New PHP Language Features Make Your Static Code Analysis Tools Miss Vulnerabilities

Lin Wang, Yuan Zhang, Xin Tan, Shengke Ye and Min Yang
Fudan University, Shanghai, China
{wang_lin@, yuanxzhang@, 18212010028@, 21212010046@m., m_yang@}fudan.edu.cn

***Abstract*—Due to the nature of directly interacting with user inputs, PHP applications are susceptible to taint-style vulnerabilities. To detect such vulnerabilities, Static Code Analysis Tools (SCATs) are widely used for their broad code coverage and scalability. Modeling language features (i.e., to represent and simulate the behavior of program codes) is the keystone of SCATs' vulnerability detection capabilities. Meanwhile, being an actively maintained language, the PHP community introduces several new language features almost every year, rendering many unmodeled features. Though efforts have been made to reduce the number of unmodeled features, e.g., proposing new modeling methods, the impact of the introduction of new PHP features on SCAT during the language evolution is not well-conscious and systematically assessed. To fill the gap, this paper performs a systematic study of new language features and their impact on the ability of SCATs to detect taint-style vulnerabilities in PHP codes. To be specific, we identify 25 widely-used new language features that potentially compromise SCATs' vulnerability detection capabilities. Besides, we assess the impact of these new features on five open-source SCATs and show that the vulnerability detection ability is significantly compromised, with each SCAT affected by 10 features on average. To mitigate the impact, we conduct a theoretical analysis to diagnose the underlying reasons and propose several effective adaptation strategies. Finally, we provide key insights and implications for various stakeholders in static code analysis, emphasizing the need for them to recognize and proactively address the potential effects of language evolution.**

***Index Terms*—PHP Language Feature, Static Code Analysis, Web Security, Vulnerability Detection**

I. Introduction

PHP is the dominant server-side programming language in Web development, with a market share of 77.4% [1], powering millions of websites worldwide. Given that PHP applications directly interact with user requests, they are susceptible to taint-style vulnerabilities—a class of vulnerabilities manifesting when user-supplied data are used for critical operations (e.g., command execution) without adequate sanitization [2], [3]. Taint-style vulnerabilities include command injection, SQL injection, path traversal, and other widespread types of high-risk vulnerabilities [4], [5] that pose serious risks to web applications, ranging from unauthorized data access to data to potential compromise of the entire system. To detect taint-style vulnerabilities, static code analysis is widely used to secure the PHP code [6]–[8]. Unlike dynamic analysis, which requires code execution, static code analysis examines the entire source code at a static level, ensuring comprehensive code coverage and efficient handling of large codebases.

To develop a practical Static Code Analysis Tool (SCAT), modeling language features (i.e., understanding and simulating the semantics of programming codes) is crucial [9]–[11]. Meanwhile, as an actively maintained language, PHP has been updated every year and introduces several new features in each new version with the aim to enhance the language capability or optimize the programming experience (e.g., the built-in `iterable` type in PHP 7.1). In particular, from 2016 to 2022, the PHP community released seven new versions, introducing more than 70 new features. Unfortunately, the existing SCATs might not model these new language features, thus undermining their effectiveness. For example, in 2021, five CVEs are discovered in Symfony (the most popular PHP application framework [12], [13]). When analyzing these vulnerabilities using PHPJoern (a popular open-source SCAT for PHP), it fails to detect two out of the five CVEs [14], [15]. This oversight is attributed to the PHPJoern's lack of support for a specific language feature introduced in PHP 7.1, namely *Keys in* `list()`, which is present in the vulnerable code [16], [17]. Moreover, this feature prevents the exposure of vulnerabilities not only in Symfony but also in the downstream projects of Symfony, such as Laravel [18], Drupal [19], Joomla [20], etc, dramatically increasing the security risk to end-users. In conclusion, the new language features introduced during the PHP evolution may have severe impacts on the ability of SCATs in vulnerability detection.

However, the impact of new language features on the ability of SCATs to detect taint-style vulnerabilities has not yet been well-conscious and quantitatively assessed. Existing works primarily concentrate on two aspects. First, they investigate methods to precisely model challenging features in SCATs, aiming to improve their accuracy and effectiveness [9], [10]. Second, researchers identify the most frequently used PHP features to guide prioritization in static analysis modeling [21]–[23]. However, these works overlook the real impact of PHP feature evolution on SCATs. Besides, they mainly focus on old PHP features (i.e., features before PHP 7.0). Consequently, the impact of newer language features on SCATs' capabilities remains inadequately understood and quantitatively assessed. This lack of comprehensive analysis of recent PHP features potentially limits the effectiveness of current static analysis approaches in addressing modern PHP codebases.

To fill the gap, this paper performs a systematic study of new language features and their impact on the ability of SCATs to detect taint-style vulnerabilities in PHP codes. Specifically, our

study makes the following contributions by addressing three research questions:

• *RQ1 Landscape: What are the new language features that potentially affect vulnerability detection, and how prevalent are they in real-world PHP projects?* We identify 25 new language features by thoroughly examining the official documentation of PHP 7.1∼8.0. In addition, we show that these new language features are widely used in real-world PHP projects, present in 72.43% of the top 1,000 PHP projects. This result justifies the necessity of further research on the impact of the new features.

• *RQ2 Impact: How new language features impact the ability of SCATs to detect taint-style vulnerabilities?* We evaluate five open-source SCATs using a self-constructed test suite. Our test suit can effectively isolate the influence of new language features from other factors that may affect vulnerability detection. As a result, we can precisely identify how new features affect each SCAT's performance through the test suite. Our findings reveal that the ability of all tested SCATs is significantly compromised by the new features, with an average of 10 unsupported features per SCAT. These results underscore the critical importance of actively adapting SCATs to accommodate new language features, as failure to do so may result in overlooking severe vulnerabilities.

• *RQ3 Adaptation: Why do SCATs fail and how to adapt them to new language features?* We first conduct a theoretical analysis of SCATs to locate the internal steps that fail due to new language features. Based on this analysis, we propose solutions and apply them to enhance an open-source SCAT, PHPJoern. The enhanced tool is then evaluated using our test suite. Results indicate that the enhanced PHPJoern successfully detects vulnerabilities in codes incorporating new language features, thus validating the effectiveness of the proposed solutions. This approach provides valuable insights and a practical reference for developers seeking to adapt their SCATs to new language features.

Finally, we summarize the experimental insights and broad implications for various stakeholders in this field. For SCAT developers, our study prompts them to implement measures that mitigate the potential negative impacts of new language features. For SCAT users, the findings emphasize the importance of considering whether a particular SCAT supports the language features present in the code under analysis when selecting a tool. For researchers, our study highlights the potential effects of language evolution, encouraging further investigation into this area. For the security community, it motivates the need for shared static analysis libraries, which significantly facilitate adaptation to new language features.

II. Landscape (RQ1)

This section defines the scope of new PHP language features studied in this paper and validates their significance through an analysis of their adoption rates in real-world PHP projects. In particular, we set to address the following two sub-questions:

- *RQ1.1*: What new language features may affect taint-style vulnerability detection in recent PHP releases?
- *RQ1.2*: How often are they used in real-world PHP projects?

