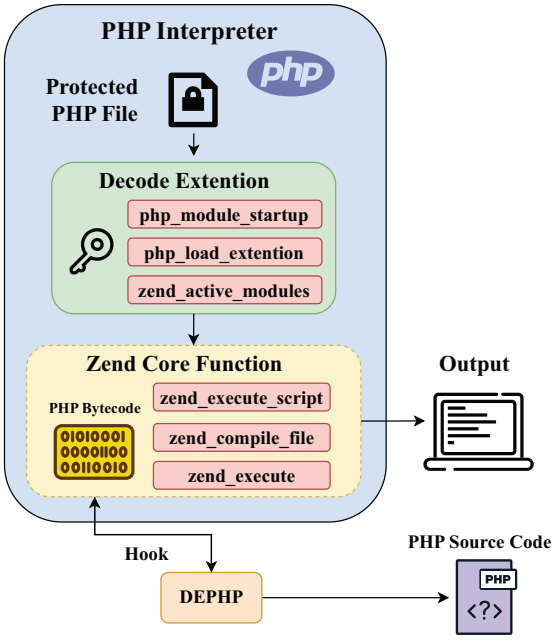

Fig. 10. Hooking key functions to retrieve PHP bytecode

B. Evaluation Metrics

Similar to the evaluation methods in the decompilation research of C [12]–[15] and Java [16] [17], we analyze the proposed decompilation algorithm from two dimensions to comprehensively evaluate our DEPHP. We use PhpMetric [44] to assist us in calculating various evaluation metrics.

1) Correctness: Correctness assesses the accuracy of the decompilation algorithm, measuring whether the decompiled code can faithfully restore the original program's functionality and structure.

- **Logical Lines of Code (LLOC)**: Logical lines of code refer to the number of meaningful code lines, excluding empty lines, comments, and other non-functional lines. In reverse engineering works targeting other languages, lines of code (LOC) is often used as a critical metric for evaluating decompilation quality, as more compact code generally indicates better structure and readability [12]. In our work on decompiling PHP, using standard line counts for comparison lacks accuracy, as spaces and comments do not contribute to the core logic.
- **Classes**: The total number of defined classes in the project. The number of classes reflects how well the decompiler preserves the object-oriented design. Significant changes in the number of classes may indicate that the decompiler either lost some classes or incorrectly merged them.
- **Methods**: The total number of defined methods in the project. Changes in the number of methods may indicate

TABLE V
STATISTICS OF DECOMPILATION RESULTS FOR EACH PROJECT

Dataset	LLOC			Classes			Methods			AAC			AEC			DIT			GOTOs			CC		
	Orig	**DP**	ZD	Orig	**DP**	ZD	Orig	**DP**	ZD	Orig	**DP**	ZD	Orig	**DP**	ZD	Orig	**DP**	ZD	Orig	**DP**	ZD	Orig	**DP**	ZD
WordPress	127268	**119810**	39997	555	**554**	275	5577	**5526**	2103	1.82	**1.73**	0.82	2.13	**2.08**	1.66	1.27	**1.27**	1.34	0	**0**	0	47.6	**46.36**	28.67
Drupal	600563	**515846**	3190	10944	**10081**	25	44238	**38441**	184	2.37	**1.93**	0.68	3.54	**2.59**	1.76	1.22	**1.26**	1.31	19	**0**	0	6.66	**5.56**	26.56
Joomla	321864	**269026**	7345	4688	**4166**	112	21018	**17174**	426	2.94	**2.29**	1.04	4.06	**3.3**	2.18	1.19	**1.22**	1.59	5	**0**	2	12.82	**11.74**	9.2
OpenCart	108036	**98719**	2262	1913	**1891**	36	6033	**5402**	208	1.52	**1.28**	0.25	1.97	**1.58**	2.69	1.16	**1.16**	1.07	9	**0**	2	11.33	**9.97**	10.08
Laravel	372846	**220829**	2184	5494	**4552**	73	29182	**21715**	288	2.18	**1.37**	0.52	3.35	**2.45**	2.34	1.16	**1.18**	1.46	6	**0**	9	8.75	**7.39**	3.25
Symfony	98195	**64567**	86	1231	**1001**	13	7277	**5345**	3	2.38	**1.13**	0.54	3.9	**2.47**	1.08	1.23	**1.24**	2	19	**0**	0	17.84	**12.46**	1.31
CodeIgniter	194669	**105368**	300	2347	**2272**	14	10096	**9041**	25	2.16	**1.5**	0	2.93	**2.23**	1.29	1.19	**1.2**	1.5	0	**0**	0	7.61	**6.75**	3
Yii2	290014	**184861**	8343	3673	**3492**	214	17445	**15259**	540	2.33	**1.68**	1.3	3.15	**2.43**	1.82	1.21	**1.22**	1.45	6	**0**	2	9.83	**8.72**	7.29

[a]**Orig**: Original, [b]**DP**: DEPHP, [c]**ZD**: Zend-Decoder

missing or incorrectly merged functionalities during decompilation.

