UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Detect Web Vulnerabilities Using Knowledge Graphs

Rafael Francisco Rosa Mesquita Ramires

Mestrado em Engenharia Informática

Dissertação orientada por:
Profª Doutora Ibéria Vitória de Sousa Medeiros
Profª Doutora Ana Luísa do Carmo Correia Respício

2023

Acknowledgments

I would like to express my deep gratitude to my supervisors, Prof[a] Ibéria Medeiros and Prof[a] Ana Respício, for their unwavering support, invaluable guidance, and endless patience throughout the entire process of researching and writing this thesis. Their insight and expertise have been instrumental in shaping my work, and I am truly fortunate to have had the opportunity to learn from them.

I would also like to thank my family, especially my mother Margarida and my father Francisco, I wouldn't be where I am today without their love and guidance. Also, to my brother Renato for bothering me but also supporting me every time I needed it. A mention as well to my aunts Gracinha and Bela, my godfather Fabrício and my grandfather Chico. Their tireless belief in me and my abilities has been a driving force in my pursuit of this degree.

I am incredibly grateful for the understanding, and support of my girlfriend Madalena, who has been my rock throughout this journey.

Additionally, I would like to express my sincere appreciation to FCUL colleagues Afonso, Bernardo, Carlota, Carolina, João, Lívio, Pedro, and many others, who have been a constant source of encouragement and support throughout my academic journey. Above all, their unwavering friendship and support have made this journey not only bearable but enjoyable. I am grateful for their presence in my life, and I hope to continue to count on their friendship in the years to come. I can not forget the ones that were here before and keep being so important in my life, a mention to them Afonso, Bruno, Serrano, Toste, Esteves, Zé, Amaro, Rita, and Sofia.

Finally, to my international friends that were a big part of my masters through my Erasmus period in Luxembourg, Erik, Giulia, Emi, Stamatia, also Piotr and Aleksandra, they helped me expand my horizons and taught me a new way of living.

This work was partially supported by the national funds through P2020 with reference to the XIVT project (LISBOA-01-0247, FEDER-039238, an ITEA3 European project (I3C4-17039)), and through FCT with reference to LASIGE Research Unit (UIDB/00408/2020 and UIDP/00408/2020).

Resumo

Nos últimos anos, o uso de aplicações web tem crescido de forma exponencial. A principal razão para isso acontecer é a expansão da Internet e o fácil acesso a estas por diferentes a dispositivos, tais como computadores, smartphones ou tablets, que são as principais fontes de ligação às aplicações web. Dada esta facilidade de acesso, o risco de invasão é maior, sendo que um maior número de aplicações na rede representa um risco mais significativo para a realização de ataques e exploração bem-sucedida de vulnerabilidades. Apesar da existência de vulnerabilidades e das suas classes já serem conhecidas à muito, este tema continua aberto para muita discussão, principalmente devido à falta de conhecimento sobre como detetar essas vulnerabilidades no código e como corrigi-las de maneira eficiente. O crescente risco para os utilizadores e a possibilidade de melhorar a eficácia na deteção de vulnerabilidades foram as principais razões que nos levaram a abordar o tema nesta dissertação, enfatizando a necessidade e importância de produzir aplicações mais seguras.

Injeções de SQL (*SQL injections* / SQLi) e cross-site scripting (XSS) são duas das vulnerabilidades mais comuns e perigosas em aplicações web e é nelas que nos focamos. Uma invasão bem-sucedida duma aplicação que contenha este tipo de vulnerabilidades pode causar danos tremendos aos utilizadores e empresas.

Uma das técnicas mais utilizadas para detetar vulnerabilidades em aplicações web é a análise estática de código. Esta técnica possibilita a análise de todo o código de uma aplicação sem executar. Esta técnica é benéfica uma vez que o código pode ser corrigido antes da aplicação ser terminada e colocada no mercado, mas ao mesmo tempo constitui uma tarefa complexa.

Detetar vulnerabilidades em aplicações web é de extrema importância por forma a prevenir ataques sobre as mesmas e representa um grande desafio da atualidade. Uma das maneiras mais promissoras de abordar este problema é por meio do uso de grafos de propriedades de código (como árvores de sintaxe abstrata (*abstract sintaxe tree* / AST), gráficos de controlo de fluxo (*control flow graphs* / CFG), gráficos de dependência de variáveis (*dependence variable graphs* / DVG), etc.) que representam características e propriedades do código de uma maneira que pode ser analisada estaticamente.

Embora cada uma delas represente um aspeto estrutural do código, estas apresentam limitações em alguns pontos essenciais para uma análise completa e eficiente do

programa. Por exemplo, as ASTs contêm todas as informações do programa, mas devido à sua dimensão, pesquisar nelas é muito desafiante, consome muito tempo e requer muita capacidade de computação. Já os CFGs dão acesso a dependências de fluxo que mostram os caminhos por onde a informação pode fluir no programa, mas não incluem dependências de variáveis que são imprescindíveis para a deteção de vulnerabilidades. Os DVGs são o oposto dos de fluxo, pois só têm dependências de dados que nos permitem saber quais variáveis dependem de outras, mas não há dependências de fluxo. O gráfico de dependência de programa (*program dependence graph* / PDG) é mais completo em relação aos dois anteriores, incluindo tanto fluxo quanto dependências de dados. Mas, ainda assim, tal como as duas estruturas anteriores, só permite realizar análise intra-procedural, que não é ideal para aplicações web, dado que estas são compostas por várias funções. Os grafos de chamadas de funções (*function call graphs* / FCG) são adequados para análise inter-procedural, mas não incluem nenhum tipo de dependência, ou seja, não teríamos informação suficiente para detetar vulnerabilidades.

Esta dissertação apresenta uma nova abordagem de análise estática para detetar vulnerabilidades em aplicações web escritas em PHP, através da ferramenta *Knowledge-based Agent-system Vulnerability detector* (KAVe). Este sistema visa melhorar a deteção de vulnerabilidade, pelo uso de grafos de conhecimento que incorporam a combinação de vários grafos de propriedades de código, o grafo de conhecimento é navegado por um sistema multiagente que executa análise de comprometimento para identificar eficientemente as possíveis vulnerabilidades na aplicação. Podemos considerar como principal objetivo o desenvolvimento da ferramenta KAVe, sendo esta capaz de detetar vulnerabilidades em aplicações web PHP de forma robusta e eficaz. A ideia é que esta ferramenta possa ajudar tanto programadores quanto profissionais de segurança a melhorar a segurança das suas aplicações web e proteger os usuários de possíveis ataques. Para alcançar este objetivo, é possível dividi-lo em sub-objetivos que precisam de ser alcançados. Primeiramente, é necessário analisar e entender o código da aplicação web, identificando os seus componentes e dependências, tais como chamadas de funções, variáveis e controlo de fluxo. Em seguida, deve-se construir CFGs e DVGs e combiná-los em PDGs, de maneira a estes que representem as todas as dependências duma função numa estrutura que possa ser analisada estaticamente. Os PDGs serão posteriormente combinados com um FCG num grafo de conhecimento (*knowledge graph* / KG) permitindo assim que a análise passe a ser realizada entre funções (inter-procedural). É importante que o KG seja bem organizado para facilitar a navegação e permitir uma deteção mais eficiente das vulnerabilidades. Para isso, é necessário também podar o KG para reduzir o número de caminhos irrelevantes ou redundantes que não contribuem para a deteção de vulnerabilidades.

A navegação pelo KG é realizada por meio de um sistema multiagente, que deve ser capaz de identificar as potenciais vulnerabilidades de forma eficaz. Por fim, as vulnerabilidades sinalizadas devem ser validadas para verificar se representam vulnerabilidades

reais, assim como para contabilizar o número de falsos positivos. Ao completar estes objetivos estabelecemos contribuições importantes, tais como: um estudo aprofundado sobre diferentes tipos de representações de código e como as utilizar para identificar vulnerabilidades; uma nova abordagem para um modelo de KG que combina as melhores características de diferentes grafos de representação de código, utilizando as informações mais relevantes de cada uma para detetar vulnerabilidades, além de incluir informações adicionais que geralmente não são apresentadas em grafos de propriedades de código, mas são relevantes para fins de eficiência da análise; um método de poda para remover informações irrelevantes do KG, tornando-o mais gerenciável e eficiente para análise; um sistema multiagente capaz de percorrer o KG de forma eficiente e identificar possíveis vulnerabilidades; e a ferramenta KAVe, que utiliza o modelo de KG combinado com o método de poda e o sistema multiagente para identificar vulnerabilidades.

Além disso, a dissertação inclui uma avaliação completa e validação da ferramenta, que envolveu a seleção de aplicações web de teste, comparação de desempenho com outras ferramentas existentes e ainda validação das SQLI e XSS identificadas. Estas contribuições são valiosas para a área de segurança de software, pois a deteção de vulnerabilidades é essencial para proteger sistemas e usuários contra ataques maliciosos. Com o modelo de KG combinado com o método de poda e o sistema multiagente propostos, é possível melhorar significativamente a eficiência e a precisão na deteção de vulnerabilidades em código fonte, tornando a ferramenta KAVe uma importante adição para a deteção de vulnerabilidades em software.

Os resultados demonstram que o KAVe fornece um método mais eficaz e eficiente para detetar vulnerabilidades em aplicações web PHP, contribuindo para o campo da segurança web e oferecendo uma ferramenta valiosa para desenvolvedores e profissionais da área. A ferramenta encontrou um total de 169 vulnerabilidades em 12 aplicações web de código aberto com apenas 2 falsos positivos, tendo então uma precisão de 98,81%. Quando comparado diretamente com as ferramentas WAP [1] e Pixy (das ferramentas mais conceituadas da área), em 5 aplicações web o KAVe foi capaz de detetar 70 vulnerabilidades reais (RV) e teve apenas 1 falso positivo (FP), estimando uma precisão de 98,59%; por sua vez WAP, detetou 19 vulnerabilidades reais e teve 21 falsos positivos, terminando com 47,5% de precisão; por fim, o Pixy conseguiu detetar apenas 11 reais vulnerabilidades e teve 38 falsos positivos, terminando com 26,8% de precisão.

Estes resultados são bastante promissores, demonstram que a nossa metodologia funciona e estabelece uma base para novos trabalhos que queiram tirar proveito da integração grafos de conhecimento e sistemas multiagente para detetar vulnerabilidades, sendo que estas áreas do conhecimento nunca tinham sido antes combinadas.

Palavras-chave: vulnerabilidades em aplicações web, análise estática, grafos de propriedades de funções, grafos de conhecimento, sistemas multiagente

Abstract

The increasing reliance on the web for various applications has led to an increase in the number of web-based attacks and vulnerabilities. When exploited, such vulnerabilities as Cross-site Scripting (XSS) and SQL injection (SQLi) can cause severe damage to companies, such as theft of vast amounts of user credentials and access to undue data. One of the most used methods to detect web vulnerabilities is static analysis, which analyzes all application code without running it, which is beneficial so the code can be corrected prior to execution, but at the same time a complex task.

This dissertation presents a novel approach for detecting vulnerabilities in PHP web applications by developing a knowledge-based agent-system vulnerability detector (KAVe). The system aims to improve upon existing vulnerability detection tools by incorporating knowledge graphs generated by combining the most important part of multiple code property graphs to be then navigated by a multi-agent system that will perform taint analysis to efficiently identify potential security weaknesses. The study objectives include code parsing and analysis, graph construction, knowledge graph creation, graph pruning, multi-agent navigation, vulnerability detection, validation, and comparison with existing tools.

The results demonstrate that KAVe provides a more effective and efficient method for detecting vulnerabilities in PHP web applications, contributing to the web security field and offering a valuable tool for developers and security professionals. The tool found 169 vulnerabilities over 12 open-source web applications, with a precision of 98.81%.