A. Targeted Feature Selection (RQ 1.1)

Criteria. SCATs rely on converting source code into an intermediate representation, such as an Abstract Syntax Tree (AST), to facilitate vulnerability detection. This process is fundamental to a SCAT's ability to comprehend and analyze code effectively. Consequently, this study focuses on new language features that potentially impede a SCAT's code comprehension capabilities. Specifically, we identify two categories of such features: those introducing new tokens and those introducing new syntax. Tokens are basic units of a programming language, including keywords and symbols, while syntax refers to the rules that govern how these tokens can be combined to form valid code. New tokens or syntax can challenge SCATs' parsing mechanisms, potentially affecting their ability to accurately analyze code for vulnerabilities.

Method. The identification of such language features involves a two-step process. We first determine whether a new feature introduces a new token by generating AST for code snippets incorporating the features and examining the resulting AST nodes. We verify through documentation whether these node types are specifically introduced to accommodate the feature post its inception. The presence of such nodes indicates the introduction of a new token. If no new token is identified, we then proceed to assess whether a new feature introduces a new syntax. This is done by attempting to compile feature-present code snippets using a PHP compiler that predates the feature's introduction. A syntax error during this compilation process signifies the presence of new syntax.

Results. Our analysis covers all core features of PHP 7.1, 7.2, 7.3, 7.4, and 8.0. The non-core features are developed by third parties, and newer released features (i.e., features from PHP 8.1 or 8.2) may have yet to gain widespread adoption, so we don't take them in this study. The AST generation tool is PHP-AST [24]. Consequently, we analyze a total of 60 new language features, and 25 out of them are identified as the targeted features for our study. The brief of targeted features is displayed in Table VIII (in Appendix).

B. Feature Usage Analysis (RQ 1.2)

To investigate the usage frequency of features, we first propose a method to detect the targeted features within PHP code. Then, we apply the detection method on 816 popular PHP projects and report our findings.

Feature Detection Method. To detect specific language features in code, there are three primary approaches: string-based matching, machine learning-based matching and AST-based matching. String-based matching identifies distinctive string patterns associated with a feature but often fails to capture all possible variations, resulting in low recall. Machine learning-based matching requires an extensive labeled dataset to learn code patterns [25], which is challenging to construct and maintain. In contrast, AST-based matching abstracts the code into a tree structure and detects new features on the structural representation [26]. It can identify novel syntactic constructs

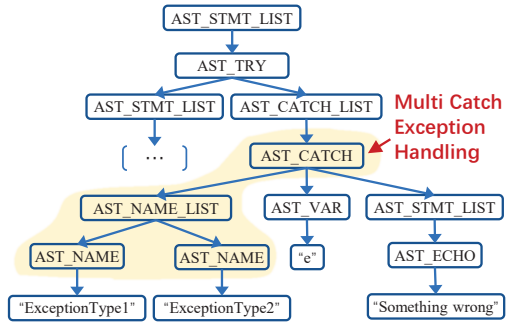

Fig. 1: Signature of *Multi catch exception handling*.

```php
try {
    $bar = foo();
} catch(ExceptionType1 | ExceptionType2 $e){
    echo ("Something wrong")
}
```

Listing 1: *Multi catch exception handling* example.

that may not be easily recognized by string patterns or existing machine learning models because it elides implementation details and captures the underlying code structure. Consequently, the AST-based method is selected for feature detection.

The implementation of the AST-based method requires the summarization of AST signatures, which are unique AST structures associated with specific language features. For features that introduce a new token, we use their introduced kind of AST nodes as their signature, as described in RQ1.1 (see §II-A). For features that introduce a new syntax, we first locate the kind of AST node representing the code structure where the new feature can be applied. Then, we investigate which child-parent connections among AST nodes represent the new syntax. Finally, we consider both nodes and their connections as the signature. The summarized signatures are present in Table VIII (in Appendix).

To detect a feature based on the summarized signature, we first traverse an AST to locate the AST node of the signature by comparing AST kind and flag. If the signature comprises the AST structure, we then examine the ancestors, descendants, or code lines of the node to check whether they satisfy the AST structure. We take feature *Multi catch exception handling* as an example to illustrate the process. This feature enables the handling of multiple exception types in a single catch block. A code example using this feature is shown in Listing 1, and its AST is displayed in Figure 1 (we omit the AST of `try` block because it isn't our focus) where structure in yellow shadow is the signature of this feature. When detecting the feature, we first locate the AST node with kind value `AST_CATCH`. Then, we search for the `AST_NAME_LIST` node within the children of the identified key node. Finally, we examine if the `AST_NAME_LIST` node has multiple `AST_NAME` nodes as children. If it is, the feature is present. Otherwise, the feature is absent.

TABLE I: Size and Scale of Collected PHP Projects in Year.

Last Release	# of Proj.	# of PHP Files	# of PHP Code Lines
2017	25	4,106	361k
2018	22	5,162	433k
2019	31	3,864	437k
2020	43	17,298	2,190k
2021	62	11,492	1,568k
2022	181	34,883	4,875k
2023	452	310,002	41,289k
Total	**816**	**386,807**	**51,253k**

TABLE II: Usage of New Features among Real-world Projects.

Language Features	% of Proj.	# of Proj.	# of Files
Void function	57.23%	467	51,214
Nullable type	49.75%	406	22,796
Class constant visibility	44.61%	364	14,085
Typed properties	31.25%	255	12,579
Symmetric array destructuring	30.51%	249	3,580
Arbitrary expression support for `new` *and* `instanceof`	28.55%	233	1,772
Union type	24.51%	200	3,133
`mixed` *type*	21.08%	172	3,158
`iterable` *pseudo-type*	18.75%	153	2,170
`object` *type*	17.89%	146	1,419
Arrow functions	17.65%	144	2,110
Constructor property promotion	13.97%	114	5,031
Static return type	13.11%	107	759
Non-capturing catch	11.40%	93	864
`match` *expression*	10.66%	87	462
Named argument	10.17%	83	1,066
Nullsafe operator	9.44%	77	632
`::class` *on objects*	8.95%	73	456
Multi catch exception handling	8.82%	72	368
`throw` *expression*	7.48%	61	191
Unpacking inside arrays	6.00%	49	182
Support for keys in `list()`	0.74%	6	9
Class constant dereferencability	0.12%	1	1
All Features	**72.43%**	**591**	**56,471**

Feature Usage Investigation. We first collect a number of real-world PHP projects. To be specific, we download the top 1,000 PHP projects from GitHub star ranking list [27]. For each project, we retrieve its latest released (tagged) version and record the corresponding release time. Then, we verify these projects and remove some of them if they ❶ have no released version; or ❷ have no PHP files (that are suffixed with .php/.phar/.php5); or ❸ are released earlier than the release time of PHP 7.1 (1st December 2016). Finally, our dataset contains 816 projects. We organize these projects by release year and provide their size and scale information in Table I. Next, we employ our proposed method to detect new language features among the collected projects. The process involves transforming PHP code into ASTs using PHP-AST(90), and searching the feature signatures based on these ASTs.

The results are presented in Table II, which show that 23 features are detected in 72.43% projects. Notably, 13 features are found to be used by at least 100 projects, underscoring their widespread acceptance. *Void function* emerges as particularly popular, utilized in 57.23% of the examined projects.