- **Average Afferent Coupling (AAC)**: This measures the number of other classes that depend on a given class. If the decompiled code significantly alters the dependency relationships, it may indicate that the functionality of certain modules has been changed or lost. A large increase in AAC may suggest unnecessary coupling introduced by the decompiler.
- **Average Efferent Coupling (AEC)**: This measures the number of other classes that a given class depends on. If the AEC of the decompiled code is significantly higher than the original, it could suggest an increase in external dependencies introduced by the decompiler.
- **Depth of Inheritance Tree (DIT)**: This represents the depth of the inheritance tree, reflecting the complexity and hierarchy of class inheritance. If the decompiled code alters the inheritance hierarchy, it may indicate that the decompiler failed to accurately preserve the original code's structure.

2) Structuredness: Structuredness evaluates the structural quality of the decompiled output code. It can be assessed using the two following metrics. These metrics reflect whether the recovered code maintains good readability and whether it contains excessive redundancy.

- **Goto**: The number of `goto` statements in the decompiled code reflects the algorithm's ability to accurately recover the original control flow [12]–[15]. The presence of additional `goto` statements serves as an indicator of whether a decompilation system can fully restore the control flow of the source code. If the control flow cannot be completely recovered, the generated code will include `goto` statements to ensure semantic correctness. However, such inclusion can negatively affect the readability of the code.
- **Cyclomatic Complexity (CC)**: Cyclomatic complexity [45] measures the logical complexity of the code by counting the number of independent paths through the code, such as those introduced by decision points (e.g., `if`, `while`, `for`). Changes in cyclomatic complexity can indicate how well the logical structure of the code is preserved. A significant increase in complexity may indicate redundant or unnecessary logic introduced by the decompiler, while a decrease could suggest missing logical components.

C. Evaluation of Decompilation Algorithm

1) Evaluation Results of Correctness and Structuredness: The experimental results are shown in Table V. For each project in the dataset, we compute the metrics mentioned earlier. It can be observed that our DEPHP effectively reconstructs the classes and methods from the original project. The inheritance relationships between the reconstructed classes are also similar to those in the original project, while no redundant `goto` statements are introduced, ensuring a well-structured decompiled code. Since different projects may have varying coding styles, the number of logical lines of code in the recovered code may appear different from the original, but this does not indicate that the decompilation algorithm is ineffective. For comparison and presentation purposes, we calculate the percentages of average deviation of each metric in the dataset, as shown in Table VI.

TABLE VI
AVERAGE DEVIATION FOR EACH METRIC ACROSS ALL PROJECTS

Metrics	Average Deviation	
	DEPHP	**Zend-Decoder**
LLOC	25.26%	94.99%
Classes	8.03%	92.14%
Methods	14.73%	94.00%
AAC	26.19%	71.51%
AEC	22.24%	44.48%
DIT	1.25%	23.57%
CC	13.48%	77.47%

The results provide unambiguous evidence that Zend-Decoder is essentially incapable of correctly restoring code, as the number of classes and functions recovered in each project is very small. One reason for this is that the tool no longer supports modern PHP versions, only supporting syntax up to PHP 5.6, making it impossible to recognize the bytecode for decompilation. Another reason is that Zend-Decoder's decompilation approach, which relies on PHP bytecode pattern matching, is no longer sufficient to handle complex control

flows in real-world projects. This further highlights the robustness of our structural analysis algorithm.

In the structural analysis of the decompiled results, it can be observed that the use of `goto` statements is very rare in various PHP projects. This is because PHP is widely used in web development, where code tends to focus on clear flow and layered structures. The use of `goto` could make the program's control flow harder to trace, which contradicts the needs of web application development. Through the structural analysis algorithm of our DEPHP, no extraneous `goto` statements appeared in any of the decompiled projects, further demonstrating that the algorithm covers the full range of PHP control flow patterns.

2) Recovery of jumping out of nested loops: To verify whether our improved SESS algorithm can address the issue of jumping out of nested loops in the decompilation, we counted the number of `break` statements for nested loops before and after decompilation, as shown in Table VII. After decompilation, the number of such statements was nearly restored to the original count. It can be noted that the number of statements restored by DEPHP may be slightly more or slightly less than the original, as during the decompilation process, regardless of whether the control flow comes from a statement like `break` for nested loops, as long as a SESS region is detected, the improved SESS algorithm will trigger and decompile it into statements like `break 2`. However, the final semantics remain consistent. Moreover, in subsequent experiments in Section V-D, we proved that these minor differences in code recovery do not affect the vulnerability patterns, control flow, data flow, and other information in the output code. Therefore, we demonstrate that our improved SESS algorithm can successfully address the special syntax for jumping out of nested loops in PHP.

