KAVE: A Tool to Detect XSS and SQLi Vulnerabilities using a Multi-Agent System over a Multi-Layer Knowledge Graph

Rafael Ramires[1,2]
Mike Papadakis
rframires@fc.ul.pt
michail.papadakis@uni.lu
[2]SnT, University of Luxembourg
Luxembourg

Ana Respício
Ibéria Medeiros
alrespicio@fc.ul.pt
ivmedeiros@fc.ul.pt
[1]LASIGE, DI, Faculdade de Ciências, Universidade de Lisboa, Portugal

ABSTRACT

Web applications have been widely adopted to access a myriad of services, regardless of their criticality and context. Applications developers have accelerated their efforts to meet the demands of a competitive and dynamic market for innovative products. Despite considerable efforts to detect and mitigate vulnerabilities in applications, their prevalence continues to increase, primarily due to the rapid pace of software development, which often prioritizes deployment speed, compromising security. This paper presents KAVE, a static analysis tool that leverages a multi-layer knowledge graph and a multi-agent system to detect web application vulnerabilities with high precision. This paper showcases KAVE's implementation and ability to identify SQL injection (SQLi) and cross-site scripting (XSS) vulnerabilities in real-world PHP applications.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Web Application Vulnerabilities; Static Analysis; Multi-Layer Knowledge Graph; Multi-Agent System; Software Security.

1 INTRODUCTION

The increasing reliance on online services has driven the exponential growth of web applications. However, this rapid expansion has also led to a surge in cyber attacks exploiting vulnerabilities in web applications, posing significant risks to users and organizations [4, 6]. Among the most prevalent and impactful vulnerabilities are SQL Injection (SQLi) and Cross-Site Scripting (XSS) [3, 9, 22], which can lead to unauthorized database access, data theft, and malicious script execution. These vulnerabilities are particularly common in PHP web applications due to their loosely typed nature and widespread use in web development.

Static application security testing (SAST) tools (commonly known as static analysis tools) have been seen as a proactive solution for identifying vulnerabilities during software development by analyzing the source code without executing it. This enables developers to address security flaws before deployment. However, existing SASTs often suffer from high false positive rates and limited precision, which hinder their practical adoption [19]. Additionally, many tools rely on code representations such as Abstract Syntax Trees (ASTs) [5, 15, 16, 18], which lack the comprehensive inter-procedural insights necessary for effective vulnerability detection.

To address these issues, we present KAVE (**K**nowledge-based **A**gent-system **V**ulnerability d**E**tector) [21], an innovative static analysis tool that combines advanced graph-based code representations with multi-agent systems to detect vulnerabilities in PHP web applications with high precision and efficacy. The key features of KAVE include: (1) A Multi-Layer Knowledge Graph (MLKG) that integrates diverse code representation graphs: Function Call Graphs (FCG), Control Flow Graphs (CFG), Dependence Variable Graphs (DVG) [24], and Program Dependence Graphs (PDG) [8, 25], enriched with security properties such as where entry points, sensitive sinks, and sanitization functions are present in the code. (2) A Multi-Agent System (MAS) that performs static taint analysis over the MLKG to effectively and efficiently navigate the graph and identify potential vulnerabilities. (3) A pruning method that strategically reduces irrelevant paths in the MLKG, optimizing computational efficiency while maintaining analysis accuracy.

This paper focuses on showcasing KAVE's vulnerability detection process, which outperforms traditional SASTs tools like WAP [15], Pixy [10], and PHPCorrector [16].

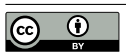
2 KAVE OVERVIEW

2.1 Key Features and Architecture

KAVE is a tool designed to detect vulnerabilities in PHP web applications. By combining advanced graph-based code structures that represent complex inter-procedural relationships with multi-agent systems (performing taint analysis), KAVE achieves low false-positive rates while performing inter-procedural analysis.

The architecture of KAVE consists of three main modules, as illustrated in Figure 1:

(1) **Code Property Graph Generator:** Parses PHP source code into different types of code property graphs, namely FCG, CFG, DVG, and PDG graphs [8, 24, 25].
(2) **Knowledge Graph Creator:** Constructs the MLKG by integrating the FCG, CFG, DVG, and PDG graphs, and then enrich it with security properties, specifying the entry points (e.g., user input like `$_GET`), sensitive sinks (e.g., `mysqli_query`), and sanitization (e.g., `htmlentities`). These security properties are added here to enhance the graph's utility in vulnerability detection.