Keywords: web application vulnerabilities, static analysis, code property graphs, knowledge graph, multi-agent system

Contents

List of Figures **15**

List of Tables **17**

List of Algorithms **19**

List of Acronyms **21**

1 Introduction **1**
1.1 Motivation 1
1.2 Objectives 3
1.3 Contributions 4
1.4 Document Structure 4

2 Background and Related Work **7**
2.1 Vulnerabilities in Web Applications 7
2.2 Static Analysis 9
2.3 Taint Analysis 10
2.4 Types of Property Graphs 10
2.4.1 Abstract Syntax Trees (AST) 11
2.4.2 Function Call Graph (FCG) 12
2.4.3 Dependence Variable Graph (DVG) 13
2.4.4 Control Flow Graphs (CFG) 14
2.4.5 Program Dependence Graph (PDG) 15
2.5 Knowledge Graph (KG) 16
2.6 Multi-agents intelligent systems 16
2.6.1 Intelligent Agents 16
2.6.2 Multi-Agents Systems 18
2.7 Related Work 18
2.7.1 Use of Static Analysis in Vulnerability Detection 18
2.7.2 Use of Graphs in Vulnerability Detection 20

3 Knowledge-based Agent-system Vulnerability-detector (KAVe) 25
3.1 Challenges . . . 25
3.1.1 Dealing with the complexity of the web applications source code . 25
3.1.2 Integration of already existing tools into our solution . . . 26
3.1.3 Converting the source code statically into the property graphs . . 26
3.1.4 Combining different property graphs into one extended knowledge graph . . . 27
3.1.5 Implementing pruning over the knowledge graph without losing essential information . . . 28
3.1.6 Leveraging a multi-agent system to identify the vulnerabilities . . 28
3.2 Architecture . . . 29
3.2.1 Dependence Variable Graph . . . 30
3.2.2 Control Flow Graph . . . 32
3.2.3 Program Dependency Graph . . . 33
3.2.4 Function Call Graph . . . 35
3.2.5 Knowledge Graph . . . 35
3.3 Pruning . . . 37
3.4 Vulnerability Detection . . . 40
3.4.1 Travel Agent . . . 41
3.4.2 Verification Agent . . . 42
3.4.3 Translation Agent . . . 42
3.4.4 Flow & Data Agents . . . 43

4 Implementation 45
4.1 Tools and Packages . . . 45
4.1.1 Function Call Graph . . . 45
4.1.2 NetworkX . . . 46
4.2 Property Graphs Algorithms . . . 46
4.2.1 Dependence Variable Graph . . . 46
4.2.2 Control Flow Graph . . . 49
4.2.3 Program Dependency Graph . . . 51
4.3 Knowledge Graph Assemble . . . 52
4.4 Pruning . . . 53
4.5 Multi-Agent System . . . 54
4.5.1 Travel Agent . . . 55
4.5.2 Verification Agent . . . 56
4.5.3 Translation Agent . . . 57
4.5.4 Flow Agent . . . 58
4.5.5 Data Agent . . . 59

5 Evaluation and Validation **63**
5.1 Evaluation Setup 64
5.2 Web Applications Data Characterization 65
5.3 Vulnerability Detection with KAVe 66
5.4 Performance Comparison 68
5.4.1 First Phase Comparison 68
5.4.2 Second Phase Comparison 69

6 Conclusion **73**
6.1 Future Work 74

Bibliography **81**

List of Figures

2.1 SQL-injection vulnerability in a PHP script [1]. 8
2.2 A PHP script containing an XSS vulnerability 8
2.3 Base PHP code for the graph generation. 11
2.4 Abstract syntax tree of the code listed in Figure 2.1 (excluding line 1 for simplicity). 12
2.5 Function Call Graph for the code displayed in Figure 2.3. 13
2.6 Dependence variable graph for the code displayed in Figure 2.3, relative to the variable `$fee`. 13
2.7 Control Flow Graph for the code displayed in Figure 2.3. 14
2.8 Program Dependence Graph for the code displayed in Figure 2.3, from line 5 to 15. 15
2.9 Agents interaction with the Environment 17

3.1 Architecture of the proposed solution. 30
3.2 Enriched Dependency Variable Graph of the variable *$fee* in the code of Fig. 2.3 31
3.3 Enriched Control Flow Graph example from the code of Fig. 2.3 between line 5 and 15. 33
3.4 Enriched Program Dependency Graph example. 35
3.5 Example of the Knowledge Graph from the code of Figure 2.3. 36
3.6 Pruning example for Fig. 2.3 39
3.7 Multi-Agent System 40

List of Tables

3.1 Table of decisions for Pruning. 38

5.1 Chosen Web Applications. 65
5.2 Knowledge graph data characterization. 66
5.3 Analysis results. 67
5.4 Comparison between KAVe, WAP and Pixy. 68
5.5 Comparison between KAVe and WAP. 69

List of Algorithms

1 Dependence Variable Graph Main Algorithm 47
2 Control Flow Graph Main Algorithm 50
3 Program Dependency Graph Main Algorithm 51
4 Knowledge Graph Main Algorithm 53
5 Multi-Agent System Main Algorithm 54
6 Travel Agent Main Algorithm 55
7 Verification Agent Main Algorithm 56
8 Translation Agent Main Algorithm 57
9 Flow Agent Main Algorithm 58
10 Data Agent Main Algorithm 59

List of Acronyms

AST Abstract Syntax Tree

CFG Control Flow Graph

DVG Dependence Variable Graph

FCG Function Call Graph

FP False Positive

KAVe Knowledge-based Agent-system Vulnerability-detector

KG Knowledge Graph

MAS Multi-Agent System

PDG Program Dependence Graph

RV Real Vulnerability

SQLi SQL Injection

WAP Web Application Protection

XSS Cross-site Scripting

Chapter 1

Introduction

The usage of web applications has impressively been growing in the past few years. The main reason for this is the expansion of the world wide web and easier access to devices like computers, smartphones, or tablets, which are the major gateways to web applications. Due to this facility, the risk of invasions and attacks is more significant, as a larger number of applications on the network represent a more prominent risk for successful vulnerability exploitation.

There are a lot of known vulnerabilities, but this subject is still open for much discussion, primarily because of the lack of knowledge about detecting these vulnerabilities in code and how to resolve them efficiently. The growing risk to users and the availability to improve the topic efficiency were the main reasons for choosing it, emphasizing the necessity and importance of producing safer applications.

SQL injections (SQLi) and Cross-site scripting (XSS) are two of the most common and dangerous vulnerabilities in web applications. A successful invasion of an application containing such vulnerabilities can bring tremendous damage to users and companies, as the severity of this attack will depend predominantly on the attacker's experience and the severity of the vulnerability.

1.1 Motivation

One of the most prominent ways to tackle the vulnerability detection problem is by means of using code property graphs (e.g., abstract syntax tree (AST), control flow graph (CFG), program dependence graph (PDG)) that would represent code information in a way that it can be analyzed statically and doing so avoiding that the code is stripped of these vulnerabilities even before the application goes live where it can be attacked. Although there are already structures to do this representation, they all somehow lack some key points essential for a complete and efficient program analysis.

To be more specific, in the case of the ASTs, they contain all the information of the program, but due to their dimension, searching for them is very challenging, time-

consuming, and computer power expensive; in CFGs, we have access to flow dependencies that show us the routes where information can flow in the program but does not include data dependencies; in dependence variable graph (DVG), we got the opposite of CFGs since we only have data dependencies that allow us to know which variables depend on others but not flow dependencies; PDG improved from CFGs and DVGs by including both flow and data dependencies but still like the two previous structures mentioned, they can only perform intra-procedural analysis and so are not optimal for web applications where we have multiple inter-procedures; function call graphs (FCGs) are fit for inter-procedural analysis but do not include any kind of dependency so we would not have enough information to detect vulnerabilities.

From the previous discussion is intuitive to say that an ideal solution would be having the best of all structures. There are several motivations for developing such a solution for PHP web application vulnerability detection that aims to improve aspects of already existing ones by taking a different approach:

- Increasing demand for web application security: As more and more businesses and organizations rely on web applications to conduct their operations, the need for robust web application security tools has increased.
- PHP is widely used: PHP is one of the most widely used programming languages for web development, making it a prime target for attackers. A better vulnerability detection tool for PHP would help to secure a large number of websites and applications.
- Evolving threats: Web application vulnerabilities and attacks are constantly evolving, so there is a need for tools that can keep up with these changes and provide effective protection.
- Improving efficiency: Existing vulnerability detection tools may not be able to fully detect all vulnerabilities or may generate a high number of false positives. A better PHP vulnerability detection tool would be more efficient, reducing the workload for security professionals.
- Innovative approach: Developing a PHP web application vulnerability detection tool that uses a novel approach or incorporates new technologies could provide more accurate and comprehensive security assessments. This could help to identify previously unknown vulnerabilities and provide a more effective defence against attacks.
- Research and development: Developing a new vulnerability detection tool can also be a research and development opportunity to improve our understanding of web security and to develop new techniques for identifying vulnerabilities.

In this thesis, we present an approach KAVe (Knowledge-based Agent-system Vulnerability-detector), a static analysis tool for detecting vulnerabilities in web applications written in PHP. The tool utilizes a combination of control flow graphs, data flow graphs, and property graphs to construct a knowledge graph (KG) of the application, which is then traversed by a multi-agent system to identify potential vulnerabilities. In this thesis, we give an in-depth explanation of the implementation and design of the tool, including the techniques used for parsing and analyzing the code, as well as the methods used for pruning and navigating the KG. The evaluation and validation of the tool are also discussed, including the selection of test web applications, the performance comparison with other existing tools, and the validation of the identified vulnerabilities. Overall, we demonstrate that KAVe is a robust and effective tool for detecting vulnerabilities in PHP web applications, particularly for SQL injections and Cross-site scripting.

1.2 Objectives

The main objective of this thesis is to construct the KAVe tool to provide a robust and effective way to detect vulnerabilities in PHP web applications that can help developers and security professionals to improve the security of their web applications and protect their users from potential attacks. To achieve this, we can divide the main objective into sub-objectives that must be met:

- Code parsing and analysis: parse and analyze the code of the web application in order to identify the various components and dependencies. This includes extracting information from the code, such as function calls, variable dependencies, and control flow.
- Graph construction: construct property graphs (e.g., control flow graphs, data flow graphs, and program dependence graphs) that represent the code information in a way that can be analyzed statically.
- Knowledge graph creation: merge the property graphs into a knowledge graph (KG) that captures the inter- and intra-procedural dependencies of the code.
- Pruning: The tool must be able to prune the KG to reduce the number of irrelevant or redundant paths that do not contribute to vulnerability detection.
- Multi-agent system for vulnerability detection: navigate the KG efficiently using a multi-agent system to identify potential vulnerabilities.
- Validation: validate the identified vulnerabilities to verify their existence and reduce the number of false positives.

1.3 Contributions

The main contributions of this dissertation are the following:

- A study of different code representations and how to identify vulnerabilities using them.
- A new approach for a KG model that combines the strengths of different code representations, taking advantage of the relevant information each one can provide for vulnerability detection. Containing as well additional information that is not usually presented in property graphs but is relevant for efficiency purposes of the analysis.
- A pruning method to irrelevant information for the vulnerability detection task (i.e., remove unnecessary nodes) from the knowledge graph, resulting in a more manageable and efficient graph for analysis.
- A multi-agent system that is able to efficiently traverse the KG and identify potential vulnerabilities.
- The KAVe tool for identification of vulnerabilities by the implementation of the KG with pruning and application of the multi-agent system.
- A detailed evaluation and validation of the tool, including the selection of test web applications, performance comparison with other existing tools, and validation of the identified SQLI and XSS.

This research led to a short paper submission, entitled *KAVe: A Knowledge-Based Multi-Agent System for Web Vulnerability Detection*, in the 23rd International Conference on Evaluation and Assessment in Software Engineering (EASE'23), ranked as A-conference in CORE.

1.4 Document Structure

This document is organised as follows:

- Chapter 2 covers the background information on web application vulnerabilities, specifically SQL injections and Cross-site scripting, static analysis techniques, different types of property graphs, and multi-agent systems, as well as a review of existing research and tools for detecting these vulnerabilities.
- Chapter 3 defines the problem of detecting vulnerabilities in web applications and presents the proposed solution, which includes the architecture of the KAVe tool and the challenges that arose during its development.

- Chapter 4 provides an in-depth explanation of the implementation and design of the KAVe tool, including the techniques used for parsing and analyzing the code, the methods used for pruning and navigating the knowledge graph, and the multi-agent system used to identify potential vulnerabilities.
- Chapter 5 describes the evaluation setup, the selection of test web applications, and the results of the performance comparison with other existing tools. It also includes the validation of the identified vulnerabilities.
- Chapter 6 provides a summary of the study and its main findings, as well as suggestions for future work.