TABLE VII
BREAK STATEMENTS COMPARISON

Project	Break statements of nested loops		
	Original	**DEPHP**	**Zend-Decoder**
Codeigniter	6	6	0
Drupal	17	30	2
Joomla	18	21	1
Laravel	29	11	0
Opencart	1	10	0
Symfony	6	22	0
Wordpress	18	21	5
Yii2	16	10	2

D. Recovering Source Code and Vulnerability Patterns from Protected PHP Code

First, we present the code recovery example for the two protection mechanisms described in Section II-B. It can be observed that our DEPHP performs well in recovering code protected by extended encryption methods, such as shown in Fig. 11(a). The recovery effect is generally weaker for non-extended obfuscation methods as shown in Fig. 11(b). This is because recovering source code for patterns such as the replacement of variable names, function names, and class names during obfuscation goes beyond the scope of our discussion. However, for the control flow obfuscation methods, our structural analysis algorithm works well, ultimately outputting clear and structured source code.

```php
<?php
$hello = "hello ";$world = "world ";
$what = "what ";$a = "a ";
$beautiful = "beautiful ";
$day = "day ";$i = 0;
while ($i <= 7) {
    print $hello;print $world;
    print PHP_EOL;print $what;
    print $a;print $beautiful;
    print $day;print PHP_EOL;
    ++$i;
}
print "that's it!" . PHP_EOL;
```

(a) Recovery of encrypted code

```php
<?php
$PJQV2 = "hello ";$r9B_C = "world ";
$hRaR7 = "what ";$p0deY = "a ";
$a253K = "beautiful ";
$IT8tn = "day ";$bafbm = 0;
while ($bafbm <= 7) {
    print $PJQV2;print $r9B_C;
    print "\n";print $hRaR7;
    print $p0deY;print $a253K;
    print $IT8tn;print "\n";
    ++$bafbm;
}
print "that's it!\n";
```

(b) Recovery of obfuscated code

Fig. 11. Example of PHP recovered code

1) Recovering SQL injection and XSS patterns: To demonstrate that our method can assist in vulnerability discovery in protected PHP code, we analyze the ability of DEPHP to recover vulnerability patterns for SQL injection and Cross-Site Scripting (XSS) vulnerabilities. SQL injection and XSS are two significant vulnerabilities in web applications [46]. Schuckert et al. [47]–[50] specifically studied the patterns of these vulnerabilities in real-world projects that contain them, and further analyzed how certain code patterns impact static analysis. Based on their research, they constructed a dataset of these code patterns [51], including 37 SQL injection vulnerability patterns and 31 XSS vulnerability patterns.

We designed the following experiment based on this dataset. Initially, we encrypted each pattern in the dataset into unreadable PHP bytecode, simulating the situation where some PHP files in closed-source projects are encrypted. In this scenario, static code analysis tools cannot detect the original vulnerabilities. Then, we used DEPHP to restore the encrypted project and employed PHP static analysis tools to determine whether the decompiled code revealed the original vulnerability patterns.

We performed a comparative analysis of the original code and the decompiled code using the static analysis tool Joern [52], which is based on the Code Property Graph (CPG) [53] [54]. The nodes in the code property graph include function nodes, class nodes, function call nodes, and other syntax elements. The specific definitions of CPG nodes are given in [55]. We conducted a comparison of the number of AST, CFG, Method, and Call nodes between decompiled and original code to analyze the accuracy of reconstructed syntax, control flow, and overall code architecture, as shown in Table VIII.

Finally, we used the vulnerability scanning rules provided by Joern to perform vulnerability discovery on the code before and after decompilation. Results are shown in Table IX, for some code patterns in the dataset, both pre-decompilation and post-decompilation scans produced false positives or false negatives. This is due to the simplicity of the vulnerability detection rules for PHP in Joern. However, in this paper,

TABLE VIII
DIFFERENCES IN CPG NODE COUNTS BEFORE AND AFTER DECOMPILATION

Metrics	Average Deviation	
	SQLi patterns	**XSS patterns**
Total nodes	3.84%	2.98%
AST nodes	4.07%	2.38%
CFG nodes	4.44%	2.11%
Method nodes	4.88%	1.47%
Call nodes	4.39%	0.51%

our focus is not on optimizing the vulnerability detection rules or methods, but rather on proving that DEPHP can successfully restore the original vulnerability code patterns, even if false positives or false negatives occur. From the experimental results, we observed that the vulnerability paths detected before and after decompilation were generally the same, demonstrating the ability of DEPHP to recover encrypted PHP files. Reducing the ratio of false positives and false negatives requires deeper research into static analysis theories, which is beyond the scope of this paper. Detailed experimental data are presented in Table X in Appendix.