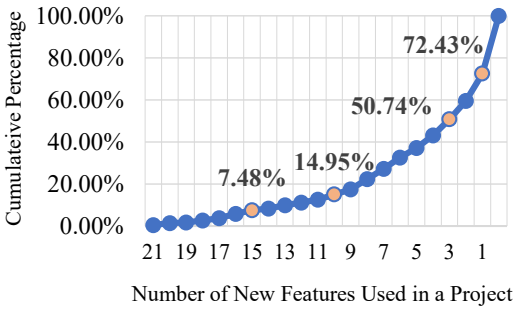

Fig. 2: Cumulative probability distribution of projects in terms of new feature usage.

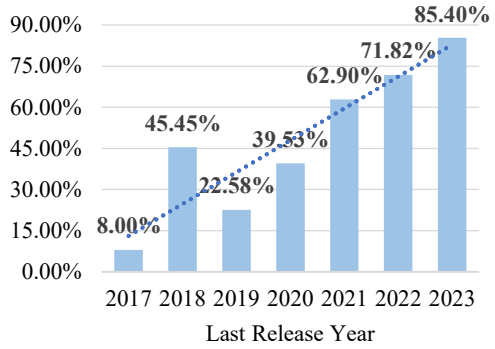

Fig. 3: Trend of new language feature adoption by year.

Nonetheless, two features, namely *Array Destructuring supports Reference Assignments* and *Null coalescing assignment operator*, are not detected in any project. This could be due to the infrequent use of references in PHP and the availability of alternative methods for achieving similar outcomes.

Figure 2 illustrates the cumulative probability distribution of projects using new PHP language features, providing insights into the feature usage across projects. The x-axis represents the number of new features used in a project, while the y-axis shows the cumulative percentage of projects. The data reveal that more than half of projects, specifically 50.74%, use at least three new features. Notably, 14.95% of projects employ ten or more new features and 7.48% implement fifteen or more. These statistics indicate that a significant portion of projects actively integrates new features, with some demonstrating extensive usage of these features.

Figure 3 illustrates the adoption trend of new PHP features, which shows the percentage of projects using at least one new feature within each year. It reveals a clear upward trend in the adoption of new features, with the percentage rising from 8.00% in 2017 to 85.40% in 2023. The overall increasing trend suggests that more recent PHP projects are increasingly likely to incorporate new language features. This can be attributed to the fact that new features are often more efficient and performant than older features, and the more recent projects are often developed using the latest version of PHP, which allows for the utilization of new features.

Conclusion: Our analysis identifies 25 new language features that potentially impact SCATs' ability to detect taint-style vulnerabilities. These features demonstrate significant adoption among real-world PHP projects, with a considerable number of projects incorporating multiple new features simultaneously. Furthermore, our temporal analysis reveals a clear upward trend in the adoption of these features, with more recent projects showing a higher likelihood of incorporating new language features.

III. Impact (RQ2)

In this section, we quantitatively assess to what extent the new language features compromise the ability of SCATs to detect taint-style vulnerabilities. Specifically, we propose an assessment method (i.e., constructing a test suite) to test the ability of five open-source SCATs and report our findings.

A. Assessment Method

To determine whether a feature would affect the ability of SCATs, we can directly examine their source code. However, it is labor-intensive and time-consuming. To address the limitation, we propose an automated approach: carefully crafting test cases that incorporate both a new language feature and a taint-style vulnerability, and testing a SCAT on the cases to evaluate the tool's effectiveness in detecting the vulnerability with the presence of new language feature.

Challenge. Constructing such a test suite is a nontrivial and challenging task. The reason is that the SCAT analysis process is complex and can be influenced by many factors, making it difficult to isolate the specific impact of new language features on vulnerability detection capabilities. For example, a SCAT's failure to detect a taint-style vulnerability may stem from its inadequate modeling of sensitive functions rather than the presence of a new language feature.

Idea. To isolate the influence of other factors, we follow the principle of controlled experiments. Specifically, there are two groups in the experiment, and they are identical except that one receives a *treatment* while the other does not. The group that receives the treatment in an experiment is called the experimental group, while the group that does not receive the treatment is called the control group. The control group provides a baseline that lets us see if the treatment has an effect. In our case, the treatment is the new feature. The experiment group case is a vulnerable code snippet with the feature present, while the control group case strictly follows the same semantics but without the feature. Our initial hypothesis is that if a feature affects a SCAT's ability to detect vulnerabilities, the SCAT is supposed to successfully detect a vulnerability in the control group but fails in the experimental group.

However, factors influencing SCATs' ability may exist in both experimental and control groups, making a SCAT fail to detect a vulnerability even in the control group and invalidating our initial hypothesis. To address this issue, we implement a strategy of code diversification, which involves creating multiple code snippets using different syntactic structures

Algorithm 1: Impact Assessment

Input : A SCAT t, targeted feature set $\{f\}$
Output: The unsupported features of t.

```
1  unsupported_feature ← []                          // to save the result
2  for f in {f} do
3      test_pairs ← GetTestPairs(f)
4      for (ctrl_case, exp_case) in test_pairs do
5          ctrl_res ← Test(t, ctrl_case)             // without feature
6          exp_res ← Test(t, exp_case)               // with feature
7          if ctrl_res is detected and exp_res isnot
             detected then
8              unsupported_feature.append(f)
9              break
10         end
11     end
12 end
13 return unsupported_feature
```

or sensitive functions for each feature. By creating diverse control-experimental test pairs for each feature, we enhance the probability of identifying at least one pair where the SCAT successfully detects the vulnerability in the control group. Consequently, we refine our hypothesis: if a feature affects a SCAT's ability, there is at least one pair where the SCAT successfully detects a vulnerability in the control group case but fails in the experimental group case.

New Approach. Following the above ideas, our test suite aims to evaluate the impact of 25 targeted new features on a SCAT's ability to detect taint-style vulnerabilities. The suite comprises multiple test pairs for each feature, with each pair consisting of two semantically equivalent pieces of vulnerable PHP code. One code piece is written with a targeted feature, annotated as an "experimental case", while the other, lacking the feature, is annotated as a "control case".

The testing procedure is presented in Algorithm 1. Firstly, we retrieve all test pairs related to a given feature. Then, we subject a SCAT to cases in each pair and record their vulnerability detection results (lines 4-6). Finally, we compare the results of each pair to conclude (lines 7-9). According to our hypothesis, if there is a pair where the SCAT successfully detects a vulnerability in its control group case but fails in the experimental group case, we consider the feature affects the SCAT's ability to detect vulnerability and mark the feature as "unsupported feature".

The other three possible results of a test pair indicate neither definitively "support" nor the "unsupport" of a new feature. To be specific, the results include: (1) The tool identifies vulnerabilities in the experimental group but fails in the control group. This indicates potential mistakes with our test cases, and we have eliminated them; (2) The tool fails to detect vulnerabilities in both cases. It indicates that certain factors existing in both control and experimental cases prevent the SCAT from detecting vulnerabilities, so we cannot draw any

```php
$user_data=$_GET['UserData']; // get data
$tainted=fn()=>$user_data; // assign
echo $tainted(); //apply
```

(a) Test case in the experimental group.

```php
$user_data=$_GET['UserData']; // get data
$tainted=function(){ // assign
    return $user_data;
}
echo $tainted(); // apply
```

(b) Test case in the control group.

Listing 2: A test pair of *Arrow functions*.

conclusion about the new feature; (3) The tool successfully identifies vulnerabilities in both cases. However, this does not necessarily imply that the SCAT supports the new feature, as it may bypass the analysis of the new feature to detect the vulnerability. For example, *Void function* adds a data type keyword, which allows the SCAT to overlook the data type and directly analyze and identify the tainted flow of a vulnerability.