Chapter 2

Background and Related Work

In this chapter, throughout Sections 2.1 to 2.6, we present essential concepts for understanding the problem we addressed and the techniques we used in our proposed solution. More specifically, the section introduces the software vulnerability problems, the two most prevalent web application vulnerabilities, and the technique most used to find them.

Afterwards, this chapter discusses a set of graph structures used to represent source code, the concept of knowledge graph for culminating series of data, and multi-agent systems.

In addition, Section 2.2 covers the related work relevant to the detection of vulnerabilities, using, in particular, data structures.

2.1 Vulnerabilities in Web Applications

Vulnerabilities in web applications have been a rising problem as their complexities increase along with the number of users online, and so do their risks. Such risks come mainly from software failures caused by design, implementation, or configuration errors. When an attacker discovers or deliberately inserts a flaw, a security breach happens with all the harm associated.

A significant research effort on web application security has been made with a particular focus on the detection and mitigation of vulnerabilities, which explains the appearance of organizations such as OWASP [2], WASC [3], or CWE [4] alongside other forums that try to identify and regulate the threats online (being OWASP the most renown internationally). From these organizations, we can observe that the most significant (e.g., most exploited) vulnerabilities are SQL injections (SQLi) and Cross-site scripting (XSS), which will be the focus of this work.

SQLi [5, 6] allows third parties to access the website's database to consult, change or delete content and insert malicious code. The vulnerability is present in scripts of SQL commands where user inputs, either via forms or URL, using GET or POST methods, are not adequately filtered, allowing intrusive database commands. To avoid it, the program-

mer must treat all parameters received by SQL statements that access the database to filter potentially dangerous strings, in addition to filtering SQL commands such as DECLARE, EXEC, SET, CAST, INSERT, DELETE, UPDATE.

```
1   $conn = mysql_connect("localhost","username","password");
2   $user = $_POST['user'];
3   $pass = $_POST['password'];
4   $query = "SELECT * FROM users WHERE username='user
5           AND password='$pass'";
6   $result = mysql_query($query);
```

Figure 2.1: SQL-injection vulnerability in a PHP script [1].

The SQLi vulnerability example of Figure 2.1 was taken from WAP [1]. As we can observe, the SQL connection to the database depends on a query that uses two user inputs without proper sanitization (cleaning of malicious input in security). Suppose a malicious user would like to attack this database. In that case, he could inject metacharacters in the input so that the query verification would always be true, and he could gain restricted access. For instance, if he enters the username as `admin'--` , he would get a query verifying whether the username is equal to `admin`, and all the other verifications would be commented, so independently from the password, he would access the admin account.

XSS allows the exploration of scripts to insert parameters in the URL. The attacker could run an external spam-sending script (usually Java scripts) from the hacked website. To avoid it, the web application must filter the occurrence of patterns such as `http://` in the URL parameters, preventing external scripts from being called.

```
1   $name = $_GET['name'];
2   echo $name;
```

Figure 2.2: A PHP script containing an XSS vulnerability

An XSS vulnerability can be as simple as the example in Figure 2.2, where some input is asked of a user and echoed in the application without sanitization. A malicious user could easily enter an input like `<script src="maliciousscript.js"> </script>` that would contain a script with limitless possibilities in how to exploit the system.

There are three distinct kinds of XSS vulnerabilities [7, 8]: stored XSS (also known as persistent), reflected XSS (also known as non-persistent), and DOM-based XSS (Document Object Model). The main difference between these is how the XSS travels before being delivered to the target.

Stored XSS usually occurs when malicious input (e.g., a malicious JavaScript) is stored on a server and later retrieved and used unsafely. In other words, it happens when the application stores input in its servers without validation and renders it on the user's browsers without sanitization, opening the door to the execution of malicious scripts. It is one of the most severe XSS since it has the prospect of attacking considerably more users than reflected or DOM-based.

Reflected XSS occurs when the malicious input (a java script) is executed and returns its output without being stored first (e.g., in a file or kept in a database). The application receives the malicious script, processes it in the server, and returns its execution output instantly. These issues often manifest when the server relies on user input to construct pages that display search results or error messages without making the data safe to render in the browser and without permanently storing the user-provided data.

DOM-based XSS is similar to reflected XSS, but in this case, the user input never leaves the browser. The application takes in the input, processes it on the victim's browser, and returns it to the user, so all the data flow takes place in the browser from the source to the sink.

These kinds of XSS are not exclusive, meaning they can overlap. For example, we can have both stored and reflected DOM-based XSS.

2.2 Static Analysis

Static analysis [9, 10] is a method of examining and evaluating software code or programs without executing it. It is used to identify potential bugs, security vulnerabilities, code quality issues, and other problems in the source code. Static analysis tools scan the code, analyze it, and generate reports on the results, providing developers with information that can be used to improve the quality and security of their software. The goal of static analysis is to identify problems early in the development process when they are easier and less expensive to fix.

Using static analysis techniques [11] allows development teams to improve code quality and achieve multiple benefits. The sooner a problem is found in the software, the cheaper it is to fix. Likewise, well-written source code dramatically reduces maintenance time (and cost), and the use of good object-oriented practices allows code to be flexible and support changes more smoothly, securely, and naturally.

The basis of this kind of analysis is checking the quality of the source code. This verification is carried out even before software execution, in opposition to unit tests, which validate the software based on the result of its execution. It is performed based on a set of pre-established rules. The analysis highlights problems to be corrected with maximal focus, resulting in greater efficiency in the quality improvement process.

2.3 Taint Analysis

Taint analysis [12, 1, 13] is a static analysis technique utilized to track the flow of user input through a program to identify possible security implications of the system.

If an outside user (an attacker) can change a variable, it will become tainted (compromised and hence suspicious), as well as other variables that depend on this one since they can potentially be at the root of a vulnerability. This analysis aims to mitigate threats such as SQLi, which generally originate when the user input is used without proper sanitization.

Sanitization in taint analysis refers to the process of removing or neutralizing potentially malicious data inputs in a program. It is performed on the tainted inputs to prevent them from causing harm, such as by exploiting vulnerabilities in the code. Sanitization can involve transforming the input data in some way, such as encoding or escaping characters or replacing malicious inputs with benign ones. The goal of sanitization is to ensure that untrusted inputs do not compromise the security or integrity of the system. Taint analysis tools create a list with these tainted variables or variables depending on them and check if they reach sensitive sinks, which represent built-in functions that, when executed with tainted variables, can derive dangerous operations such as access to databases or operating system calls. If so, these tools launch a warning of a possible security risk, e.g., a vulnerability.

On the other hand, variables can become untainted (uncompromised) if they pass through sanitization functions that will withdraw the vulnerability either by sanitizing the inputs or using prepared statements for SQL queries before they reach the sinks.

Sanitization will be different for each class of vulnerability, specifically for the two aimed kinds of vulnerabilities: For SQLi, if the application is exposed to potential hostile attacks from untrusted sources, sanitization ensures that the entered data adhere to security requirements by removing any suspicious characters that could alter SQL commands and harm the system when executed in the database. For XSS, malicious data in the form of special characters can be encoded in the inputs and compromise the system's security by performing unauthorized actions. Sanitization, in this case, involves checking, cleaning, and filtering data inputs from users, APIs, and web services of any unwanted characters or strings to prevent these types of attacks.

2.4 Types of Property Graphs

To allow a better understanding of the different types of property graphs, the graph examples will be given based on the following PHP code (Figure 2.3) adapted from code presented in [14]:

```php
function gymEntrance(int $age, bool $ismember){
    if (hasError()){ //function error
        return;
    }
    $fee = $_GET("fee");
    if(htmlentities($age) <= 18){ //is junior
        if($ismember){
            return;
        }
        $fee = $fee - 10; //discount 10
    }
    else if($age >= 60){ //is senior
        $fee = $fee - 5; //discount 5
    }
    echo($fee);
    $fee = 2;
    setFee(htmlspecialchars($fee)); //set entrance fee
    setTimer(1);  //set time inside
    if($ismember){
        setTimer(2);
        login();
    }
    return;
}
```

Figure 2.3: Base PHP code for the graph generation.

This is just a mock function in which the objective is irrelevant, although, for clarification purposes, we will explain how the function works. Basically, it simulates a gymnasium entrance scenario where people would get different price fees and times inside depending on their age and if they are already members (the dependencies are what interest us).

2.4.1 Abstract Syntax Trees (AST)

The Abstract Syntax Tree (AST) [15, 16] is a syntactic structure that relies on the programming source code designed to represent instructions and functions within a program. Ordinarily, the AST has a hierarchical design, starting from the base functions and flowing down to show more associated functions and the code inside them. Utilizing this tree allows programmers to analyze code without looking through the source code, which is complex and can expose flaws or vulnerabilities.

The tree is called abstract, not because it is difficult to understand or has an abstract shape, but because it does not account for all subtleties of actual program formulation representing only how programs are nested to build the final program. For example, distinct abstract syntax trees will not be formed independently if the variables are part of a

declaration list or a consecutive chain of declarations.

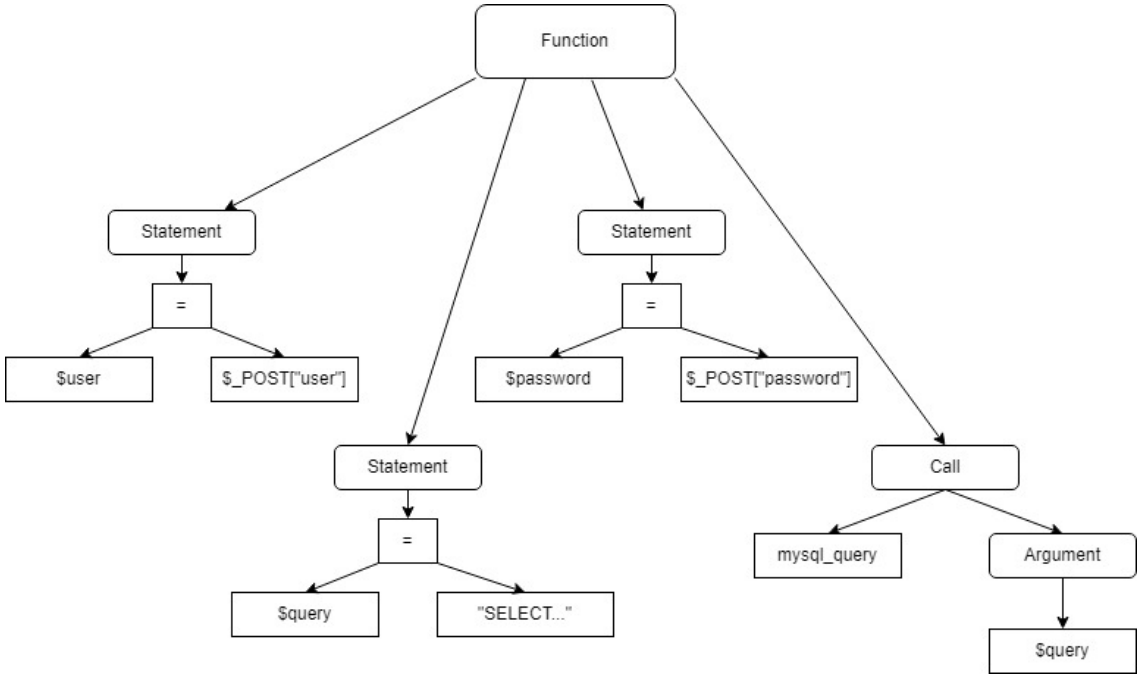

Figure 2.4: Abstract syntax tree of the code listed in Figure 2.1 (excluding line 1 for simplicity).

Figure 2.4 displays the AST of the code of Figure 2.1, lines 2 to 6. There are two classes of nodes in the AST: the inner nodes that denote operands (round corner boxes in the image) and the leaf nodes that represent the operators (rectangular boxes in the image). For instance, in the figure, an inner node could be a system/function call (Call) or an assignment statement (Statement), whereas a leaf node could be a variable ($user) or an expression/literal.