TABLE IX
RESULTS FOR VULNERABILITY DETECTION BEFORE AND AFTER DECOMPILATION

Datasets	**TP**	**TN**	**FP**	**FN**
SQLi Patterns^{+}	16	1	6	14
SQLi Patterns*	16	1	6	14
XSS Patterns^{+}	14	0	5	12
XSS Patterns*	14	0	5	12

$^{+}$: Vulnerability detection results in the original pattern,
*: Vulnerability detection results in the decompiled pattern

2) Vulnerability discovery in real-world scenarios: Using the bytecode extraction and source code recovery methods described above, we analyzed protected commercial projects in a real-world environment and successfully recovered a series of PHP source codes. By auditing the recovered source code and taking use of static analysis tools, we discovered several relevant vulnerabilities[3]. The vulnerabilities were reported to vendors and confirmed, with patches and mitigations deployed. Thus, we have demonstrated that our decompilation approach can effectively support comprehensive security analysis of PHP projects with encrypted code, serving as a valuable supplement and enhancement to static analysis and other source code analysis methods.

VI. RELATED WORK

A. Intermediate Representation Techniques in Decompilation

One of the most widely used linear IRs is LLVM IR, introduced in 2004 by Lattner et al. [56]. Initially developed for the Clang compiler [57], LLVM IR translates C/C++ into machine code for various CPU architectures, and is commonly used in decompiling native machine code.

In 2017, Federico et al. [58] [59] integrated LLVM IR with QEMU [60] to translate binary programs into LLVM IR, enabling analysis across 17 architectures. Similarly, Kirchner et al. [61] used the Dagger decompiler to convert binaries into LLVM IR for program analysis.

Beyond LLVM, Brumley et al. introduced the BAP IR in 2011 [62], which was later used in the Phoenix decompiler [12]. Additionally, IRs for higher-level languages, such as Grimp for the JVM [63], have also been used in decompilation. Miecznikowski et al. [64] utilized Grimp to create Dava, a Java bytecode decompiler.

There is currently no intermediate representation specifically designed for the decompilation of PHP bytecode. Moreover, due to the fundamental differences between the PHP interpreter and compiled languages, it is not feasible to directly apply intermediate representation mechanisms used in compiled languages such as C or Java to PHP bytecode. Our work addresses this gap in the field.

B. Control Flow Analysis Techniques in Decompilation

Early decompilers relied on direct pattern matching to map control flows from target code to high-level languages [65]. However, this approach was ineffective in the presence of compiler optimizations and complex control flows.

To address these challenges, Housel et al. [66] proposed interval analysis for control flow analysis, which evolved into the structured analysis approach by Cifuentes et al. [67]. This method used linear IR to generate control flow graphs (CFGs) and perform structural analysis, becoming a standard for decompiling languages like C/C++ and Java.

Engel et al. [32] improved this by analyzing single-entry single-successor (SESS) regions in CFGs, addressing limitations in handling break statements within loops. Brumley et al. [12] refined the SESS approach, while Yakdan et al. [13] converted irreducible graphs into reducible ones, and Gussoni et al. [14] improved graph readability.

Burk et al. [68] developed the Decomperson tool, using data from reverse engineering competitions, and Basque et al. [15] introduced de-optimization algorithms to tackle compiler optimizations.

In Java, Miecznikowski et al. [64] extended the Soot framework with Grimp to decompile Java bytecode. Nanda et al. [16] improved boolean expression decompilation, while Harrand et al. [17] created Arlecchino, a meta-decompiler combining outputs from multiple decompilers.

Industrial decompilers for C include IDA Pro [69], Binary Ninja [70], Ghidra [71], and RetDec [72]. Java decompilers like Java Decompiler [73], Jadx [74], and Fernflower [75] directly decompile bytecode due to Java's backward compatibility [76]. However, due to the unique characteristics of PHP, these methods are not easily applicable to PHP. Therefore, we have developed a systematic decompilation approach tailored to PHP bytecode.

[3]https://sites.google.com/view/vulnerabilityfoundbydephp

VII. Conclusion

In this paper, we present a PHP bytecode-based source code recovery method comprising three key components: (1) a unified intermediate representation framework that normalizes bytecode across PHP versions; (2) a structural analysis algorithm, extending SESS, for optimized control flow graph reduction tailored to PHP's syntax; and (3) an abstract syntax tree generation process that reconstructs the original source directly from the reduced graph. We evaluate our approach on over three million lines of PHP code, achieving 92% class recovery and 85% method recovery, fully restoring SQL-injection and XSS vulnerability patterns, and uncovering real-world flaws (6 CVEs assigned). These results demonstrate the accuracy, structural fidelity, and effectiveness of our method in the static security analysis of encrypted PHP code, thereby advancing the capabilities of static code analysis.