Implementation. To implement our idea, we first design a code template for both experimental and control group cases. The template consists of three essential components necessary to trigger a taint-style vulnerability: user data retrieval, data assignment, and sensitive function employment. To illustrate, we present a test pair that applies the template for feature *Arrow functions* in Listing 2. In the experimental case (Listing 2a), user data are obtained through `$_GET` method, subsequently assigned to the variable `$tainted` through *Arrow functions* (`fn()=>`), and finally applied to the sensitive function `echo`. The control group case (Listing 2b) maintains an identical structure, with the key difference lying in the data assignment process where the new feature is substituted with equivalent code that performs the same function.

Secondly, we determine the code structures and sensitive functions to diversify the test pairs for each feature and enhance the comprehensiveness of our assessment. In terms of the three components in a test case, we only use `$_GET` for user data retrieval because we assume that this PHP fundamental function to access user requests should be supported by most SCATs and it is not necessary to diverse. For data assignment, it involves the implementation of the new feature so we diverse this part by encompassing various implementation methods. For example, features with multiple implementation possibilities, such as *Arbitrary expression support for `new` and `instanceof`* which can use either `new` or `instanceof`, are represented by separate test pairs for each implementation method. This approach results in a total of 38 implementations for the 25 features. For sensitive function employment, we design to include a broad spectrum of common sensitive functions. In particular, we select 29 functions associated with 7 prevalent taint-style vulnerabilities for our test cases. We list the detailed information of the 38 implementations in Appendix and 29 sensitive functions in Table III.

Based on our design, we manually compose for ❶ 1 user

TABLE III: Sensitive Functions.

Vulnerability	# of Funcs.	Funcs.
Directory traversal	5	`fopen`, `dir`, `dirname`, `opendir`, `scandir`
Command injection	7	`exec`, `passthru`, `system`, `proc_open`, `pcntl_exec`, `shell_exec`, `popen`
Cross site scripting	3	`echo`, `print`, `print_r`
SQL injection	6	`query`, `pg_query`, `mysql_query`, `mysqli_query`, `pg_send_query`, `mysqli_real_query`
Code injection	1	`eval`
Arbitrary file inclusion	4	`include`, `include_once`, `require`, `require_once`
Arbitrary file reading	3	`readfile`, `file`, `file_get_contents`

TABLE IV: Results of Impact Assessment.

Features	Progp.	RIPS	PHPJ.	WAP	phpS.
Named argument	×	×	×	×	×
Nullsafe operator	×	×	×	×	-
Support for keys in `list()`	-	×	×	×	×
Class constant dereferencability	×	-	×	×	×
Arrow functions	×	-	×	×	×
Null coalescing assignment operators	×	-	×	×	×
Constructor property promotion	×	-	×	-	-
Symmetric array destructuring	-	×	×	-	-
`match` expression	×	-	×	-	-
Class constant visibility	-	-	×	×	-
Typed properties	-	-	×	×	-
`throw` expression	×	-	×	-	-
Nullable type	-	-	×	×	-
Array Destructuring supports Reference Assignments	-	-	×	×	-
Union type	-	-	×	×	-
Unpacking inside arrays	-	-	×	-	-
Static return type	-	-	×	-	-
Non-capturing catch	-	-	×	-	-
Multi catch exception handling	-	-	×	-	-
Arbitrary expression support for `new` and `instanceof`	-	-	×	-	-
Void function	-	-	-	×	-
`::class` on objects	-	-	-	×	-
`iterable` pseudo-type	-	-	-	-	-
`object` type	-	-	-	-	-
`mixed` type	-	-	-	-	-
# of Unsupported Features	**8/25**	**4/25**	**20/25**	**13/25**	**5/25**
# of Unsupported 7 Widely-Adopted Features	**0/7**	**1/7**	**6/7**	**5/7**	**0/7**

×: the feature is not supported by the SCAT.
The top 7 widely-adopted features are in bold.

data retrieval code snippet; ❷ 38 feature-present data assignment snippets and corresponding 38 feature-absent equal-semantics snippets; ❸ 29 sensitive function employment snippets. Next, we automatically assemble the user data retrieval snippet, a data assignment snippet pair, and a sensitive function employment snippet in turn to generate test pairs. In the end, we generate 1102 ($1 \times 38 \times 29$) test pairs for the test suite.

B. Assessment Results and Analysis

We test the impact of new features on five SCATs using our test suite and set to answer the following questions:

- *RQ2.1*: What is the overall impact of the new features on the SCATs?
- *RQ2.2*: How about the impact of the seven most widely adopted features on the SCATs?
- *RQ2.3*: How are the unsupported features distributed across PHP versions?
- *RQ2.4*: What features have the most impact?
- *RQ2.5*: Is newer released SCAT less impacted by new language features?

SCAT Selection. We use keywords like "PHP code analysis" to search for related papers and open-source tools. We then examine their title, abstracts, or documents to assess whether they aim to develop or release a code analysis tool. For related papers, we iteratively evaluate relevance of the citations and references of them, repeating this process until no new relevant works are identified. As a result, we get 17 PHP code analysis tools as the initial list.

We select SCATs that are ❶ designed for detecting taint-style vulnerabilities in PHP. For example, LChecker [28] and PHPstan [29] are excluded as they are intended for tricky bugs, not vulnerabilities. ❷ employing the static analysis technique. The detection tools rely on deep learning techniques like DeepTective [30] are omitted; ❸ open-source and recently released (i.e., after 2015). Finally, the latest version and release time of selected SCATs are:

1) Progpilot (v1.0.2, January 2023 [31]);
2) RIPS [10], [11] (v0.55, June 2017 [32]);
3) PHPJoern [33] (April 2017 [34]);
4) WAP [35], [36] (v2.1, November 2015 [37]);
5) phpSAFE [38] (April 2015 [39]).

Note that the open-source version of PHPJoern lacks the implementation of vulnerability detection. Hence, we implement the detection function by ourselves as described in their paper [33]. Our experiments, as reported in §IV, demonstrate the correctness of our implementation. The running environment of Progpilot, RIPS, phpSAFE, and PHPJoern is PHP 8.0, and that of WAP is JDK 1.8.

RQ2.1. The results are displayed in Table IV. It shows that the selected SCATs' ability to detect taint-style vulnerabilities is significantly impacted by the new features. Every SCAT has new features that it does not support, with 10 unsupported features on average for each. In addition, the extent to which SCATs are affected differs. RIPS exhibits minimal susceptibility to targeted language features, affected by only four, whereas phpSAFE and Progpilot follow, being affected by five and eight features, respectively. In contrast, WAP and PHPJoern are the most vulnerable, impacted by 13 and 20 features, respectively.

RQ2.2. We further count the results of the most popular seven features (based on Table II). As shown in the last line of Table IV, the capability of the five SCATs to handle the top 7 new features differs greatly. Progpilot and phpSAFE are not impacted by the top 7 features, while WAP and PHPJoern have 6 and 5 unsupported features, respectively. In addition, SCAT's support for these 7 popular new features is generally better and receives less impact than the other 18 features. This may be partly because developers carefully handle popular features.

TABLE V: Distribution of Unsupported Features in PHP Release Versions.