2.4.2 Function Call Graph (FCG)

A Function Call Graph (FCG) [17, 16, 15] is one of the most common and basic graphs used in code analysis. It is a directed graph representing the relationship between the caller and called functions. More particularly, each node represents a function, and each edge unites two functions directed from caller to callee. For instance, a cycle in the graph would indicate a loop (recursive procedure call) in the code.

Compared to other kinds of code property graphs, the main advantage it presents is that an FCG will allow an inter-procedural analysis since it is defined at a program level while the others are defined at a function level, so by using it in parallel, we get a much broader solution.

Considering that we have a program in which the main function only calls the `gymEntrance` function and that neither `setFee`, `setTimer` nor `login` functions call any other function, we would end up with an FCG like the one presented in Figure 2.5, where every caller function node has an edge directed to each of their callee functions.

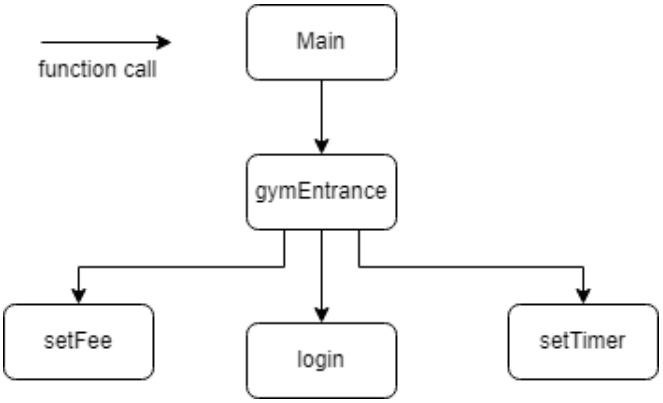

Figure 2.5: Function Call Graph for the code displayed in Figure 2.3.

2.4.3 Dependence Variable Graph (DVG)

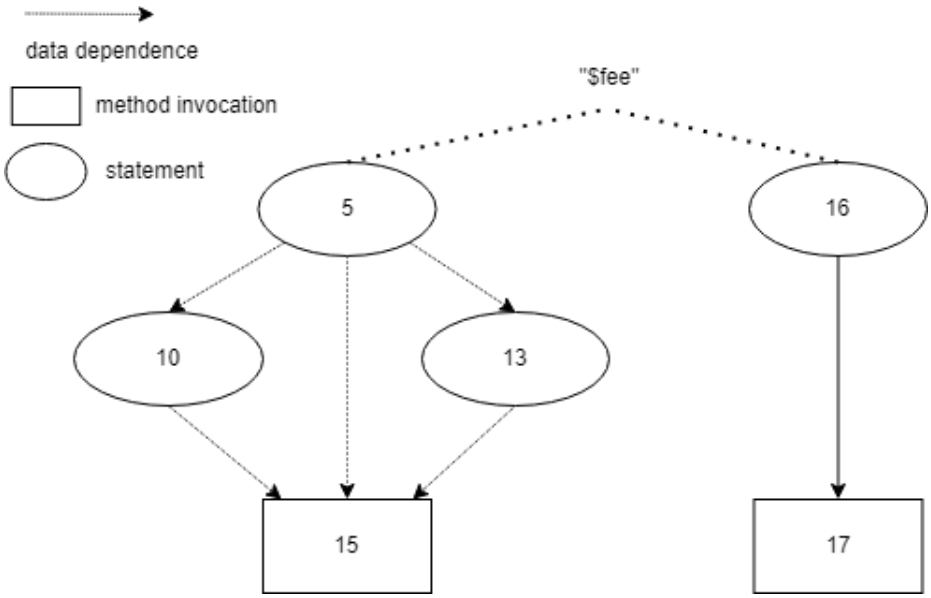

Figure 2.6: Dependence variable graph for the code displayed in Figure 2.3, relative to the variable `$fee`.

Dependence Variable Graphs [17] are directed graphs that represent dependency relationships between variables belonging to the same code structure. This kind of graph has mainly been used as a base structure to eliminate the actions of malicious code obfuscation techniques.

In this approach, each code instruction corresponds to a node, and for each variable that this instruction handles, an oriented edge will be inserted, linking this instruction to the next line of code that manipulates this same variable.

The example in Figure 2.6 represents the data dependencies in the initial function in Figure 2.3 (with the line numbers in the nodes for simplicity), which in this case are only related to the `$fee` variable that will be changed accordingly if the user is either junior or senior. The `$fee` in line 5 is initialized with a value given by the user and then used as a discount that could be applied to lines 10 or 13 according to its age. The resulting `$fee` will be printed in line 15, then it is redefined on line 16 just to give an example of dependency break, and the new value is given to the function `setFee` on line 17.

2.4.4 Control Flow Graphs (CFG)

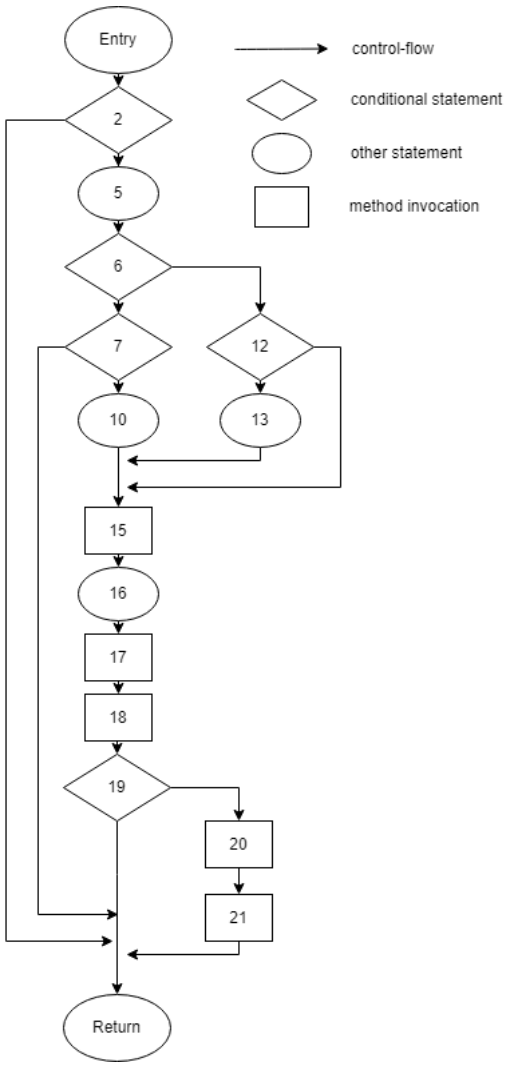

Figure 2.7: Control Flow Graph for the code displayed in Figure 2.3.

A Control Flow Graph [16, 18, 19] is a graphical representation of the flow of control through a program, showing the possible paths that the execution can take based on the decisions made in the code. It consists of nodes, which represent basic blocks of code, and directed edges, which represent the flow of control from one basic block to another. The edges, when coming out of conditional statement nodes, can take values of either true or false accordingly to the value of its inherited predicate and will accordingly follow the respective path of flow.

A well-built CFG should accurately represent all flow dependencies in the code. This is a critical aspect because if the CFG fails to maintain the structure of the source code when being used for comparison or code optimization, it can result in incorrect and potentially harmful outcomes.

Because CFGs lack information on data flow, they are not the most suitable choice for analyzing vulnerabilities in code.

The graph in Figure 2.7 represents all the possible paths that the execution can take from the function's entry to a return statement.

2.4.5 Program Dependence Graph (PDG)

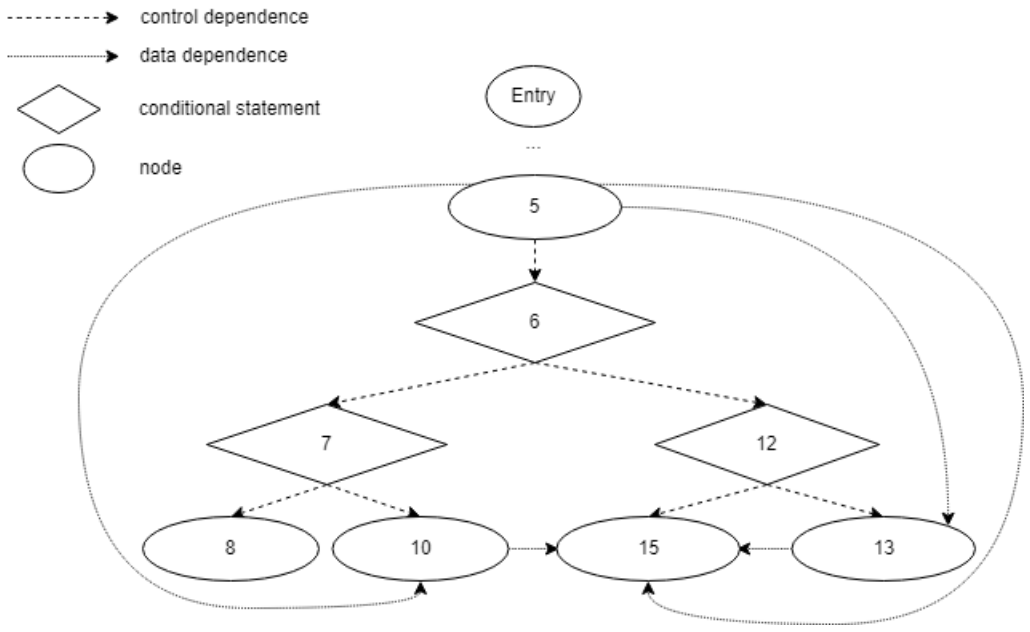

Figure 2.8: Program Dependence Graph for the code displayed in Figure 2.3, from line 5 to 15.

The Program Dependence Graph [17, 20, 21, 16] is one of the most complete data structures used in static analysis since it allows us to represent both data and control dependencies compared to the other graphs where we would only have one of them.

The nodes in the graph represent individual statements, and the edges represent dependencies between those statements.

We can find two types of edges in PDG. For instance, data dependence edges imply that a receiving node relies on the value of a variable defined previously in the outgoing node. Control dependence edges imply that the execution of a statement depends on a predicate.

The PDG supports incremental optimization, allowing transformations to be initiated and applied only to affected dependencies.

As we can see in Figure 2.8, the PDG resembles the constitution of the CFG but introduces slicing to consecutive statements (statements that will occur in the sequence of a previous one independently from the result). Same as the CFG, it includes and control dependency edges, but it also adds both data dependencies from the DVGs.

2.5 Knowledge Graph (KG)

A knowledge graph [22, 23] is a powerful tool for representing and organizing knowledge in a structured and interconnected manner. It can help to uncover new insights and relationships within data, making it a valuable resource for a wide range of applications.

One of the key features of a knowledge graph is its ability to capture relationships between different concepts and entities. By representing these relationships as edges between nodes, a knowledge graph can provide a detailed and comprehensive understanding of a particular subject or domain. This not only makes it easier to navigate and explore complex networks of information but also allows for the discovery of new connections and insights that might not be immediately apparent.

Because of their ability to capture and organize knowledge in this way, knowledge graphs are increasingly being used in a variety of applications. For example, they are often used in search engines to provide more accurate and relevant results and in recommendation systems to suggest products, services, or content that might be of interest to a user. They are also being used in natural language processing to help machines understand and interpret human language, as well as in areas such as artificial intelligence and machine learning to improve the accuracy and relevance of predictions and recommendations.

Figure 3.5 displays an example of a KG, which we will explain in Chapter 3.

2.6 Multi-agents intelligent systems

2.6.1 Intelligent Agents

There is no universal consensus about what an intelligent agent (from now on referred to as only agent for simplicity) definition could be since this term creates some controversy

and debate in the academic world [24]. Mainly this discussion occurs due to the fact that agents differ a lot in certain key domains, e.g., some believe that it is crucial that the agent is able to learn from his own background, but some applications require the exact opposite that the agent keeps a uniform behavior during its lifetime.