Acknowledgement

This project is supported by National Natural Science Foundation of China (Grant No.92467201) and the Youth Innovation Promotion Association CAS (Grant No.Y2023047).

References

[1] "Usage statistics of php for websites," accessed: 2025-04-16. [Online]. Available: https://w3techs.com/technologies/details/pl-php

[2] N. C. Brown, P. Weill-Tessier, M. Sekula, A.-L. Costache, and M. Kölling, "Novice use of the java programming language," *ACM Transactions on Computing Education*, vol. 23, no. 1, pp. 1–24, 2022.

[3] A. Ranjan, R. Kumar, and J. Dhar, "A comparative study between dynamic web scripting languages," in *Data Engineering and Management: Second International Conference, ICDEM 2010, Tiruchirappalli, India, July 29-31, 2010. Revised Selected Papers*. Springer, 2012, pp. 288–295.

[4] M. Kazerounian, B. M. Ren, and J. S. Foster, "Sound, heuristic type annotation inference for ruby," in *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, 2020, pp. 112–125.

[5] A. E. Kwame, E. M. Martey, and A. G. Chris, "Qualitative assessment of compiled, interpreted and hybrid programming languages," *Communications on Applied Electronics*, vol. 7, no. 7, pp. 8–13, 2017.

[6] D. Weißer, J. Dahse, and T. Holz, "Security analysis of php bytecode protection mechanisms," in *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015. Proceedings 18*. Springer, 2015, pp. 493–514.

[7] M. Hills, "Evolution of dynamic feature usage in php," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 525–529.

[8] M. Hills, P. Klint, and J. Vinju, "An empirical study of php feature usage: a static analysis perspective," in *Proceedings of the 2013 international symposium on software testing and analysis*, 2013, pp. 325–335.

[9] "Migrating from php 8.0.x to php 8.1.x," accessed: 2025-04-16. [Online]. Available: https://www.php.net/manual/en/migration81.incompatible.php

[10] "Migrating from php 7.4.x to php 8.0.x," accessed: 2025-04-16. [Online]. Available: https://www.php.net/manual/en/migration81.incompatible.php

[11] "Migrating from php 7.3.x to php 7.4.x," accessed: 2025-04-16. [Online]. Available: https://www.php.net/manual/en/migration81.incompatible.php

[12] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo, "Native x86 decompilation using {Semantics-Preserving} structural analysis and iterative {Control-Flow} structuring," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 353–368.

[13] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations." in *NDSS*. Citeseer, 2015.

[14] A. Gussoni, A. Di Federico, P. Fezzardi, and G. Agosta, "A comb for decompiled c code," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020, pp. 637–651.

[15] Z. L. Basque, A. P. Bajaj, W. Gibbs, J. O'Kain, D. Miao, T. Bao, A. Doupé, Y. Shoshitaishvili, and R. Wang, "Ahoy sailr! there is no need to dream of c: A compiler-aware structuring algorithm for binary decompilation," in *Proceedings of the USENIX Security Symposium*, 2024.

[16] M. G. Nanda and S. Arun-Kumar, "Decompiling boolean expressions from java™ bytecode," in *Proceedings of the 9th India Software Engineering Conference*, 2016, pp. 59–69.

[17] N. Harrand, C. Soto-Valero, M. Monperrus, and B. Baudry, "Java decompiler diversity and its application to meta-decompilation," *Journal of Systems and Software*, vol. 168, p. 110645, 2020.

[18] "Php: Hypertext preprocessor," accessed: 2025-04-16. [Online]. Available: https://www.php.net/

[19] "php7-internal," accessed: 2025-04-16. [Online]. Available: https://github.com/pangudashu/php7-internal

[20] "Zend engine," accessed: 2025-04-16. [Online]. Available: https://www.phpinternalsbook.com/php7/zend_engine.html

[21] "Zend engine 2 opcodes," accessed: 2025-04-16. [Online]. Available: http://php.adamharvey.name/manual/en/internals2.opcodes.php

[22] I. Khairunisa and H. Kabetta, "Php source code protection using layout obfuscation and aes-256 encryption algorithm," in *2021 6th International Workshop on Big Data and Information Security (IWBIS)*. IEEE, 2021, pp. 133–138.

[23] "Yak pro - php obfuscator," accessed: 2025-04-16. [Online]. Available: https://github.com/pk-fr/yakpro-po

[24] "Sourceguardian," accessed: 2025-04-16. [Online]. Available: https://www.sourceguardian.com/

[25] G. Morrisett, K. Crary, N. Glew, and D. Walker, "Stack-based typed assembly language," in *International Workshop on Types in Compilation*. Springer, 1998, pp. 28–52.