This work is licensed under a Creative Commons Attribution 4.0 International License.
FSE Companion '25, June 23–28, 2025, Trondheim, Norway
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1276-0/2025/06.
https://doi.org/10.1145/3696630.3728601

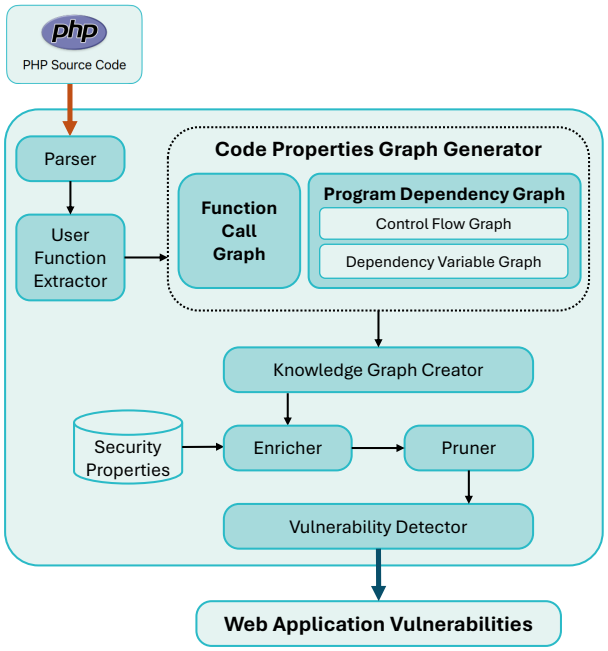


Figure 1: KAVE System Architecture.

(3) **Pruner:** Removes nodes from the MLKG that are unnecessary for the analysis (e.g., functions that do not have entry points as input arguments and contain sensitive sinks in their body).
(4) **Vulnerability Detector:** Utilizes the MAS to navigate the MLKG and identify vulnerabilities.

2.2 Multi-Layer Knowledge Graph (MLKG)

The MLKG [7, 21] serves as the core data structure for KAVE by integrating multiple code representation graphs to a unified one:

- The first layer consists of the FCG [2, 24], where nodes represent user functions and edges represent function calls.
- The second layer links each FCG node to its corresponding PDG [8, 25], which encapsulates both control flow (CFG) and data dependencies (DVG).
- The third layer aggregates security properties [21] from the PDGs into the FCG nodes, enabling efficient navigation and effective analysis.

One example of how the MLKG would look like can be found in Fig. 2. On the left side of the figure, we can observe the FCG with its functions as nodes, each pointing to the PDG that constitutes it. On the right side, a PDG is depicted, containing the control flows, data dependencies, and the interconnections between them. Security properties presented in PDG (in blue) are then aggregated in the respective function node (in blue).

The integration of these specific graphs is essential for achieving a comprehensive representation of the application's code. The FCG provides an overview of inter-procedural relationships, making it possible to track how data flows between functions. The PDG combines control flow and data dependencies within individual functions, offering a detailed view of intra-procedural behavior. By linking these layers, the MLKG ensures that both high-level and fine-grained information about the application is available.

One of the key advantages of this hierarchical structure [21] is its ability to focus computational resources on relevant parts of the code. For instance, if no evidence of potential vulnerabilities is found in the first layer (FCG), there is no need to traverse deeper into the internal layers (PDGs). This targeted approach reduces unnecessary computations and improves overall efficiency. Additionally, by isolating security properties such as entry points, sensitive sinks, and sanitization functions in a separate layer, the MLKG allows for precise pruning and prioritization during vulnerability detection.

Compared to traditional tools that analyze entire codebases exhaustively, KAVE's hierarchical approach leverages the MLKG to streamline analysis by narrowing down potential vulnerability paths early in the process. This design not only optimizes computational effort but also enhances scalability, making it suitable for large and complex web applications.