Although there is no official definition, it is crucial to establish some type of ground on this matter. Some of the most reliable definitions for an agent are:

- ”An agent is anything that can be viewed as perceiving its environment through sensors and SENSOR acting upon that environment through actuators” [25].
- Computer systems that live in complex and dynamic environments feel and act autonomously in this environment and, doing so, accomplish a set of goals or tasks for which they were designed [26].
- An agent is a persistent software entity dedicated to a specific purpose. Persistence distinguishes agents from subroutines because they have their ideas about how to perform tasks and have their agendas. Specific purpose distinguishes them from applications multi-function [27].
- Intelligent agents are entities of software that perform a set of operations on behalf of a given user or another program with a degree of independence or autonomy. In doing so, use some knowledge or representations of the goals and desires of the user [28].

Based on the definitions, we can generally describe an agent as a computational entity that works continuously and autonomously in a restricted environment in which there may be other agents with or without common characteristics. Informally, an agent could be someone or something that acts as a proxy with the specific purpose of performing actions that can be understood as beneficial within the context in which he acts.

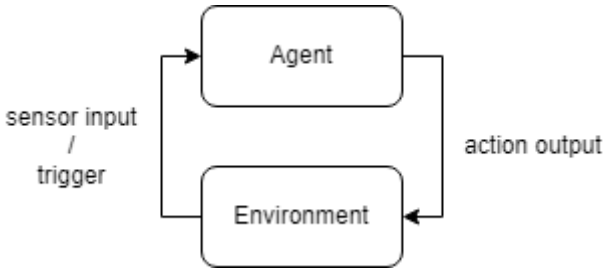

Figure 2.9: Agents interaction with the Environment

A computational system can be seen as an agent if it can perceive its environment through sensors/triggers and act on it through actuators (e.g., Figure 2.9), adding characteristics such as autonomy, communication, reactivity, and proactivity.

2.6.2 Multi-Agents Systems

A multi-agent system is a system in which two or more agents interact or work together to perform a particular set of tasks. These agents exhibit two fundamental characteristics: being able to act autonomously, making decisions leading to the satisfaction of their objectives; being able to interact with other agents using protocols of human-inspired social interaction and including at least some of the following functionalities: coordination, cooperation, competition, and negotiation.

It is a relatively new field in computer science, the beginning of research in this field took place in the 1980s, but only in the mid-1990s has it gained notoriety [29]. One of the essential points to allow the construction of societies of agents consists of managing the interactions and dependencies of the activities of the different agents in the context of the multi-agent system, e.g., to coordinate these agents. This way, coordination plays an essential role since these systems are inherently distributed [30].

Different coordination methodologies have been proposed by different authors, divided into two main groups: methodologies applicable in domains containing competitive agents ("self-interested"), e.g., agents concerned with their own good, and methodologies applicable to domains containing cooperative agents, e.g., agents that include a notion of concern for the good of the whole (in our case we will use this second kind of agent).

Multi-agent systems include several agents that interact or work together, which may comprise homogeneous or heterogeneous agents. Each agent is an element capable of autonomous problem-solving and operates asynchronously concerning other agents. For an agent to operate as part of the system, it is necessary to have an infrastructure that allows communication and/or interaction between the agents.

Each agent will have a sphere of influence on the environment, e.g., it will be able to influence different parts of the environment [31]. These spheres of influence may coincide depending on the relationships existing between agents.

2.7 Related Work

2.7.1 Use of Static Analysis in Vulnerability Detection

Over the years, a series of static analysis tools (SATs) have been developed to analyze source code to detect vulnerabilities that make it susceptible to attacks. Back in 2004, Huang et al. [9] were one of the first to address the topic of static analysis in the PHP context. The authors presented a lattice-based algorithm approach that tainted code through type systems and typestate. In this process, distinct sections that included the tainted "vulnerable" code would be instrumented with runtime guards, allowing verification and validation to occur without user intervention. Following this study, they presented another method [10] called bounded model checking (BMC). It is somewhat more complex

than the previous method but has some advantages in terms of precision, soundness, and providing counter-examples as well as a complete verification.

Wijngaard et. al. [17] presented dependence analysis, a form of static analysis that evaluates program operations in order to produce execution order constraints. These constraints have the purpose of controlling whether an operation data depends on another operation and if it should be executed only after the previous. The authors take advantage of merging PDGs (generated from combining control and data dependencies) from single procedures of the application within a call graph, which will generate a System Dependence Graph (SDG) that represents the entire structure of the program.

They state that although there is extensive research on intra-procedural dependence analysis, there are no PDG or SDG libraries for dynamic languages such as PHP. Therefore they tried to implement those libraries by studying dependence analysis on PHP and concluded that although there is a prominent number of PHP language features that are challenging to model, they are barely used, and even if it is not possible to get an accurate result, a partial result would still be valuable.

Another important work was WAP (*Web Application Protection*) [1] is a tool developed by Medeiros et al. that connects taint analysis in parallel with data mining to detect vulnerabilities in PHP source code. Initially, the taint analysis identifies potential vulnerabilities in the code, and then the data mining is used to predict which of these candidates are false positives, bringing together two approaches not often incorporated alongside each other.

Besides finding vulnerabilities, WAP explores a methodology to protect web applications automatically while allowing the programmer to learn from his mistakes by being integrated into the process.

The approach analyzes the source code, searching for input validation vulnerabilities, and then the code corrector adjusts the code accordingly by defining a fixture and inserting it into the code, which consequently involves altering the PHP file. Most of the time, the fixture consists in calling a function that sanitizes the data before it reaches a sensitive sink without changing the correct behavior of the application, which is validated by using regression to test the results.

Kim et al. [32] proposed a new technique called *FuzzOrigin* to detect universal cross-site scripting (UXSS) vulnerabilities in web browsers. Origin fuzzing is a black-box testing technique that modifies the origin of a website during execution to trigger potential UXSS vulnerabilities. The authors claim that their approach can effectively detect UXSS vulnerabilities in a variety of browsers and can be integrated with existing security testing tools.

The methodology consists of several steps:

1. Identify the target browser's origin-handling mechanism and extract a set of origin strings that can be used to construct test cases.

2. They then generate a set of test cases by combining the origin strings with different payloads and executing them against the target browser.

3. They monitor the browser's behavior during execution, looking for abnormal behaviors or crashes that indicate the presence of a UXSS vulnerability.

4. If a potential vulnerability is detected, it is further investigated to confirm its presence and to determine the root cause of the vulnerability.

The authors have evaluated their approach on popular web browsers, including Chrome, Firefox, and Safari, and found several previously unknown UXSS vulnerabilities. They have also compared their approach with other existing techniques, such as input validation, output encoding, and browser-specific fuzzing, and have shown that origin fuzzing is more effective in detecting UXSS vulnerabilities. Additionally, they provide a detailed methodology for conducting origin fuzzing, which can be used as a reference for other researchers interested in this area.

2.7.2 Use of Graphs in Vulnerability Detection

In 1987, Ferrante et al. [20] presented the concept of program dependency graphs that were first introduced to perform program slicing. This representation exposed dependencies between statements and predicates, allowing the control of the propagation of attacker-controlled data in particular. In these graphs, computationally related elements of the program were connected by dependencies. Thus a single passage through the dependences is enough to perform optimizations allowing transformations such as vectorization that previously required special treatment of control dependence to be performed uniformly for both control and data dependencies. From this knowledge, a lot of studies derived to use PDGs as a way to find vulnerabilities in code.

PIDGIN [21] is a tool with the primary goal of finding security guarantees in legacy programs to create new or adjust existing policies with application development and develop policies based on known vulnerabilities. The need for a tool like this urges to the fact that nowadays, many applications are working with confidential and untrusted data from users, so developers should be aware of how this information flows through the apps and guarantee that the data is secured, the information flow is different for every application so security guarantees should be application-specific. Tools that already exist are inadequate since they often ignore significant information flows or require a lot of local annotations, which prevents functionality in terms of testing and deployment. Other than that, they also lack support to enforce application-specific policies.

To solve these problems, PIDGIN enables the specification and enforcement of security guarantees specific to different applications. It analyzes and understands the programs by combining the usage of PDG's that capture the entire information flows in the application and a custom query language for PDG. The queries in this language express properties

about the paths and can be used to specify global security policies. These queries also express application-specific guarantees, making PIDGIN scalable and allowing developers to quickly explore an application's security policies and determine if they are satisfied. The enforcement of these policies does not prevent the development or testing of the application since the program does not depend on them, solving another problem of already existing tools. PIDGIN also enables regression testing of information security guarantees to allow developers to know if changes in the code made the application violate previously insured policies.

Since PHP is so widely spread, there are many security studies that were successful in finding out vulnerabilities, however, only in a controlled manner, with only a few projects being tested.

Backes et al. [16] focused on making these techniques more scalable to be applied to more websites and improve the overall security of web applications. To achieve this goal, a series of inter-procedural analysis techniques were applied to code property graphs that include the program's syntax, control flow, and data dependencies in the same graph structure, making it a great and adaptable structure to analyze large amounts of code. The code property graphs were stored in highly efficient graph databases that allow graph transversals to demonstrate the technique's feasibility and allow all users to query the database to find out multiple kinds of vulnerabilities. The program was evaluated in 1,854 popular PHP projects on GitHub, totaling almost 80 million lines of code, finding out hundreds of vulnerabilities.

MERLIN [13] is a tool that brings a new approach to detecting input validation vulnerabilities in web apps, aiming to solve problems that previous tools found in detecting vulnerabilities in code written in multiple languages and learning how to detect vulnerabilities automatically. It functions using an intermediate language parsed from Java bytecode generated from diverse high-level languages used in web apps. It combines data flow analysis (with CFGs generated from the Java bytecode) and the intermediate code to detect potentially vulnerable code.

They discover vulnerabilities automatically by performing taint analysis along with machine learning, starting by vectorizing the tainted code into an attribute vector and then classifying it as vulnerable or not vulnerable with a machine learning classifier that learns which kinds of instructions are related to vulnerabilities and, therefore can predict other vulnerabilities not considered in the beginning.

The chosen intermediate language used to translate the high-level languages was Jimple since it is easily obtained from Java bytecode, so if a language can be translated to Java bytecode, it can be translated to Jimple. In the specific case of MERLIN, they focused on PHP translation, from which they tried to interpret the functions. By the end, they were able to detect SQLi, cross-site scripting, remote file inclusion, local file inclusion, directory/path traversal, source code disclosure, operating system command injection, and

PHP command injection, which are all high-risk vulnerabilities.

BEACON [33] is a new method for grey-box fuzzing that aims to improve the effectiveness of fuzz testing in web applications by using a combination of symbolic execution, machine learning, and provable path pruning. The method uses symbolic execution to generate test inputs that can reach a specific part of the code, called a ”fuzz target,” and a machine learning model to predict which inputs are most likely to trigger bugs. Provable path pruning is used to eliminate paths that cannot lead to bugs and reduces the number of paths that need to be tested. This improves the efficiency of the fuzz testing process by reducing the search space. To achieve this, BEACON uses a CFG of the software to identify the potential fuzz targets and generate the inputs. This method can be seen as a significant contribution to the field of vulnerability detection in web applications as it helps to find more vulnerabilities and also reduce the time and resources needed for the process.

FUGIO [34] is an Automatic Exploit Generation tool that was designed to exploit PHP Object Injection vulnerabilities. The authors propose a three-step methodology:

1. Vulnerability detection: In this step, the tool uses static analysis to identify the locations of PHP Object Injection vulnerabilities in the code.

2. Exploit generation: Once the vulnerabilities have been identified, FUGIO generates a set of payloads that can be used to exploit them. The payloads are created by analyzing the code and identifying the objects and methods that can be used to perform the desired actions.

3. Exploit execution: The final step is to execute the generated payloads and check whether the vulnerability can be successfully exploited.

It uses graph-based representations to analyze the PHP code and generate payloads. The tool uses a control-flow graph to represent the control flow of the program and a call graph to represent the relationships between functions and methods. These graphs are used to identify the objects and methods that can be used to perform the desired actions and to generate payloads that can be used to exploit the vulnerabilities.

In particular, they use a specific type of graph called Object-Oriented Call Graph (OO-Call graph) which models the interactions between objects, classes, and methods in the PHP code. This graph is used to identify the objects and methods that can be used to perform the desired actions, and to generate payloads that can be used to exploit the vulnerabilities.

It is capable of generating both local and remote exploits, and it does not require any knowledge of the inner workings of the vulnerable code or the target system. The tool can be used to find and exploit PHP Object Injection vulnerabilities in a fully automated manner, making it a useful tool for both security researchers and attackers.