[26] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 627–642.

[27] K. Troshina, Y. Derevenets, and A. Chernov, "Reconstruction of composite types for decompilation," in *2010 10th IEEE Working conference on source code analysis and manipulation*. IEEE, 2010, pp. 179–188.

[28] F. E. Allen, "Control flow analysis," *ACM Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[29] C. Cifuentes and K. J. Gough, "A methodology for decompilation," in *Proceedings of the XIX Conferencia Latinoamericana de Inform atica*, 1993, pp. 257–266.

[30] A. W. Appel, "Modern compiler implementation in c: Basic techniques." *Computers and Mathematics with Applications*, vol. 7, no. 33, p. 139, 1997.

[31] D. Chisnall, "The challenge of cross-language interoperability," *Communications of the ACM*, vol. 56, no. 12, pp. 50–56, 2013.

[32] F. Engel, R. Leupers, G. Ascheid, M. Ferger, and M. Beemster, "Enhanced structural analysis for c code reconstruction from ir code," in *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems*, 2011, pp. 21–27.

[33] W. Xie and F. Ting, "Design and implementation of the virtual machine constructing on register," in *2008 International Conference on Computer Science and Software Engineering*, vol. 2. IEEE, 2008, pp. 424–430.

[34] "Break control structure," accessed: 2025-04-16. [Online]. Available: https://www.php.net/manual/en/control-structures.break.php

[35] "Continue control structure," accessed: 2025-04-16. [Online]. Available: https://www.php.net/manual/en/control-structures.continue.php

[36] "php-parser," accessed: 2025-04-16. [Online]. Available: https://github.com/VKCOM/php-parser

[37] "Usage statistics and market shares of content management systems," accessed: 2025-04-16. [Online]. Available: https://w3techs.com/technologies/overview/content_management

[38] "Php developer ecosystem 2023," accessed: 2025-04-16. [Online]. Available: https://www.jetbrains.com/lp/devecosystem-2023/php/

[39] "Opcache," accessed: 2025-04-16. [Online]. Available: https://www.php.net/manual/en/book.opcache.php

[40] "Php source code repository," accessed: 2025-04-16. [Online]. Available: https://github.com/php/php-src

[41] "Writing php extensions," accessed: 2025-04-16. [Online]. Available: https://www.zend.com/resources/writing-php-extensions

[42] "Zend-decoder," accessed: 2025-04-16. [Online]. Available: https://github.com/Tools2/Zend-Decoder

[43] "Php end of life (eol)," accessed: 2025-04-16. [Online]. Available: https://www.php.net/eol.php