2.3 Multi-Agent System (MAS)

The MAS in KAVE [21] comprises specialized agents that operate autonomously within the MLKG:

- The **Travel Agent** Identifies entry points in the MLKG and initiates taint propagation.
- The **Verification Agent** validates potential vulnerabilities by analyzing their propagation paths.
- The **Translation Agent** examines how variable definitions change between functions.
- The **Flow Agent** tracks tainted variables across control flow paths within a function's PDG.
- The **Data Agent** examines data dependencies to determine whether tainted variables reach sensitive sinks.

These agents coordinate their efforts to efficiently traverse the MLKG and identify vulnerabilities avoiding the exhaustive exploration of irrelevant paths. Figure 3 presents the MAS with the interactions on the different layers of MLKG.

The Travel Agent identifies potential vulnerability paths in the FCG by locating entry points and sensitive sink pairs. It then communicates these paths to the Verification Agent, which validates the existence of data and control flow dependencies along the identified paths. For a deeper analysis, the Verification Agent delegates tasks to the Translation Agent, which examines how variables are transformed between functions in the PDG. Finally, the Flow Agent and Data Agent are activated to assess control flow and data dependencies within specific PDGs, ensuring that vulnerabilities are

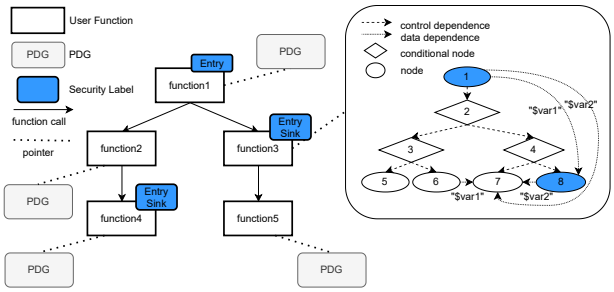


Figure 2: Multi-Layer Knowledge Graph Example.

confirmed only when all necessary conditions are met. This multi-agent collaboration ensures that only relevant parts of the MLKG are analyzed, reducing computational overhead.

2.4 Pruning Method

To optimize performance, KAVe [21] employs a pruning method that removes unnecessary nodes from the MLKG: *i*) Nodes without entry points, sensitive sinks, or sanitization functions are removed because they do not contribute to vulnerability detection. *ii*) Functions that are isolated or do not propagate tainted data are converted into connectors or trimmed entirely.

This approach significantly reduces the size of the MLKG while preserving all critical information required for accurate vulnerability detection. This is facilitated by the MLKG design, which separates the security properties of the code into a distinct layer, independent of its generation, as we can notice in Figure 4.

3 EXPERIMENTAL EVALUATION

3.1 Previous Results Summary

KAVe [21] has been evaluated in 12 open-source PHP web applications, where it detects 235 vulnerabilities with precision 95.9%. These results demonstrate the KAVe's ability to outperform traditional tools such as WAP [15], Pixy [10] and PHPCorrector [16] in terms of precision and false-positive rates. For instance, while Pixy flags a significant number of vulnerabilities, it suffers from a high false-positive rate, resulting in low precision. In contrast, WAP and PHPCorrector had fewer false positives but missed several vulnerabilities, which affected their recall. KAVe achieved a joint optimal result by combining high precision with low false positives and minimal loss of true positives (i.e., false negatives).

3.2 New Vulnerability Scenarios Evaluation

Although KAVe demonstrated strong performance in real-world web applications, testing in individual SARD code snippets [17] revealed areas for improvement [21]. Specifically, in certain edge cases for SQLi and XSS [11, 12, 23], vulnerabilities were not fully addressed by the tool. To further refine its capabilities, we conducted

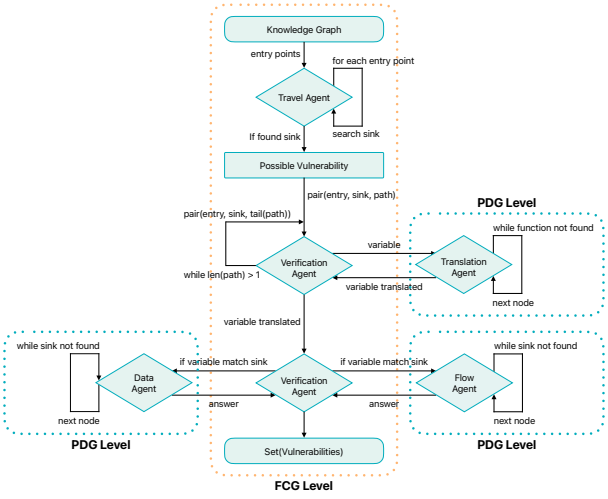


Figure 3: Multi-Agent System Architecture.