Release	Progp.	RIPS	PHPJ.	WAP	phpS.	Total*
7.1	0	2	5	4	1	6
7.2	0	0	0	0	0	0
7.3	0	0	1	1	0	1
7.4	2	0	4	3	2	4
8.0	6	2	10	5	2	11

* It means the total number of distinct unsupported features among all SCATs in the PHP release version.

Another reason may be that these popular features have less impact on flow analysis, thus less impacts.

RQ2.3. We count the distribution of versions for unsupported features. As shown in Table V, almost every release introduces new features that impact SCAT's vulnerability detection capabilities. This corroborates the prevalence of introducing unsupported features during language evolution.

RQ2.4. We further analyze the semantics of the unsupported features. We find that the features that have a greater impact on control flow or data flow have a greater impact on SCAT. Take the two features that affect all five SCATs as an example. *Named argument* introduces a new way of passing parameters, and *Nullsafe operator* introduces a new operator variable assignment. Both of them are related to data flow, which is important for taint-style vulnerability detection.

RQ2.5. In Table IV, we arranged the SCATs by release time from newest to oldest, which shows that the more recently released software does not exhibit significantly improved compatibility with the features. Therefore, we can infer that even the latest SCAT tools may not pay enough attention to recently introduced new features.

Conclusion: SCATs' vulnerability detection ability is heavily affected by the introduction of new features, especially by the features related to data or control flow. Moreover, almost every PHP version introduces unsupported features. This challenge persists over time, showing no signs of abatement.

IV. Adaptation (RQ3)

In this section, we analyze the root cause of SCATs not supporting new features and explore the adaptation strategies. In particular, we set out to answer the following sub-questions:

- RQ3.1: Why do SCATs fail to detect vulnerabilities? What are the internal steps that lead to the failures?
- RQ3.2: What strategies could be taken to adapt SCATs to the new features?
- RQ3.3: How do the proposed strategies work in practice?

A. Theoretical Analysis (RQ 3.1)

SCAT Breakdown. The typical process of analyzing vulnerabilities by a SCAT is shown in Figure 4. We introduce its four main components in the following.

1) Code Parsing: SCAT first parses source code into Intermediate Representation (IR), a crucial step that creates a more manageable and analyzable form of the code to aid further flow analysis. Most SCATs adopt the AST as their IR format, as it accurately captures the abstract syntactic composition of the source code, preserving its logical structure while abstracting away syntactic details.

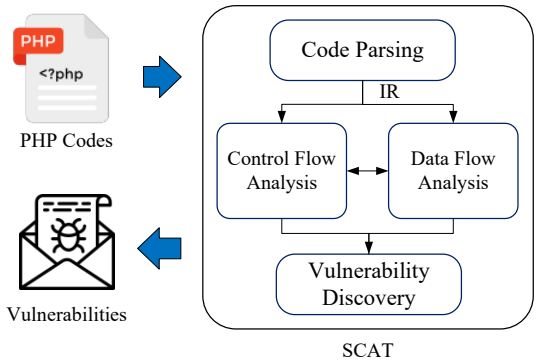

Fig. 4: Typical SCAT Analysis Procedure.

2) Control Flow Analysis: SCAT performs control flow analysis to model the execution path of programs. On the one hand, SCAT analyzes the control structures inside of a function, such as loop statements, to model the intra-procedural control flow. On the other hand, SCAT analyzes the function call statements and call targets to model the inter-procedural control flow.

3) Data Flow Analysis: SCAT further performs data flow analysis along the control flow path to comprehensively evaluate program behavior. It traces data usage by examining assignments, function calls, and other data-manipulating operations. This process enables SCAT to track how variables are defined and utilized throughout program execution.

4) Vulnerabilities Discovery: SCAT leverages the results of control flow and data flow analyses to detect taint-style vulnerabilities. In particular, it employs specialized algorithms to identify anomalous data flows that may indicate security weaknesses. These algorithms trace the propagation of untrusted user input (tainted data) through the analyzed flows. SCAT flags a potential vulnerability when it discovers a path allowing tainted data to reach a sensitive operation or function without proper validation or sanitization.

Diagnosis of Failures. The introduction of new features generates three distinct issues involving three of the above components, elaborated below:

1) Incomplete Code Parsing: The introduction of new language features in code typically involves either new tokens or new syntax, which can significantly impact the code parsing process of static analysis tools. When new tokens are introduced, the parser may fail to recognize these previously undefined lexemes, potentially misclassifying them or generating errors during tokenization. Similarly, the incorporation of new syntax can challenge the parser's ability to construct a valid parse tree, as these syntactic structures may not conform

TABLE VI: Results of Diagnosis.

Failed Step	Causes	Involved Language Features
Code Parsing	New token or syntax	All features
Control Flow Analysis	New function definition	*Arrow functions*
	New function utilization	*Arrow functions*, *Nullsafe operator*, *Arbitrary expression support for* `new` *and* `instanceof`, *Class constant dereferencability*
Data Flow Analysis	New data definition	*Symmetric array destructuring*, *Support for keys in* `list()`, *Constructor property promotion*
	New data utilization	*Unpacking inside arrays*, `match` *expression*, *Constructor property promotion*, *Nullsafe operator*

to the existing grammar rules. Consequently, the IR generated by the parser becomes incomplete or inaccurate.

2) Missing Control Flow: Some new features introduce new methods of function definition and invocation. For example, feature *Arrow functions* introduces an alternative method of defining an anonymous function. Consequently, existing SCATs may fail to correctly identify the new execution paths, resulting in incomplete or inaccurate control flow graphs.

3) Missing Data Flow: Some new features introduce new methods of data definition and utilization. *Symmetric array destructuring* introduces a new method for using data in `list()`. Consequently, existing SCATs, which typically rely on predefined data flow models, may fail to capture these nuanced data interactions introduced by new language features. This limitation can lead to incomplete or inaccurate data flow graphs, potentially causing the tools to overlook critical data propagation paths.

In the end, based on the preceding analysis, we present the diagnosis of each language feature in Table VI. It indicates that all features introduce either new tokens or syntax, thus affecting the code-parsing phase. Furthermore, four features affect control flow analysis as they introducing new methods of function definition or utilization. Moreover, seven features are found to impact data flow analysis through the introduction of new data definitions or utilization mechanisms.

B. Adaptation Strategy (RQ 3.2)

To mitigate the aforementioned issues, we develop the following adaptation strategies:

Enhancing Code Parsing. To fix a parser, it typically involves: adding new IR elements, such as AST nodes, for newly introduced tokens; adjusting the parsing logic to recognize and process the new syntax; and conducting comprehensive testing to ensure the correct handling of new features. However, this process is time-intensive and prone to errors. As an alternative, we can utilize and maintain an up-to-date version of an existing code parser. This approach allows us to circumvent the substantial costs of developing a parser while ensuring the accuracy of the generated IR.

Improving Control/Data Flow Analysis. We first extend the SCAT to recognize the new elements introduced by new features to the IR, such as the new types of AST nodes, as the flow analysis relies on the IR structure. Next, the control flow analysis algorithms are updated. It involves a systematic examination of how the new language feature affects program execution and updating the control flow generation process accordingly. Likewise, the data flow analysis is adjusted by comprehending how the new features affect data propagation, including newly introduced methods of data definition, utilization, etc. Throughout this entire process, new edges and nodes may be incorporated to accurately represent the control or data flow of the new features.