[44] "Phpmetrics," accessed: 2025-04-16. [Online]. Available: https://github.com/phpmetrics/PhpMetrics
[45] C. Ebert, J. Cain, G. Antoniol, S. Counsell, and P. Laplante, "Cyclomatic complexity," *IEEE software*, vol. 33, no. 6, pp. 27–29, 2016.
[46] Mitre, "Cwe top 25 most dangerous software weaknesses - 2024." [Online]. Available: https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html
[47] F. Schuckert, M. Hildner, B. Katt, and H. Langweg, "Source code patterns of cross site scripting in php open source projects," 2018.
[48] F. Schuckert, B. Katt, and H. Langweg, "Source code patterns of sql injection vulnerabilities," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, 2017, pp. 1–7.
[49] ——, "Difficult xss code patterns for static code analysis tools," in *Computer Security: ESORICS 2019 International Workshops, IOSec, MSTEC, and FINSEC, Luxembourg City, Luxembourg, September 26–27, 2019, Revised Selected Papers 2*. Springer, 2020, pp. 123–139.
[50] ——, "Diffcult sqli code patterns for static code analysis tools," in *Norsk IKT-konferanse for forskning og utdanning*, no. 3, 2020.
[51] F. Schuckert, "Sca patterns." [Online]. Available: https://github.com/fschuckert/sca_patterns
[52] Joernio, "Joern: A static analysis tool for vulnerability discovery." [Online]. Available: https://github.com/joernio/joern
[53] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 590–604.
[54] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and flexible discovery of php application vulnerabilities," in *2017 IEEE european symposium on security and privacy (EuroS&P)*. IEEE, 2017, pp. 334–349.
[55] "Joern," accessed: 2025-04-16. [Online]. Available: https://cpg.joern.io/
[56] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.
[57] C. Lattner, "Llvm and clang: Next generation compiler technology," in *The BSD conference*, vol. 5, 2008, pp. 1–20.
[58] A. Di Federico, M. Payer, and G. Agosta, "rev. ng: a unified binary analysis framework to recover cfgs and function boundaries," in *Proceedings of the 26th International Conference on Compiler Construction*, 2017, pp. 131–141.
[59] A. Di Federico, P. Fezzardi, and G. Agosta, "rev. ng: A multi-architecture framework for reverse engineering and vulnerability discovery," in *2018 International Carnahan Conference on Security Technology (ICCST)*. IEEE, 2018, pp. 1–5.
[60] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX annual technical conference, FREENIX Track*, vol. 41, no. 46. California, USA, 2005, pp. 10–5555.
[61] K. Kirchner and S. Rosenthaler, "bin2llvm: analysis of binary programs using llvm intermediate representation," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, 2017, pp. 1–7.
[62] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*. Springer, 2011, pp. 463–469.
[63] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.
[64] J. Miecznikowski and L. Hendren, "Decompiling java bytecode: Problems, traps and pitfalls," in *International Conference on Compiler Construction*. Springer, 2002, pp. 111–127.
[65] M. H. Halstead, *Machine-independent computer programming*. Spartan Books, 1962.
[66] B. C. HOUSEL III, *A study of decompiling machine languages into high-Level machine independent languages*. Purdue University, 1973.
[67] C. Cifuentes, "A structuring algorithm for decompilation," in *Proceedings of the XIX Conferencia Latinoamericana de Inform atica*, 1993, pp. 267–276.
[68] K. Burk, F. Pagani, C. Kruegel, and G. Vigna, "Decomperson: How humans decompile and what we can learn from it," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2765–2782.
[69] "Ida pro," accessed: 2025-04-16. [Online]. Available: https://hex-rays.com/ida-pro
[70] "Binary ninja," accessed: 2025-04-16. [Online]. Available: https://binary.ninja/
[71] "Ghidra," accessed: 2025-04-16. [Online]. Available: https://github.com/NationalSecurityAgency/ghidra
[72] "Retdec," accessed: 2025-04-16. [Online]. Available: https://github.com/avast/retdec
[73] "Java decompiler," accessed: 2025-04-16. [Online]. Available: https://java-decompiler.github.io/
[74] "Jadx," accessed: 2025-04-16. [Online]. Available: https://github.com/skylot/jadx
[75] "Fernflower," accessed: 2025-04-16. [Online]. Available: https://github.com/fesh0r/fernflower
[76] S. Drossopoulou, D. Wragg, and S. Eisenbach, "What is java binary compatibility?" in *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1998, pp. 341–361.

APPENDIX

CPG node counts and vulnerability scan results before and after decompiling. By observing the changes in AST nodes before and after decompilation, we can determine whether the original syntax has been restored. Similarly, comparing the number of CFG nodes before and after decompilation can help us judge whether the original control flow has been restored. It is worth noting that in Joern, Method and Call nodes do not only represent the definition and invocation of ordinary functions or class methods, but they also treat all operators, (e.g., $+$, $-$, $\star$), as methods and invocations. This results in a higher number of Method and Call nodes in the code property graph, but it also better reflects the structure of the code.