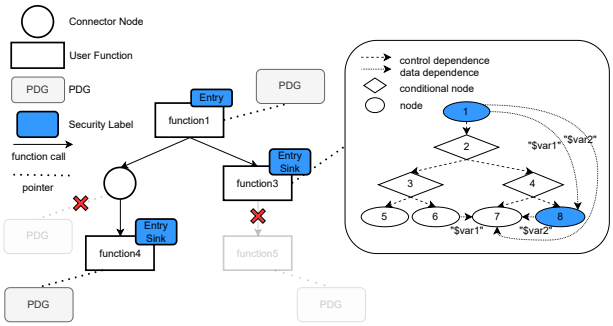


Figure 4: Prunning result for the MLKG in Figure 2.

an extended evaluation in which we focused on these isolated corner cases. This evaluation involved:

- Analysis of SQLi corner cases [11, 14], such as complex query concatenations and nested SQL statements.
- Analysis of XSS corner cases, including DOM-based XSS [13, 23] patterns and unconventional input/output flows.

This evaluation aimed to identify specific vulnerability scenarios that are not yet fully supported by KAVe, providing insights for future enhancements. The targeted analyses revealed that KAVe successfully detected 55.7% of SQLi corner cases and 91.4% of XSS corner cases in the test dataset. These results align with expectations: XSS vulnerabilities are generally simpler to identify through static analysis due to their well-defined patterns and reliance on specific entry points and sinks. In contrast, SQLi vulnerabilities often involve more complex data flows and sanitization scenarios, which can be harder to capture accurately. Tables 1 and 2 summarize the results of these focused tests.

For SQLi, KAVe performed well in certain test suites, such as *Multi Query* and *Real Query*, achieving a perfect detection rate of 100%. However, the tool struggled significantly in the *MySQLi Multi Query* and *MySQLi Real Query* test suites, where detection rates were only 7.3%. Upon further analysis, these failures were primarily due to KAVe incorrectly labelling some scenarios as unsafe when they were actually safe. This suggests that KAVe currently lacks the ability to recognize certain sanitization mechanisms specific to

Table 1: Detection Results for SQLi Corner Cases

Test Suite	Total Cases	Detected Cases	Accuracy (%)
Handmade Corner Cases	28	25	89.3
Multi Query	492	492	100.0
MySQL Query	39	31	79.5
MySQLi Multi Query	492	36	7.3
MySQLi Real Query	492	36	7.3
Real Query	492	492	100.0
Total	**1996**	**1112**	**55.7**

Table 2: Detection Results for XSS Corner Cases

Test Suite	Total Cases	Detected Cases	Accuracy (%)
DOM-based XSS	20	18	90.0
Reflected XSS	15	14	93.3
Stored XSS	15	14	93.3
Unconventional Flows	10	9	90.0
Complex Input/Output Flows	10	9	90.0
Total	**70**	**64**	**91.4**

MySQLi queries. Addressing this limitation will require extending KAVe's knowledge base to include additional sanitization functions and improving its handling of complex sanitization scenarios.

For XSS vulnerabilities, KAVe demonstrated strong performance across all tested scenarios, with detection rates exceeding 90% for each test suite. The high accuracy in identifying DOM-based XSS, reflected XSS, and stored XSS vulnerabilities highlights KAVe's ability to effectively track tainted data from entry points to sensitive sinks in a variety of contexts.

3.3 Efficiency Analysis

The runtime performance during the extended evaluation remained consistent with previous results. Consistently, KAVe analyzed individual code snippets in under 0.1 seconds and previously larger applications ranging from 270 to 7704 lines of code in times concise between 1 and 8 seconds for approximately. This efficiency is attributed to the pruning method applied to the MLKG, which reduces irrelevant paths and optimizes computational effort.