C. Adaptation Practice and Evaluation (RQ 3.3)

To further validate the effectiveness of our proposed adaptation strategies, we apply them to enhance PHPJoern as a representative case study. This choice is motivated by PHPJoern's popularity as an analysis platform for PHP [40]–[42], and its exhibition of the highest number of unsupported features during our impact assessment (see Table IV). Then, we evaluate the enhanced PHPJoern from three aspects to demonstrate the effectiveness of our proposed strategies.

PHPJoern Enhancement. Firstly, we upgrade the third-party parser. PHPJoern uses PHP-AST, as its parser but its version is outdated and deprecated. Hence, we upgrade it to the latest version capable of handling features before PHP 8.0. Secondly, we extend PHPJoern to recognize the new AST nodes and structures introduced by the new features. This involves updating the AST node definitions in PHPJoern's codebase, potentially creating new classes or extending existing ones, and implementing appropriate visitor methods for these new node types to ensure proper traversal. Finally, the control flow and data flow analyzers are updated. For control flow analysis, we locate the function definitions and usage introduced by the new features, analyze the involved function call relationships, and then add the missing inter-process control flow edges. For data flow analysis, we model the new data propagation methods, i.e., data definition and utilization, and modify propagation rules to account for novel data passing mechanisms introduced by the new language features.

Evaluation. We implement the enhanced PHPJoern and evaluate it by comparing it with the original PHPJoern from three distinct aspects: code parsing capability, proficiency in control and data flow analysis, and the impact of new language features on vulnerability detection.

1) Code Parsing Capability. We evaluate it by comparing the number of successfully parsed PHP files. To be specific, we select 10 applications with the most used features based on the results of feature usage in §II-B (RQ1). Then, we subject their PHP files to the original and enhanced PHPJoern respectively. The tool would report an error for a file if it encounters a code structure that can't be handled. Therefore, we take the file without reporting an error as a successful parsing file and calculate the total number. As presented in Table VII, the results indicate that the enhanced PHPJoern significantly increases the parsing success rate, parsing more than 95% files for 7 out of 10 applications. Upon closer inspection, we

TABLE VII: Results of Comparison Between the Original and Enhanced PHPJoern.

Applications	File Parsing Success Rate (Original→Enhanced)	Increased Parsed Files	Increased CPG Nodes	Increased CPG Edges
EasyAdminBundle	60.22%→100%	181	119,399	20,266
Laravel Framework	80.91%→99.67%	290	419,592	76,715
ORM	59.66%→100%	119	86,940	14,954
Symfony	69.70%→97.73%	1,807	1,129,774	235,081
Spiral Framework	59.29%→80.50%	370	127,480	21,228
API Platform Core	59.92%→79.27%	197	143,228	24,968
PIM Community Dev	70.43%→97.27%	3,449	1,642,119	283,064
Phpactor	62.41%→99.97%	1,173	513,036	94,923
OroPlatform	77.28%→99.99%	4,343	3,303,346	64,1256
TYPO3	69.04%→88.04%	902	982,988	263,955

find that the failure of the parsing is primarily due to some deprecated language features that are not supported.

2) Proficiency in Control and Data Flow Analysis. PHPJoern generates Code Property Graphs (CPGs) based on the IR after code parsing, which incorporates both data and control flow. Therefore, our evaluation focuses on the completeness of the generated CPG. To be specific, we compare the number of CPG nodes and edges produced by the enhanced PHPJoern against the original version. As presented in the final two columns of Table VII, the enhanced PHPJoern greatly increases the number of CPG nodes and edges, yielding at least 10,000 increments to each project. These results indicate a significant improvement in the completeness of the generated CPGs, validating the effectiveness of the enhancements made to PHPJoern.

3) Impact of New Language Features. To assess the impact, we test the ability of the enhanced PHPJoern in detecting vulnerabilities with the presence of new language features. Specifically, we reuse the test suite constructed in §III-B (RQ2) and follow the identical steps. The results reveal a significant improvement: the enhanced PHPJoern gets *zero* unsupported features, in stark contrast to the original version which failed to support 20 features. This demonstrates the efficacy of our proposed adaptation strategies in enabling SCATs to accommodate new language features.

Conclusion: New language features affect source code parsing and control/data flow analysis during vulnerability detection. To mitigate the issue, we propose adaptation strategies to upgrade code parsers and correct inaccuracies of control and data flow. We then implement these strategies to enhance PHPJoern. Our evaluation results indicate that the enhanced PHPJoern outperforms its original version in terms of handling new features. This demonstrates the effectiveness of our strategies, suggesting their potential applicability in adapting SCATs to evolving programming languages.

V. Implications

This section synthesizes the experimental findings and elucidates their broad implications for key stakeholders in the field, including SCAT developers, users, researchers, and the broader security community. These insights aim to inform and guide future developments and applications in the domain of static code analysis and vulnerability detection.

- **To SCAT Developers.** Our research reveals that programming language evolution can significantly affect the ability of SCATs to detect vulnerabilities. Therefore, it is required for SCAT developers to adapt their tools to new language features in time. To reduce the adaptation cost, we suggest the developers use an actively maintained third-party parser instead of developing one themselves. This approach allows for efficient updates to support new language features without laborious custom development. Moreover, SCAT developers should pay more attention to the features that affect control or data flow analysis, as these features are more likely to compromise tool functionality and potentially lead to overlooked critical vulnerabilities.
- **To SCAT Users.** Our work shows that the unsupported features would greatly affect the ability of SCATs to detect vulnerabilities. To mitigate these risks, users should conduct comprehensive compatibility assessments before deploying SCATs, ensuring that the chosen tool aligns with the specific language features present in their codebase. To facilitate this process, we suggest SCAT users to reuse our test suite. In particular, we have provided detailed instructions on coding the automated scripts for various SCATs and release the source code at *10.6084/m9.figshare.25584585*, ensuring a streamlined and efficient approach for different testing scenarios. Moreover, our test suite is applicable to most PHP features. To include a new PHP feature, it is just required to prepare 2 new data assignment snippets (both feature-present and -absent).
- **To Researchers.** This study highlights the need for researchers to understand and address the challenges of SCATs posed by the continuous evolution of programming languages. Our research primarily focuses on features that explicitly introduce new tokens or syntax; however, it acknowledges that the impact of language evolution extends beyond this scope. The findings suggest a broader research agenda for advancing static code analysis. Future investigations could explore additional language features, such as examining how the introduction of new object-oriented programming features affects SCAT performance.
- **To Security Community.** Our study demonstrates that new

language features affect source code parsing and control/-data flow analysis during vulnerability detection processes in SCAT. Therefore, it motivates the development of a shared library that abstracts control and data flow into an intermediate form to mitigate the impact. The necessity for such a library is further emphasized by its current absence and the lack of awareness surrounding its potential benefits. This gap is evidenced by the observation that the five SCATs examined in this study employ four distinct code parsers.

VI. Threats to Validity

Internal Threats. Due to the intricate nature of the SCAT analysis process, SCAT's effectiveness is susceptible to various factors, which poses an internal threat to the validity of our impact assessment results (RQ2). To mitigate the threats, we employ diverse code structures, aiming to circumvent irrelevant factors and facilitate accurate impact assessment. Two experienced PHP researchers collaborate to explore potential code structures, including feature implementations and sink functions, thereby enhancing variability. Despite the great care taken by researchers, there is still the possibility of overlooking certain cases, which cannot be completely avoided.