VulEye [35] is a novel Graph Neural Network (GNN) based approach for automatically detecting vulnerabilities in PHP applications. The approach uses a Program Dependence Graph (PDG) to represent the PHP source code, slices the PDG with sensitive functions contained in the source code into sub-graphs called Sub-Dependence Graphs (SDGs), and uses the SDGs as input for a GNN model. The model includes three stack units with a GCN layer, Top-k pooling layer, and attention layer and uses MLP (Multi-Layer Perceptron) and softmax as a classifier to predict if the SDG is vulnerable. The method is evaluated on a PHP vulnerability test suite in the Software Assurance Reference Dataset, and it is found to have a macro-average F1 score of 99% in the binary classification task and 95% in the multi-classes classification task. The approach is also able to locate the precise area of the vulnerability.

Chapter 3

Knowledge-based Agent-system Vulnerability-detector (KAVe)

This chapter explains our approach's methods to detect web vulnerabilities. Starting with the challenges we faced during its elaboration in Section 3.1, its architecture presented in detail in Section 3.2, how the KG pruning works in Section 3.3, and the intelligent agents' system behavior in Section 3.4.

Other than just detecting the vulnerabilities, we try to do it efficiently by taking advantage of a series of essential graph structures (CFG, DVG, FCG, and PDG) capable of representing the code and flows in different but related manners and combining them in an expanded Knowledge Graph (KG).

The KG contains several relevant pieces of information that allow an innovative kind of pruning through several stages to minimize the effort of going through it multiple times as well as skipping not pertinent parts of the graph that would otherwise need to be analyzed.

Multiple intelligent agents will then navigate the KG, creating thus an intelligent and oriented navigation system that will perform static analysis techniques such as taint analysis to locate the vulnerabilities.

3.1 Challenges

This section presents some of the challenges we faced while constructing our tool to find vulnerabilities in web applications' source code, leveraging a knowledge graph pruned for efficiency that is further analyzed by a multi-agents system. Also, we present some of the main ideas we had to overcome these challenges.

3.1.1 Dealing with the complexity of the web applications source code

One of the main challenges we faced was to deal the extensive and constantly changing nature of the PHP language, which makes it difficult to create a comprehensive tool that

can identify all potential vulnerabilities. Additionally, PHP code is often complex and may include multiple layers of abstraction. Different web applications may have different requirements and dependencies, making it difficult for a tool to analyze and identify vulnerabilities accurately.

Furthermore, PHP code is often integrated with other technologies, such as databases and web servers, which can introduce additional complexities and challenges for a vulnerability-finding tool. Overall, building a tool to find vulnerabilities in PHP code requires a deep understanding of the PHP language and its common subject to pitfalls and vulnerabilities.

As we gather from these complications, building a tool that is comprehensive enough to deal with all these complications is close to impossible and would take a lot of time and effort to be built. So for our limited time span to build the proposed solution, we focused on discovering the main common patterns that can represent vulnerabilities in order to find the largest amount we could. Also, we avoided getting into details of sources outside of PHP even though they could be integrated into the web application code. Since they less often contain vulnerabilities, the ones that could arise would require a different type of analysis.

3.1.2 Integration of already existing tools into our solution

We may encounter several potential challenges and problems when integrating existing software tools using custom code. It is often difficult to understand the different tools as each software tool may have unique APIs and requirements for integration, making it demanding to understand how they work and how they can be integrated. Writing custom code to integrate multiple software tools can be complex and time-consuming, especially if they have complex outputs and need to be combined (in our case, where we need to combine multiple graphs).

It also clearly has its benefits, if we had a tool that already provided some graphs we needed, it would have taken the burden of thinking about how to build it. Overall, making custom code can provide more flexibility and control over the integration process, so after testing some existing tools, we decided it would be beneficial to only keep a few and mainly build custom ones.

3.1.3 Converting the source code statically into the property graphs

To convert source code into property graphs, we needed to create tools to parse and convert it into a graph data structure. This process is even more challenging as we have to do it by employing static analysis, not being able to run the code to check where the information flows.

To build a parser tool to convert source code into a property graph, one must have a strong understanding of both parsing and graph data structures.

We needed to design and implement a grammar that defines the programming language's syntax we want to parse. This grammar was then used by the parser to identify the different elements of the source code, such as variables, functions, and classes, and to ensure that the code is syntactically correct.

Once the parser had been implemented, we also needed to design a data structure to represent the property graph (leveraging the Networkx [36] framework for doing so).

Once the property graph has been generated, we visualize it using a graph visualization tool (Graphviz [37]). To allow us to see the different elements of the source code and how they are connected.

3.1.4 Combining different property graphs into one extended knowledge graph

Several potential difficulties may occur when combining different property graphs into a single extended knowledge graph. For instance, property graphs may use different data formats or structures and contain errors, inconsistencies, or outdated information, making it challenging to merge them into a single graph.

The data mapping and integration from different property graphs to a standard schema can be time-consuming and error-prone, especially if the graphs have different structures or use different terminology. Also, combining multiple property graphs into a single graph can result in a very large and complex graph, which may be difficult to manage and query efficiently.

We could take a few different approaches to combine multiple property graphs into a single extended knowledge graph. The approach we chose was simply merging the graphs by combining their nodes and edges into a single graph. This can be done by iterating over the nodes and edges in each of the individual graphs and adding them to the extended knowledge graph. Since we were the ones that created the property graphs, it means we could manipulate the data in our favor to facilitate integration.

So we came up with the following plan:

- Identify the property graphs we want to combine, considering factors such as the overlap of information, the reliability of the sources, and the potential for conflicts between the graphs.
- Determine the standard schema or structure used for the combined knowledge graph. This may involve identifying shared properties or entities and defining how they will be represented in the combined graph.
- Import the data from the individual property graphs into the combined knowledge graph. This implicates mapping the data from the individual graphs to the knowledge graph and resolving any conflicts or inconsistencies.

- Validate the combined knowledge graph to ensure that all of the data has been imported correctly and that there are no errors or inconsistencies in the merged graph.

- Optimize the search over the combined knowledge graph to improve its performance and usability by partitioning it into smaller sub-graphs (in this case, PDGs) or applying other techniques to improve its efficiency, such as pruning.

Once all the property graphs are combined into a single extended knowledge graph, we can use various techniques for graph querying and visualization to explore and use the information in the graph. This can be a powerful way to gain insights and uncover relationships between different pieces of information previously scattered across multiple property graphs.

3.1.5 Implementing pruning over the knowledge graph without losing essential information

Pruning a knowledge graph is removing unnecessary or redundant information from the graph to make it more manageable and navigation over it more efficient. When implementing pruning, it is important to carefully consider what information is essential to the knowledge graph and what can be removed without losing the crucial data.

The essential information in the knowledge graph will typically include the most important nodes and relationships in the graph and any information critical for the knowledge graph to represent the application correctly. Also, we should identify any redundant or unnecessary information in the knowledge graph. This may include duplicate nodes, relationships, or information not relevant to the main purpose of the knowledge graph.

Overall, the key to implementing pruning on a knowledge graph is to carefully consider what information is essential to the graph and what can be removed, preferably by using a combination of automated algorithms and manual curation to make it as accurate and efficient as possible.

3.1.6 Leveraging a multi-agent system to identify the vulnerabilities

A multi-agent system is composed of multiple agents that can interact with each other and their environment to achieve a certain goal. In the context of identifying vulnerabilities, a multi-agent system could be used to divide the work of identifying vulnerabilities among multiple agents, allowing for more efficient and thorough analysis.

Each agent in the system could be responsible for analyzing a specific part of the system, such as a particular graph, node or edge. The agents could then communicate with each other to share information and coordinate their efforts to identify vulnerabilities.

One potential advantage of using a multi-agent system for vulnerability identification is that it can help to overcome the limitations of individual agents. For example, an individual agent may not have the expertise or knowledge to identify all possible vulnerabilities, but a group of agents with different purposes can work together to identify a broader range of vulnerabilities.

Additionally, a multi-agent system can also reduce the time and resources required for vulnerability identification by allowing multiple agents to work on the task simultaneously. This can be especially useful in large and complex systems where manually identifying vulnerabilities can be a time-consuming and resource-intensive process.

Although the advantages are clear, a series of problems may arise from the multi-agent system integration. For example, in terms of:

- Complexity: A knowledge graph is an elaborate and interconnected system, and it can be challenging to design and implement a multi-agent system that can effectively navigate and analyze its complexity.

- Communication: In a multi-agent system, the agents must be able to communicate and coordinate with one another in order to identify vulnerabilities effectively. This requires the development of robust communication systems and the ability to handle potential conflicts or misunderstandings between agents.

- Scalability: As the size and complexity of the knowledge graph grows, it may become increasingly difficult for the multi-agent system to analyze and effectively identify vulnerabilities in a timely manner.

- Performance: The performance of a multi-agent system may be limited by the speed and efficiency of the individual agents and their ability to process and analyze data.

- Interoperability: If the agents in the multi-agent system are not designed to work together seamlessly, it may not be easy to integrate them into a cohesive system and utilize their capabilities.

3.2 Architecture

The solution's architecture is designed as depicted in Fig. 3.1. Our tool KAVe will receive PHP source code and produce a list of vulnerabilities it discovers. The process involves generating CFGs and DVGs, integrating them into a program PDGs, merging the PDGs with an FCG to create a KG, pruning the KG, and finally passing it to a Vulnerability Detector, which uses a multi-agent system to analyze and identify the vulnerabilities.

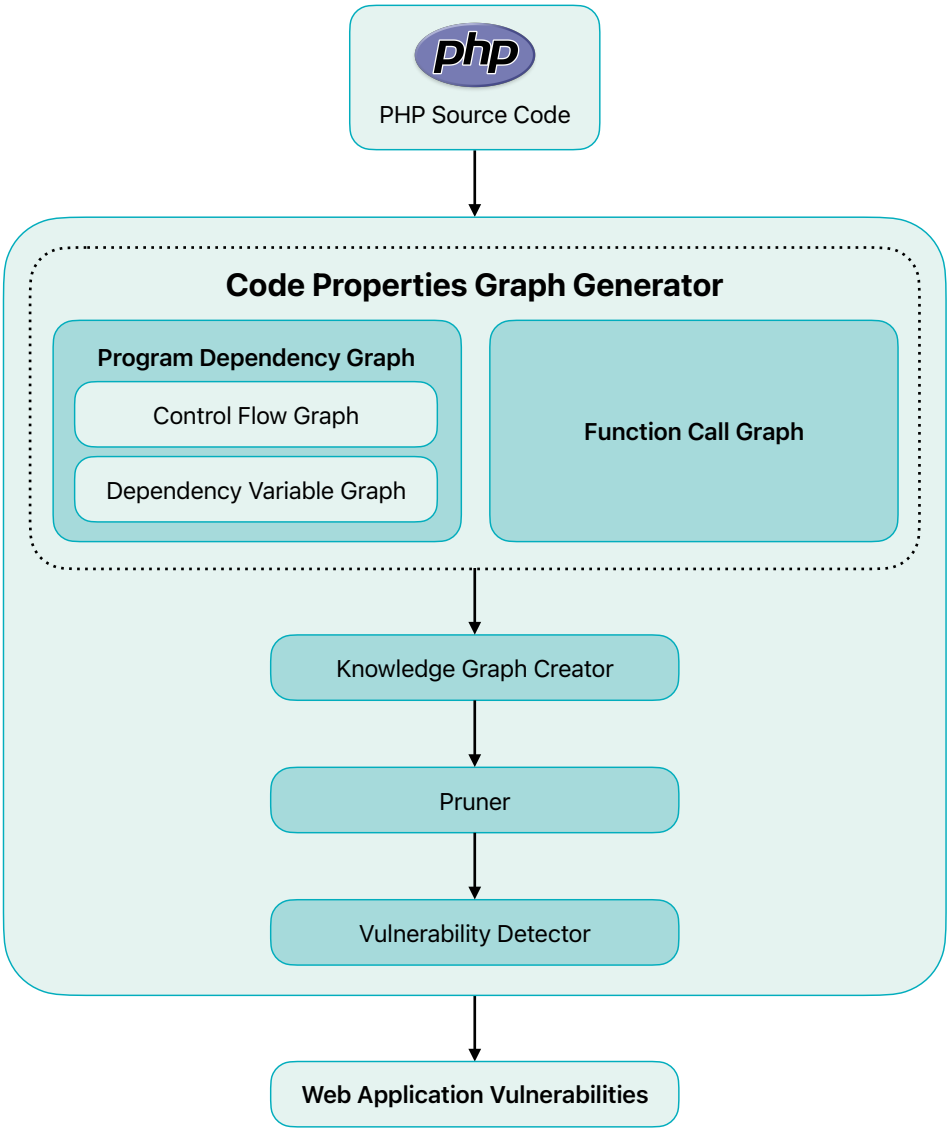

Figure 3.1: Architecture of the proposed solution.