3.4 Runtime Example

To illustrate KAVe's functionality, consider the PHP code snippet of Listing 1 that contains SQLi and XSS vulnerabilities. The code accepts user input from the `id` parameter via a `$_GET` request and directly incorporates it into an SQL query without proper sanitization or validation, also ending up echoing it. Also, if the records returned from the database contain malicious data in the `username` field, when this is used in the `echo` sensitive sink without sanitization, an XSS vulnerability is triggered. This lack of input handling makes the application vulnerable to malicious attacks.

Listing 1: A PHP script with an XSS and SQLi vulnerabilities.

```php
// Retrieve the 'id' parameter from the GET request
$id = $_GET['id'];

// Construct an SQL query without sanitizing the input
$query = "SELECT * FROM users WHERE id = $id";

// Execute the query
$result = mysqli_query($connection, $query);

// Fetch and display results
while ($row = mysqli_fetch_assoc($result))
    echo "User: " . $row['username'];
```

If a malicious user provides input such as `1 OR 1=1`, the query becomes: `SELECT * FROM users WHERE id = 1 OR 1=1;` This allows attackers to retrieve all records from the `users` table, bypassing authentication mechanisms.

Listing 2: Output of KAVe over the code of Listing 1.

```
Number of vulnerabilities: 2

Total vulnerabilities found:
XSS: 1
SQLi: 1

Graph stats:
N graphs: 8
N functions: 1
N variables: 5
N nodes: 8
N edges: 7
Elapsed time: 0.09 seconds
```

Executing KAVe with this code snippet, saved as `file1.php`, and using the following command: `python3 main.py /file1.php`, the output generated by the tool provides detailed information about detected vulnerabilities and graph statistics (Listing 2).

4 CONCLUSION

This paper presented KAVe, a static analysis tool that combines a MLKG with a MAS to detect vulnerabilities in web applications with high precision and efficiency. By integrating diverse code representations such as FCG, CFG, DVG, and PDG, KAVe provides a comprehensive framework for inter-procedural vulnerability detection. The pruning method improves the system's efficiency by discarding irrelevant paths, while the MAS performs effective static taint analysis to identify SQLi and XSS vulnerabilities.

4.1 Availability

KAVe is an open-source tool designed to be accessible to developers and researchers alike. The tool, along with its documentation, sample datasets, and installation instructions, is available on GitHub [20] at [https://github.com/rframires/KAVe], and includes: (i) The source code repository containing all necessary files for installation and usage. (ii) A step-by-step user guide detailing how to analyze PHP applications using KAVe. (iii) Sample vulnerable PHP applications and code snippets for testing and experimentation.

The tool requires Python 3 as the runtime environment and depends on two Python libraries, namely `matplotlib` for visualizations and `networkx` [1] for graph processing. To deploy KAVe, users can clone the repository, install the dependencies using `pip`, and run the tool directly on their PHP source code. Detailed instructions and example applications are provided in the repository to help users get started with analyzing vulnerabilities.

4.2 Limitations & Future Work

While KAVe demonstrates significant improvements in precision and efficiency for detecting vulnerabilities in PHP web applications, it has limitations. Firstly, KAVe is specifically designed for PHP applications and does not currently support other programming languages. This limits its applicability to a broader range of web applications written in other languages, such as JavaScript or Python. Secondly, the reliance on static analysis means that KAVe cannot detect vulnerabilities that depend on runtime behavior or dynamic inputs, which are often necessary for identifying certain types of security flaws. Additionally, the MAS relies on predefined rules and heuristics, which may not adapt well to novel or evolving vulnerability patterns without manual updates.

As future work, we aim to address these limitations. Also, we aim to explore artificial intelligence techniques, such as Large Language Models, to further reduce the false positives and improve vulnerability detection accuracy.

ACKNOWLEDGMENTS

This work was partially supported by P2030 through project I2DT, ref. COMPETE2030-FEDER-00389100, an ITEA4 European project (ref. 22025), and by FCT through the LASIGE Research Unit, ref. UIDB/00408/2025-LASIGE.

REFERENCES

[1] Pieter J. Swart Aric A. Hagberg, Daniel A. Schult. 2008. Exploring network structure, dynamics, and function using NetworkX. https://networkx.org/

[2] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *Proceedings of the 2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. 334–349.

[3] The MITRE Corporation. 2023. Common Weakness Enumeration. https://cwe.mitre.org/.