TABLE X
CPG NODE COUNTS AND VULNERABILITY SCAN RESULTS BEFORE AND AFTER DECOMPILING OF SQLI AND XSS PATTERN DATASET

Patterns	Nodes^{+}	Nodes*	AST^{+}	AST*	CFG^{+}	CFG*	Method^{+}	Method*	Call^{+}	Call*	Paths^{+}	Paths*
SQLi pattern dataset												
check_type_preg_match_fixed	272	263	265	255	226	217	19	18	42	41	2	2
complex_querybuild_vuln	792	826	782	809	680	708	43	46	152	150	3	4
dao_sink_vuln	534	530	526	521	455	452	32	32	102	99	2	2
database_global_array_vuln	241	233	234	225	197	188	18	17	33	32	2	2
database_global_vuln	235	227	227	218	190	181	18	17	32	31	2	2
database_static_method_vuln	359	352	351	342	287	276	28	27	48	46	1	1
db_quote_fixed	268	257	261	249	222	211	19	18	40	38	2	2
db_wrapper_vuln	272	261	265	253	218	207	19	18	41	39	3	3
eventmanager_vuln	599	588	588	577	482	475	32	31	118	113	0	0
explode_implode_vuln	270	259	263	251	223	212	19	18	42	40	2	2
failed_san_first_two_vuln	338	328	329	319	285	277	22	21	57	56	2	2
failed_san_not_init_vuln	284	273	276	264	233	222	20	19	43	41	2	2
func_get_args_vuln	269	261	261	253	219	213	19	19	43	41	0	0
__get_method_exists_vuln	383	380	374	370	312	309	27	26	56	54	0	0
getparams_vuln	322	311	314	302	260	250	23	22	48	45	2	2
get_set_vuln	318	317	310	308	256	255	21	21	50	48	2	2
inerhit_query_build_vuln	456	466	447	456	371	383	32	32	64	67	0	0
json_decode_vuln	393	328	382	318	320	259	27	23	57	46	2	2
plugin_vuln	1524	1530	1508	1514	1306	1316	80	80	341	340	0	0
put_get_env_vuln	262	251	255	243	217	206	19	18	41	39	0	0
redirect_before_sink_vuln	279	261	272	253	234	216	21	19	43	41	2	2
reflectionclass_vuln	400	407	390	396	318	325	30	30	57	57	0	0
sanitize_function_exists_fixed	306	277	299	269	252	223	22	19	45	41	2	2
sanitize_global_db_var_fixed	494	514	483	502	400	420	34	35	67	64	0	0
sanitize_htmlentities_fixed	340	318	332	309	276	255	24	22	50	45	2	2
simple_sample_vuln	251	240	244	232	205	194	17	16	39	37	2	2
singleton_classes_vuln	402	404	391	392	314	316	28	28	60	58	0	0
singleton_set_vuln	328	335	320	325	259	262	23	24	49	49	0	0
singleton_store_class_vuln	459	473	450	462	363	373	31	32	73	73	0	0
singleton_vuln	340	351	332	341	273	280	24	25	50	50	0	0
sink_imported_function_vuln	300	286	293	278	242	229	19	18	50	47	2	2
soap_vuln	375	367	366	357	310	302	27	26	55	53	0	0
source_imported_var_vuln	264	253	257	245	210	199	18	17	40	38	0	0
source_stored_class_vuln	405	413	397	404	339	347	29	30	57	57	0	0
sprintf_vuln	251	240	244	232	205	194	17	16	39	37	2	2
str_replace_fixed	347	334	339	326	287	276	20	19	64	62	2	2
whitelist_array_fixed	299	288	291	280	249	240	18	17	55	54	2	2
XSS pattern dataset												
class_variable_str_assignment_vuln	147	154	143	148	111	114	13	13	14	14	0	0
db_source_dao_vuln	467	498	458	488	381	410	30	31	72	76	1	1
db_source_inheritance_vuln	466	491	457	481	381	404	30	31	66	68	1	1
db_source_wrapper_vuln	552	592	541	580	426	466	35	37	81	87	0	0
failed_san_conditional_vuln	257	257	252	252	224	224	16	16	51	51	4	4
failed_san_insufficient_array_vuln	118	120	114	115	89	90	9	9	14	14	2	2
failed_san_insufficient_string_vuln	106	109	102	104	77	78	9	9	8	8	1	1
failed_san_stripslashes_vuln	60	62	57	58	41	41	5	5	6	6	1	1
failed_san_str_replace_vuln	76	78	73	74	57	57	5	5	10	10	1	1
global_keyword_vuln	70	77	66	72	46	52	7	9	7	7	0	0
global_require_vuln	66	68	63	64	39	39	6	6	4	4	0	0
la_brackets_vuln	63	63	59	58	41	40	5	5	4	4	1	1
la_list_vuln	63	63	59	58	41	40	5	5	4	4	1	1
return_by_reference_vuln	87	87	82	82	60	60	5	5	14	14	1	1
san_htmlentities_fixed	54	56	51	52	35	35	5	5	4	4	1	1
san_htmlpurifier_fixed	83888	83477	83646	83065	76614	76279	1894	1865	23555	23367	1	1
san_htmlspecialchars_fixed	54	56	51	52	35	35	5	5	4	4	1	1
san_preg_replace_fixed	63	65	60	61	44	44	5	5	5	5	1	1
san_reset_not_vaild_fixed	66	68	63	64	47	47	6	6	6	6	1	1
simple_xss_vuln	48	50	45	46	29	29	4	4	3	3	1	1
singleton_vuln	210	214	203	205	160	160	20	20	22	22	0	0
sink_print_r_vuln	60	62	57	58	41	41	5	5	6	6	1	1
source_cookie_vuln	48	50	45	46	29	29	4	4	3	3	0	0
source_foreach_vuln	99	101	96	97	78	78	8	8	13	13	1	1
source_server_self_vuln	48	50	45	46	29	29	4	4	3	3	0	0
template_ob_vuln	98	93	94	89	68	63	11	10	11	11	0	0
template_xml_vuln	127	127	122	122	99	99	10	10	15	15	0	0
unc_call_user_func_vuln	77	76	73	70	53	50	7	7	7	7	0	0
unc_call_user_vuln	60	59	57	55	40	37	6	6	4	4	0	0
unc_string_func_vuln	64	66	61	61	43	43	6	6	5	5	0	0
unserialize_vuln	60	62	57	58	40	40	5	5	6	6	1	1

$^{+}$: CPG nodes of each type and vulnerability paths in the original pattern,
*: CPG nodes of each type and vulnerability paths in the decompiled pattern