External Threats. The representativeness of the selected features in our work may be a threat to external validity. To reduce the threat, we adopt a multifaceted approach. Firstly, we select features from five releases spanning five years, covering a wide period. Secondly, we scrutinize all 70 features introduced within the five releases. Thirdly, to ensure the accuracy of selection for features that have potential impacts, in addition to reading the documents, we parse many feature-present snippets into ASTs to ascertain a full understanding of the changes brought by a new feature. Furthermore, the impacts of selected features involve three steps of the SCAT analysis procedure, which is identical to the theoretical analysis in RQ3. This indicates that our findings are representative and general to the static vulnerability detectors of PHP.

VII. Related Work

To address vulnerabilities in PHP codes, many efforts have been put into developing practical SCATs. Our research does not introduce new techniques for discovering vulnerabilities; instead, it concentrates on the challenges faced by these tools.

On the one hand, PHP is a dynamically typed language, making it challenging for both programmers and tools to reason about programs, and researchers have conducted various investigations to explore which features deserve to be modeled so that developers can focus on these features. Hills *et al.* [21]–[23] undertake a series of investigations on PHP language features, including frequently used features and prevalent code patterns. They also point out that the frequency has evolved with time and PHP programmers and SCAT developers should be conscious of the evolution. On the other hand, researchers recently found that coding style can also affect the effectiveness of static analysis. Medeiros and Neves [43] published the first work assessing the effect of coding styles on the ability to detect vulnerabilities of SCATs (i.e., RIPS, WAP, and phpSAFE). Their findings indicate that a greater distance between the query source and the sink would introduce more false negatives. Additionally, Kassar *et al.* discovered that certain code patterns, with an average of more than 21 patterns per PHP application, can significantly hinder the effectiveness of SCATs in code analysis. Based on these works, our study further emphasizes the need for keeping pace with the programming language evolution by proving that SCAT's ability to detect taint-style vulnerability is significantly compromised by introducing new features and proposing valuable suggestions.

VIII. Conclusion

As PHP is a rapidly evolving language, existing SCATs may fail to support new PHP features in time, which would significantly undermine the ability of SCATs to detect taint-style vulnerabilities. This paper conducts a systematic study of new language features and their impact on the ability of SCATs to detect taint-style vulnerabilities in PHP codes. Specifically, we identify 25 new features that potentially compromise the SCATs' ability and prove their widely adoption in popular PHP applications. Through the construction of a test suite and evaluation of five open-source SCATs, we reveal that these tools' capabilities are significantly impaired by the new features, with an average of 10 unsupported features per SCAT. To mitigate the impact, we locate the root cause and develop efficient adaptation strategies, suggesting that developers should frequently model changes introduced by new features, especially for features with new data/function definitions or utilizations. Our study provides crucial insights and implications for various stakeholders in static code analysis, emphasizing the importance of recognizing and proactively addressing the potential effects of language evolution.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work is supported in part by National Natural Science Foundation of China (62172105, 62172104, 62102091, 62102093), and the Funding of Ministry of Industry and Information Technology of the People's Republic of China under Grant TC220H079. Yuan Zhang is supported in part by the Shanghai Rising-Star Program 21QA1400700 and the Shanghai Pilot Program for Basic Research-Fudan University 21TQ1400100 (21TQ012). Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems and Engineering Research Center of Cyber Security Auditing and Monitoring.

Appendix

Details of the targeted new features are presented in Table VIII, including ❶ the basic information of each feature, i.e., name, released version and a brief; ❷ the AST signatures used in RQ1.2 (see §II-B), i.e., AST nodes and structures; ❸ the number of feature implementations used in RQ2 (see §III-A) and we annotate how to implement in the footnotes.

TABLE VIII: Details of Targeted New Language Features.

#	Feature Name	Release	Brief	AST Node	AST structures	# of Impls.
1	*Class constant visibility*	7.1	It allows specifying the visibility of class constants using access modifiers such as `public`, `private`, or `protected`.	`AST_CLASS_CONST_GROUP`	It has a string of `public`, `private` or `protected` in the code line.	3*
2	`iterable` *pseudo-type*	7.1	It introduces a new type which accepts data that can be looped by `foreach`, such as an array.	`AST_TYPE` flag: `TYPE_ITERABLE`	/	2†
3	*Multi catch exception handling*	7.1	It allows the handling of multiple exceptions within a single catch block.	`AST_CATCH`	The `AST_NAME_LIST` of it has multiple `AST_NAME` nodes as children.	1
4	*Nullable type*	7.1	By prefixing a type name with a question mark, it allows the specified type to accept `null`.	`AST_NULLABLE_TYPE`	/	2†
5	*Support for keys in* `list()`	7.1	It allows to assign values from an array to individual variables using specific keys.	`AST_ARRAY` flag: `ARRAY_SYNTAX_LIST`	The `AST_ARRAY_ELEM` of it has two children: `AST_VAR` and a key string.	1
6	*Symmetric array destructuring*	7.1	It allows using shorthand `[]` for array destructuring, which is symmetric to array constructing.	`AST_ASSIGN`	It has two `AST_ARRAY` nodes with flag: `ARRAY_SYNTAX_SHORT` as children.	1
7	*Void function*	7.1	It allows to explicitly claim a function without return value as `void`.	`AST_TYPE` flag: `TYPE_VOID`	/	1
8	`object` *type*	7.2	It introduce a new type which accepts a class instance.	`AST_TYPE` flag: `TYPE_OBJECT`	/	2†
9	*Array Destructuring supports Reference Assignments*	7.3	It allows a variable to be assigned through reference in an array.	`AST_ARRAY_ELEM` flag: `BY_REFERENCE`	/	1
10	*Arrow functions*	7.4	It introduces a more concise syntax for traditional anonymous functions, with an arrow (`=>`) to separate the function's parameters from its body.	`AST_ARROW_FUNC`	/	1
11	*Null coalescing assignment operators*	7.4	It introduces a new assignment operator `??=` which assigns a value only if the variable is not null.	`AST_ASSIGN_OP` flag: `BINARY_COALESCE`	/	1
12	*Typed properties*	7.4	It allows class properties to declare type.	`AST_PROP_GROUP`	It has an `AST_TYPE` node as a child.	1
13	*Unpacking inside arrays*	7.4	By prefixing an array with `...`, it allows it to be unpacked in another array.	`AST_ARRAY`	It has an `AST_UNPACK` as a child.	1
14	*Arbitrary expression support for* `new` *and* `instanceof`	8.0	It allows `new` and `instanceof` to be used with arbitrary expressions, which is only used with a class name or a variable before.	`AST_NEW`/ `AST_INSTANCEOF`	Except for `AST_ARG_LIST`, it has a child neither `AST_VAR` nor `AST_NAME`.	2≀
15	*Class constant dereferencability*	8.0	It allows the left operand of `::` to be a constant.	`AST_CLASS_CONST`	It has an `AST_CLASS_CONST` as a child.	3‡
16	`::class` *on objects*	8.0	It allows a object variable to access its class name using `::class`.	`AST_CLASS_NAME`	It has an `AST_VAR` node as a child.	1
17	*Constructor property promotion*	8.0	It allows for the direct definition and initialization of class properties in the constructor method parameter list.	`AST_PARAM` flag: `PARAM_MODIFIER_PUBLIC`/ `PARAM_MODIFIER_PRIVATE`/ `PARAM_MODIFIER_PROTECTED`	/	3*
18	`match` *expression*	8.0	It introduces a new expression `match` which is similar to the `switch` statement, but with safer semantics.	`AST_MATCH`	/	1
19	`mixed` *type*	8.0	It introduces a new type which accepts all types of data.	`AST_TYPE` flag: `TYPE_MIXED`	/	2†
20	*Named argument*	8.0	It allows to pass arguments to a function by prefixing a value with the parameter name followed by a colon.	`AST_NAMED_ARG`	/	1
21	*Non-capturing catch*	8.0	It allows catching exceptions without capturing them to variables.	`AST_CATCH`	It has no `AST_VAR` as children.	1
22	*Nullsafe operator*	8.0	It introduces a new class member access operator `?->` which accesses the class member only if it is not null.	`AST_NULLABLE_PROP`/ `AST_NULLABLE_METHOD_CALL`	/	2°
23	*Static return type*	8.0	It allows a method to be claimed as `static`.	`AST_TYPE` flag: `TYPE_STATIC`	/	1
24	`throw` *expression*	8.0	It converts the `throw` statement into an expression	`AST_THROW`	Its direct ancestor is neither empty nor `AST_STAT_LIST`.	1
25	*Union type*	8.0	It introduces a new type which can combined different types by a pipe symbol.	`AST_TYPE_UNION`	/	2†