[4] CVE. 2023. CVE Details. https://www.cvedetails.com/browse-by-date.php.

[5] Johannes Dahse and Thorsten Holz. 2014. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *Proceedings of the Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February*. The Internet Society. https://www.ndss-symposium.org/ndss2014/simulation-built-php-features-precise-static-code-analysis

[6] John Daley. 2017. Insecure Software is Eating the World: Promoting Cybersecurity in an Age of Ubiquitous Software Embedded Systems. *Stanford Technology Law Review* 19, 3 (2017).

[7] Manlio De Domenico, Albert Solé-Ribalta, Emanuele Cozzo, Mikko Kivelä, Yamir Moreno, Mason A Porter, Sergio Gómez, and Alex Arenas. 2013. Mathematical formulation of multilayer networks. *Physical Review X* 3, 4 (2013), 041022.

[8] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (jul 1987), 319–349. https://doi.org/10.1145/24039.24041

[9] OWASP Foundation. 2023. Open Web Application Security Project. https://www.owasp.org/.

[10] N. Jovanovic, C. Kruegel, and E. Kirda. 2006. Pixy: a static analysis tool for detecting Web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy*. 6 pp.–263. https://doi.org/10.1109/SP.2006.29

[11] Kingthorin. 2023. SQL Injection. https://owasp.org/www-community/attacks/SQL_Injection.

[12] KirstenS. 2023. Cross Site Scripting (XSS). https://owasp.org/www-community/attacks/xss/.

[13] Amit Klein. 2005. DOM Based Cross Site Scripting or XSS of the Third Kind. http://www.webappsec.org/projects/articles/071105.shtml.

[14] Stephen Kost. 2007. An Introduction to SQL Injection Attacks for Oracle Developers.

[15] Ibéria Medeiros, Nuno Neves, and Miguel Correia. 2016. Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining. *IEEE Transactions on Reliability* 65, 1 (2016), 54–69. https://doi.org/10.1109/TR.2015.2457411

[16] Ricardo Morgado, Ibéria Medeiros, and Nuno Neves. 2020. Towards Web Application Security by Automated Code Correction. In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering*. 86–96.

[17] National Institute of Standards and Technology (NIST). 2023. NIST Software Assurance Reference Dataset (SARD). https://samate.nist.gov/SARD Accessed December 15, 2023.

[18] Paulo Nunes, José Fonseca, and Marco Vieira. 2015. phpSAFE: A Security Analysis Tool for OOP Web Application Plugins. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

[19] Tosin Daniel Oyetoyan, Bisera Milosheska, Mari Grini, and Daniela Soares Cruzes. 2018. Myths and Facts About Static Application Security Testing Tools: An Action Research at Telenor Digital. In *Agile Processes in Software Engineering and Extreme Programming - 19th International Conference, XP 2018, Porto, Portugal, May, Proceedings (Lecture Notes in Business Information Processing, Vol. 314)*, Juan Garbajosa, Xiaofeng Wang, and Ademar Aguiar (Eds.). Springer, 86–103. https://doi.org/10.1007/978-3-319-91602-6_6

[20] Rafael Ramires. 2024. KAVe: Knowledge-Based Multi-Agent System Vulnerability Detector. https://github.com/rframires/KAVe.git.

[21] Rafael Ramires, Ana Respício, and Ibéria Medeiros. 2024. KAVE: A Knowledge-Based Multi-Agent System for Web Vulnerability Detection. In *2024 IEEE International Conference on Web Services (ICWS)*. 489–500. https://doi.org/10.1109/ICWS62655.2024.00070

[22] Veracode. 2023. State of Software Security 2023. Annual Report on the State of Application Security. https://info.veracode.com/rs/790-ZKW-291/images/Veracode_State_of_Software_Security_2023.pdf.

[23] Dave Wichers, Arshan Dabirsiaghi, Stefano Di Paolo, Mario Heiderich, Eduardo Alberto Vela Nava, and Jeff Williams. 2023. Types of XSS. https://owasp.org/www-community/Types_of_Cross-Site_Scripting.

[24] Merijn Wijngaard. 2016. Dependence Analysis in PHP. http://www.scriptiesonline.uba.uva.nl/618176

[25] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. 590–604.