3.2.1 Dependence Variable Graph

The Dependence Variable Graph (DVG) includes all the variable dependencies of a function, defining which variables are dependent on others. In our approach, we propose the creation of several dependency variable graphs, one for each variable in each function of the program we analyze. The DVG generator works as a lexer that will static analyze the PHP source code and convert it into the multiple graphs we need.

In terms of structure, each DVG will be an oriented graph $G = (N, E)$, where N is a set whose elements are called nodes representing lines of code, and E is a set of ordered pairs of nodes whose elements are called edges. Subsequencially, each node is a tuple N_i = <*i, d, v, b, l*>where:

- *i* is the index of the line corresponding to that node;

- *d* is the relative depth where the statement corresponding to the variable associated with the DVG appears inside a function. It changes as the flow goes inside (increase) or out (decrease) of, for example, conditional clauses or cycles;
- *v* is an abstract version of the variable represented that increases each time the variable is redefined and thus loses its dependency (except for when it uses its own value for the new definition);
- *b* is an index used to determine which branch of a sequential conditional clause;
- *l* is a label given to the node depending on whether it contains either an entry point, an SQLi/XSS sensitive sink or an SQLi/XSS sanitization function after evaluating the statement.

The edges are represented by pairs $E = (N_i, L(N_i)), (N_k, L(N_k)))$ where i/k represent the indexes of the corresponding node lines, N_i/N_k correspond to the nodes, and $L(N_i)/L(N_k)$ correspond to the labels of the respective nodes, the labels are added in order to be easily accessible to facilitate the initial stage pruning and avoid repeating the passage through the DVG (see Section 3.3).

Since we have multiple DVGs for each function variable, they will be combined in another structure to be further added to the PDG.

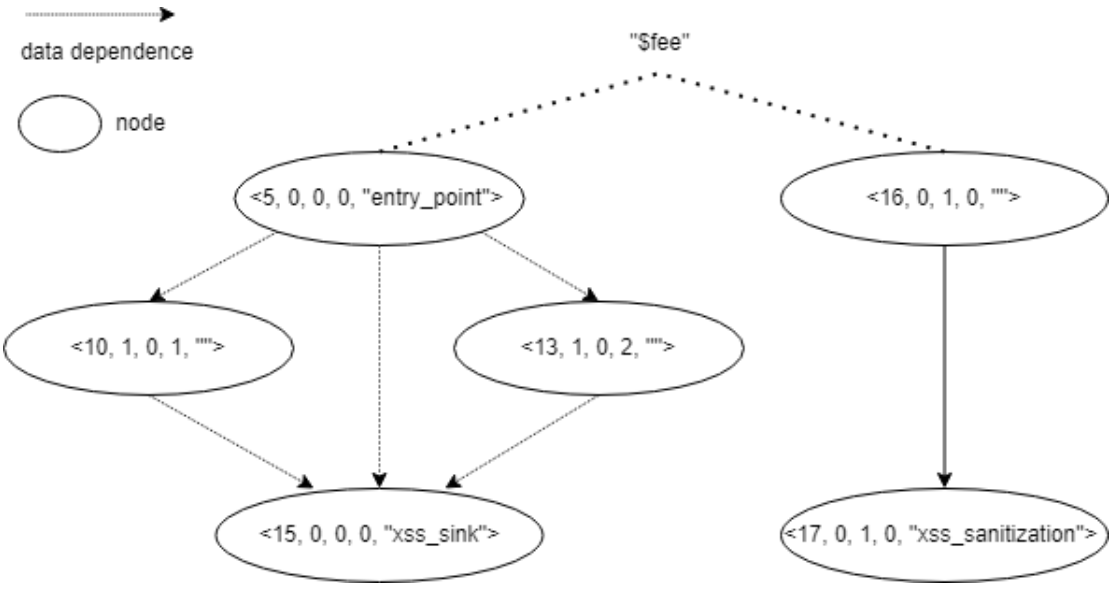

Figure 3.2: Enriched Dependency Variable Graph of the variable *$fee* in the code of Fig. 2.3

Figure 3.2 shows an interaction of the DVG of the variable *$fee* from Fig. 2.3. The variable *$fee* is defined for the first time in line 5 and outside any conditional statement. It receives an entry point ($_GET), the node <5, 0, 0, 0, "entry_point"> represents its

definition, where 5 denotes the line, and the following three zeros indicate the variable is assigned outside a conditional statement, and it is the first time it is defined. The last element ("entry_point") states the variable received an entry point.

We can notice changes on *d* and *b* values when changing to the nodes with index 10/13, where *d* increases due to getting inside a conditional statement, and *b* shows that in 10 we are on the first branch of the condition and 13 on the second branch. We can also notice an increment on *v* value at index 16 corresponding to a line where the variable *$fee* has been redefined, and it is not dependent anymore on its past version.

3.2.2 Control Flow Graph

The Control Flow Graph (CFG) contains the control flow information of a function, describing all the paths that might be traversed via the program in its execution. Since the CFG is specific to a function (intraprocedural), we will generate multiple CFGs, one for each function our program contains, to detect vulnerabilities throughout the entire program. Our CFG generator will work as a lexer, so it will analyze the PHP source code statically to generate the graphs corresponding to the CFGs.

Structure-wise, the CFC is an oriented graph $G = (N, E)$, where N is a set whose elements are called nodes representing lines of code, and E is a set of ordered pairs of nodes whose elements are called edges.

More specifically, in our case, a node is a tuple where N_i = <*i, d, r, c, l* >, where:

- *i* as the index of the line corresponding to that node;
- *d* identically to *d* in the DVG, it represents the depth of the corresponding statement relative to the function we are evaluating; this will change according to the conditional statements as well as cycles that the flow passes through;
- *r* is a boolean parameter that will inform if the node contains a return statement (*r* = *T* if N_i contains a return statement else *r* = *F*, Fig. 3.3);
- *c* is a boolean parameter that will inform if the node contains a conditional statement(*c* = *T* if N_i contains a conditional statement else *c* = *F*, Fig. 3.3);
- *l*, in similarity as in the DVGs, is a label given to the node depending on whether it contains either an entry point, an SQLi/XSS sensitive sink, or an SQLi/XSS sanitization function. After evaluating the statement, adding *l* information here will facilitate the junction with the variable dependency edges and later generate the program dependency graph.

The edges are represented by pairs of either $E = ((N_i, L(N_i)), (N_k, L(N_k)))$ and/or $E = ((N_i, L(N_i)), (R))$, where i/k represent the indexes of the corresponding node lines,

N_i/N_k correspond to the nodes, and $L(N_i)/L(N_k)$ correspond to the labels of the respective nodes, if the second node corresponds to a return statement then R will represent the unique return node as the flow of all the program paths will end in this distinguished node.

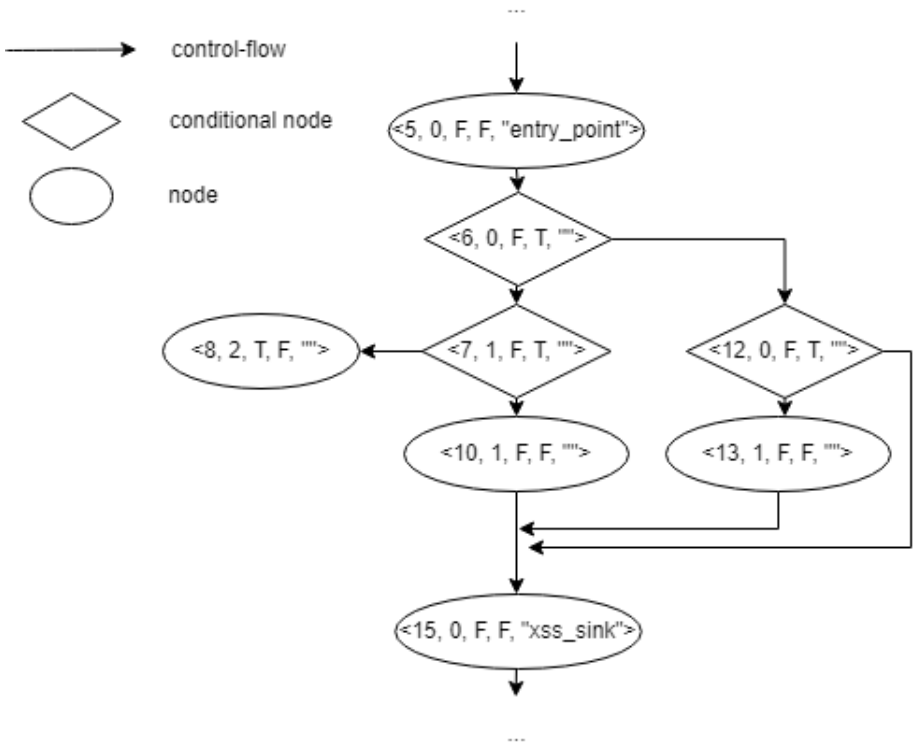

Figure 3.3: Enriched Control Flow Graph example from the code of Fig. 2.3 between line 5 and 15.

Figure 3.3 shows a representation of the CFG relative to the code in Fig. 2.3 between lines 5 and 15 (for simplicity). In the first node <5, 0, F, F, "entry_point">, we can notice the number 5 as the reference to the line number the node is associated with, followed by a 0, showing that there is no depth (outside any cycle/conditional statement), then we got two *F*'s indicating that the node does not correspond to a return or conditional statement, respectively, and finally "entry_point" showing that there is an entry point on that line.

We can see how the *d* value changes as we get inside conditional statements from index 6 to 7 and 7 to 8. The *c* value turns into *T* when it gets inside a conditional statement, e.g., index 6, similarly, the *r* value turns into *T* when it gets inside a return statement, e.g., index 8.

3.2.3 Program Dependency Graph

The Program Dependency Graph (PDG) represents a conjugation between the CFG and the DVGs, containing both control flow and data dependencies, which is a complete form

of dependency representation through graphs. We will use multiple PDGs corresponding to each function of the program and include them in the knowledge graph.

We can represent the PDG as an oriented graph $G = (N, Ef, Ed)$, where N is a set whose elements are called nodes, Ef and Ed are both sets of ordered pairs of nodes whose elements are called edges, being Ef the set of flow edged and Ed the set of data edges.
In the PDG, a node will correspond to merging the nodes in the CFGs and DVGs so that N_i = <*(i, d, r, c, l)*> ∪ <*(i, d, v, b, l)*> = <*(i, d, r, c, v, b, l)*>, having the characters representing the same meaning as the latter kinds of graphs. The variables *i*, *d*, and *l*, being familiar with CFG and DVG, will be the glue points between the nodes to assemble the new one.

Differently from the dependence variable graph or the control flow graph, the PDG has two distinct types of edges. Both will be represented as tuples:

- If it corresponds to flow dependencies, they can be of the kind $Ef = (N_i, N_k)$, where N_k represents a node whose reachability can be obtained through the N_i node;

- If it corresponds to data dependencies, they will be of the kind $Ed = (N_i, N_k, V)$, where N_k represents a node that has at least one variable that was instantiated for the last time on N_i node, and V represents the exact variables which dependency is represented on that edge. In both types of edges, N_i can represent the unique return node, and so the logical end of the function.

A PDG will then be associated with a node of the knowledge graph. So all the navigation necessary for vulnerability detection will pass through them as we no longer need the DVGs or CFGs. Alongside the PDG, it is passed to the knowledge graph a list containing all labeling in its nodes, with the objective of determining which PDGs are worth evaluation for certain vulnerability cases (this will be more detailed further ahead, Section 3.2.6).