*: used with `public`, `protected` and `private`; †: used as a function return type or a parameter type; ‡: used with a constant or a static property or method;
≀: used with `new` or `instanceof`; °: used with a class property or method.

REFERENCES

[1] (2023) Usage statistics of server-side programming languages for websites. [Online]. Available: https://w3techs.com/technologies/overview/programming_language

[2] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in Java applications with static analysis," in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*. USA: USENIX Association, 2005, p. 18.

[3] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 317–331.

[4] Acunetix. (2021) Web vulnerability report. [Online]. Available: https://cdn.acunetix.com/wp_content/uploads/2021/04/Invicti-AppSec-Indicator-Spring-2021-Edition-Acunetix-Web-Vulnerability-Report.pdf

[5] Edgescan. (2022) Vulnerability statistics report. [Online]. Available: https://www.edgescan.com/january-2022-vulnerability-statistics-snapshot/

[6] P. Black, "Static analyzers: seat belts for your code," *IEEE Security & Privacy*, vol. 10, no. 3, pp. 48–52, 2012.

[7] C. Vassallo, S. Panichella, F. Palomba *et al.*, "Context is king: the developer perspective on the usage of static analysis tools," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 38–49.

[8] L. Do, J. R. Wright, and K. Ali, "Why do software developers use static analysis tools? a user-centered study of developer needs and motivations," *IEEE Transactions on Software Engineering*, vol. 48, no. 03, pp. 835–847, mar 2022.

[9] C. Luo, P. Li, and W. Meng, "Tchecker: precise static inter-procedural analysis for detecting taint-style vulnerabilities in PHP applications," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2175–2188.

[10] J. Dahse and T. Holz, "Simulation of built-in PHP features for precise static code analysis," in *Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2014*, 2014, pp. 23–26.

[11] J. Dahse and J. Schwenk, "RIPS – a static source code analyser for vulnerabilities in PHP scripts," in *Seminar Work (Seminer Çalismasi). Horst Görtz Institute Ruhr-University Bochum*. Citeseer, 2010.

[12] (2024) Symfony. [Online]. Available: https://symfony.com/

[13] (2023) Symfony security advisories. [Online]. Available: https://symfony.com/blog/category/security-advisories

[14] (2021) CVE-2021-32693. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-32693

[15] (2021) CVE-2021-41270. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-41270

[16] (2021) Authentication granted with multiple firewalls. [Online]. Available: https://github.com/symfony/security-http/commit/6bf4c31219773a558b019ee12e54572174ff8129

[17] (2021) Use single quote to escape formulas. [Online]. Available: https://github.com/symfony/symfony/commit/3da6f2d45e7536ccb2a26f52fbaf340917e208a8#diff-e3754e3175fa2f2830913e6d53759947ca8db58d3e7b1c1b61157997536e4e91R59

[18] (2024) Laravel. [Online]. Available: https://laravel.com/

[19] (2024) Drupal. [Online]. Available: https://www.drupal.org/

[20] (2024) Joomla. [Online]. Available: https://www.joomla.org/

[21] 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.

[22] M. Hills, "Variable feature usage patterns in PHP (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. IEEE Press, 2015, p. 563–573.

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

[24] (2023) PHP-AST. [Online]. Available: https://github.com/nikic/php-ast

[25] J. Liu, J. Zeng, X. Wang, and Z. Liang, "Learning graph-based code representations for source-level functional similarity detection," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 345–357.

[26] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, ser. MSR '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 1–5.

[27] (2023) Gitstar ranking. [Online]. Available: https://gitstar-ranking.com

[28] P. Li and W. Meng, "Lchecker: Detecting loose comparison bugs in php," in *Proceedings of The Web Conference 2021*, Apr. 2021.

[29] (2023) PHPStan. [Online]. Available: https://phpstan.org/

[30] R. Rabheru, H. Hanif, and S. Maffeis, "Deeptective: Detection of php vulnerabilities using hybrid graph neural networks," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, ser. SAC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1687–1690.

[31] (2023) Progpilot. [Online]. Available: https://github.com/designsecurity/progpilot

[32] (2023) RIPS - a static source code analyser for vulnerabilities in php scripts. [Online]. Available: https://rips-scanner.sourceforge.net/

[33] M. Backes, K. Rieck, M. Skoruppa *et al.*, "Efficient and flexible discovery of PHP application vulnerabilities," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017, pp. 334–349.

[34] (2023) PHPJoern. [Online]. Available: https://github.com/malteskoruppa/phpjoern

[35] I. Medeiros, N. Neves, and M. Correia, "Detecting and removing web application vulnerabilities with static analysis and data mining," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 54–69, 2016.

[36] ——, "Equipping WAP with WEAPONS to detect vulnerabilities: practical experience report," in *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 630–637.

[37] (2023) WAP. [Online]. Available: https://awap.sourceforge.net/news.html

[38] P. J. C. Nunes, J. Fonseca, and M. Vieira, "phpSAFE: a security analysis tool for OOP web application plugins," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp. 299–306.

[39] (2023) phpSAFE. [Online]. Available: https://github.com/JoseCarlosFonseca/phpSAFE

[40] T. Unruh, B. Shastry, M. Skoruppa, F. Maggi, K. Rieck, J.-P. Seifert, and F. Yamaguchi, "Leveraging flawed tutorials for seeding large-scaleweb vulnerability discovery," in *Proceedings of the 11th USENIX Conference on Offensive Technologies*, ser. WOOT'17. USA: USENIX Association, 2017, p. 5.

[41] F. Yamaguchi, N. Golde, D. Arp *et al.*, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 590–604.

[42] S. Wi, S. Woo, J. J. Whang *et al.*, "HiddenCPG: large-scale vulnerable clone detection using subgraph isomorphism of code property graphs," in *Proceedings of the ACM Web Conference 2022*. Association for Computing Machinery, 2022, p. 755–766.

[43] I. Medeiros and N. Neves, "Impact of coding styles on behaviours of static analysis tools for web applications," in *2020 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, 2020, pp. 55–56.