As shown in Fig. 3.4, the PDG contains two different types of edges. The dotted edges correspond to the same represented in the enriched DVG of Fig. 3.2 and define data dependencies where each edge is labeled with the name of the dependent variable (in this specific graph example, only *$fee* variable has dependencies). The dashed edges correspond to the ones represented in the CFG of Fig. 3.3 and illustrate the control flow dependencies.

The node content changes accordingly to the CFG and DVGs changes; the only difference is in circumstances where the nodes do not connect any data dependencies, and in that cases, the v and b are represented as question marks as they do not contain those values.

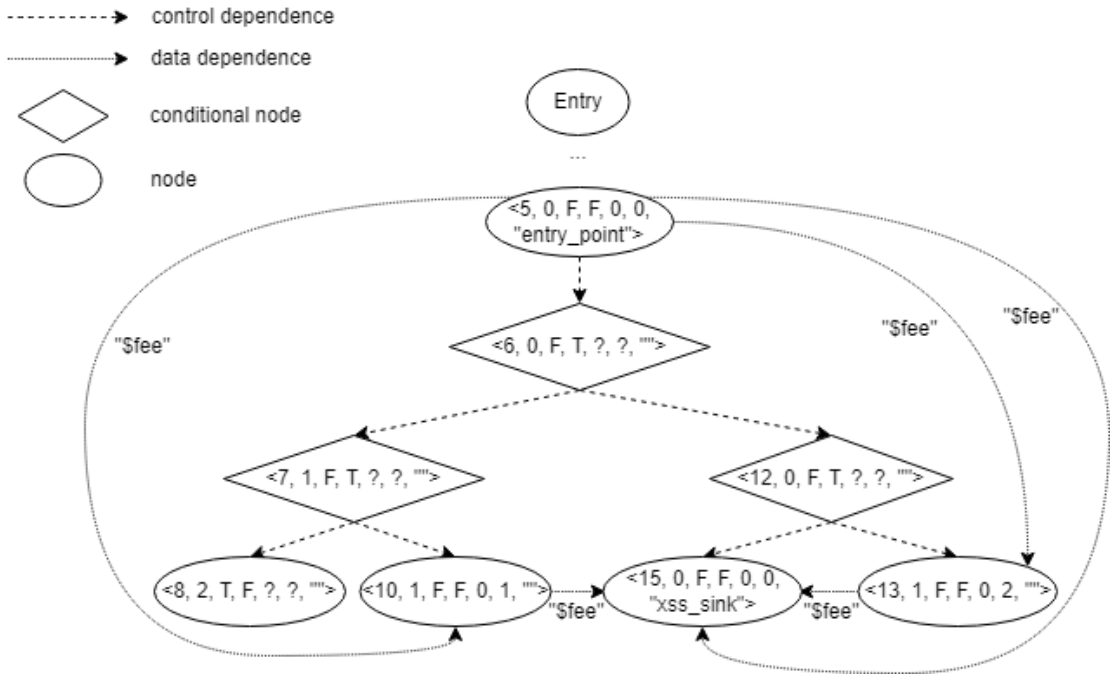

Figure 3.4: Enriched Program Dependency Graph example.

3.2.4 Function Call Graph

The Function Call Graph (FCG) is a specific kind of control flow graph that represents the call relations between program functions (we will focus on user functions for vulnerability detection purposes, although the graph is able to represent built-in functions as well), illustrating which functions can call others. In our approach, we will use FCG as a way to allow an inter-procedural analysis (between multiple functions) by combining it with the previously generated program dependency graphs. Due to its simple structure, it does not contain enough information to do any analysis, so combining it with other data structures is necessary.

After parsing the source code of the application, the FCG becomes an oriented graph $G = (F, C)$, where F is the set of nodes whose elements represent functions, and C is the set of edges, which represent ordered pairs of function calls whose elements are of the kind (F_i, F_k) being F_k the function that is being called in at least one procedure inside the function F_i (the callee).

3.2.5 Knowledge Graph

The Knowledge Graph represents the big schema of the application by merging the FCG with PGDs resulting in an enriched graph structure containing every piece of information necessary for a complete analysis. It will be the source structure that will be analyzed to find the vulnerabilities.

For the purpose of allowing inter-procedural analysis, the architecture of the KG is composed of a principal structure containing all the program functions (corresponding to the FCG structure) connected to multiple secondary structures with the individual procedures. So each node in the graph contains a pointer to a secondary graph corresponding to the PDG of the specific function. Furthermore, the edges represent function calls, and the nodes represent functions that can be labeled as they contain entry points, sensitive sinks or sanitization functions.

Specifically, we have a Knowledge Graph as an oriented graph where $G = (F, C, L, P)$, where:

- F is a set whose elements represent nodes corresponding to functions with procedure pointers to the program dependency graph of the respective function (from whose we are going to get the information when analyzing the KG);
- C represents the edges as a set of ordered pairs of function calls whose elements are of the kind (F_i, F_k), being F_k the function that is being called in at least one procedure inside the function F_i.
- L is a set of sets of labels that will be associated with each node in the KG. L_i is the set equivalent to the union of all labels in F_i.
- P is a set of PDGs where $P(F_i)$ is equivalent to the PDG corresponding to the F_i function.

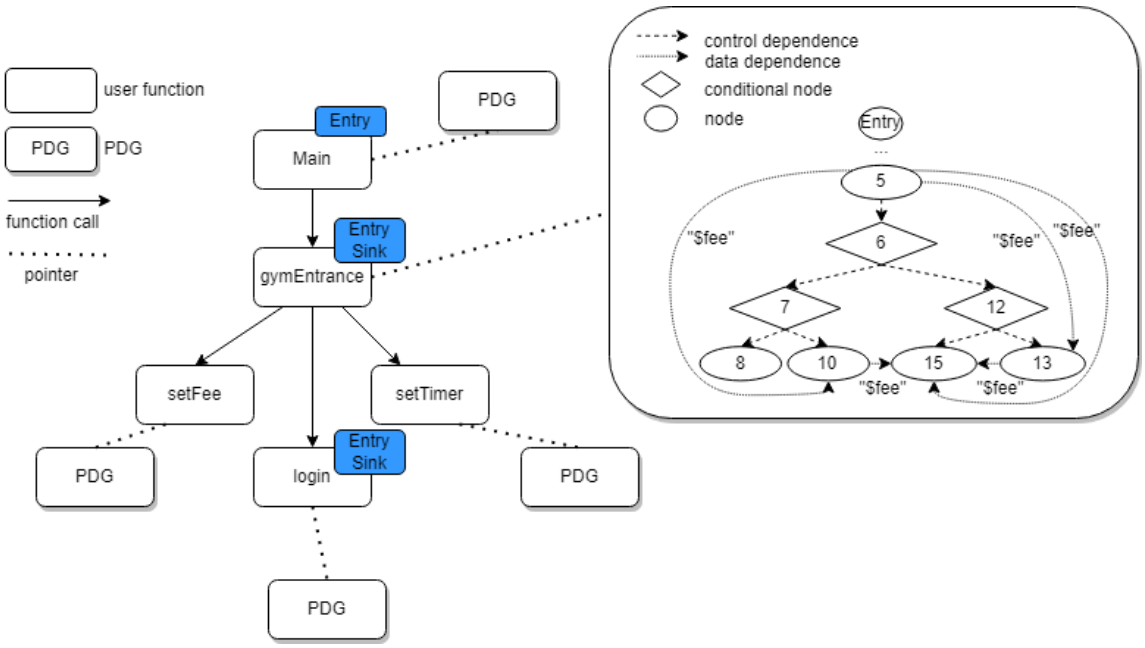

Figure 3.5: Example of the Knowledge Graph from the code of Figure 2.3.

The fact that the principal structure of nodes of the KG (corresponding to the CFG) is composed of pointers instead of the actual PDGs will allow us to be more efficient in the navigation since we can do the pruning at a superficial layer of the graph, and as we will not need to navigate through all the PDGs to reach specific targets that we will pre-define as potentially vulnerable. This will be further detailed in the Pruning and Vulnerability Detection sections (Section 3.3 and Section 3.4 correspondingly).

As we notice in Fig. 3.5, the graph shape will resemble the FCG body but with arrows to the PDGs corresponding to the different functions in the application (as a visual representation of the ordered pair concept). Both navigation in the principal structure and the individual ones are optimized via intelligent agents (see Section 3.4).

Other than containing the PDGs themselves, the nodes in the KG will also be labeled with the essential knowledge necessary for navigation and pruning. This information is previously received when the control flow and variable dependency graphs are created to be easily accessible from the beginning so that we do not require any initial passage through the KG to enhance it. More precisely, the KG graph contains all the labels inside each function, e.g., if they contain entry points, SQLi/XSS sensitive sinks, or SQLi/XSS sanitization functions.

3.3 Pruning

In our research, we emphasize the importance of pruning as a way to optimize the navigation in the knowledge graph by reducing the number of passages considerably and removing chunks of code that do not need analysis, saving computational power by disregarding them. At the end of the pruning, we will get a more navigation-efficient format of the knowledge graph and will save much running time on the process.

Our pruning solution works by checking the need for certain nodes to be in the knowledge graph. To make this evaluation, we need to take into consideration which is the content of the nodes; we do so not by analyzing the program dependency graph correspondent to the node but instead by utilizing the already associated labels and by that, saving us at least a passage for each individual graph.

We assembled a table in the following Table 3.1 containing the possible combinations of labels and correspondent decisions that we take in each scenario.

For pruning to happen, several factors are taken into consideration. Namely, we check if the node is a *Void*, meaning that the node corresponds to a function that does not call any other functions and simultaneously does not return any value. This is important because having both these characteristics allow us to infer that the function does not have an influence on other functions and so that it is isolated. Other than that, we check if the node contains any possible labels (entry point, SQLi/XSS sensitive sink, or SQLi/XSS sanitization function) as long as it contains any function entry parameters (arguments)

	Parameters	**Labels**			**Pruning Outcome**		
		Entry	Sanitization	Sink	Trim	Connector	Keep
Not Void						✓	
	✓					✓	
		✓				✓	
	✓	✓				✓	
			✓				✓
				✓			✓
	Other Combinations						✓
Void				✓	✓		
	✓				✓		
		✓			✓		
			✓		✓		
	✓	✓			✓		
	✓		✓		✓		
		✓	✓	✓	✓		
	✓	✓	✓		✓		
			✓	✓	✓		
				✓	✓		
	Other Combinations						✓

Table 3.1: Table of decisions for Pruning.

that would intuitively come from another region of the program.

As we can observe, each combination of contents can result in one of three possible outcomes. The node can either be *trimmed* and so entirely removed from the knowledge graph; turned into a *connector*, which will remain in the graph but just as a passage node where we need to consider the entry parameters and outputs but not the content of the function, and so we strip it of the program dependency graph; or we *keep* it as it is entirely relevant for the vulnerability analysis.

The most logical initial ramification factor we can contemplate is if the function is a *Void* since only then can we possibly trim it from the graph as it does not affect anything other than itself; if the function is not a *Void*, we will always need to keep it (even if as a connector) as it means information passes through it and although it may not reach a sink in the correspondent function, it can be reached in following ones.

This ponderation (choosing between *trimming*, turning into a *connector*, or *keep* it) is applied to every node in the graph except the correspondent to the main function whose existence is always necessary. After that, we check if the node contains any labels or entry parameters. If it does not carry anything, it can be trimmed/turned into a connector (depending on if it is a Void or not). Also, if it only contains either parameters, entry points, or both, it can be trimmed/turned into a connector as if it is Void we know they will not reach a sink in any other function, and if it is not a Void we can pass them along with the connector, skipping the PGD as we know inside of the function it will not reach

any sink or sanitization.

For Void functions, we can consider more specific scenarios, such as any combination including parameters, entry points and sanitization as connectors, as we do not care about sanitization if the variable will not reach a sink for sure, and also we consider the situation where the function has sinks not simultaneously with entry points or parameters as no sensitive information can be passed to the sink.

Any other combination will result in a non-pruning situation, and we must keep the nodes intact.

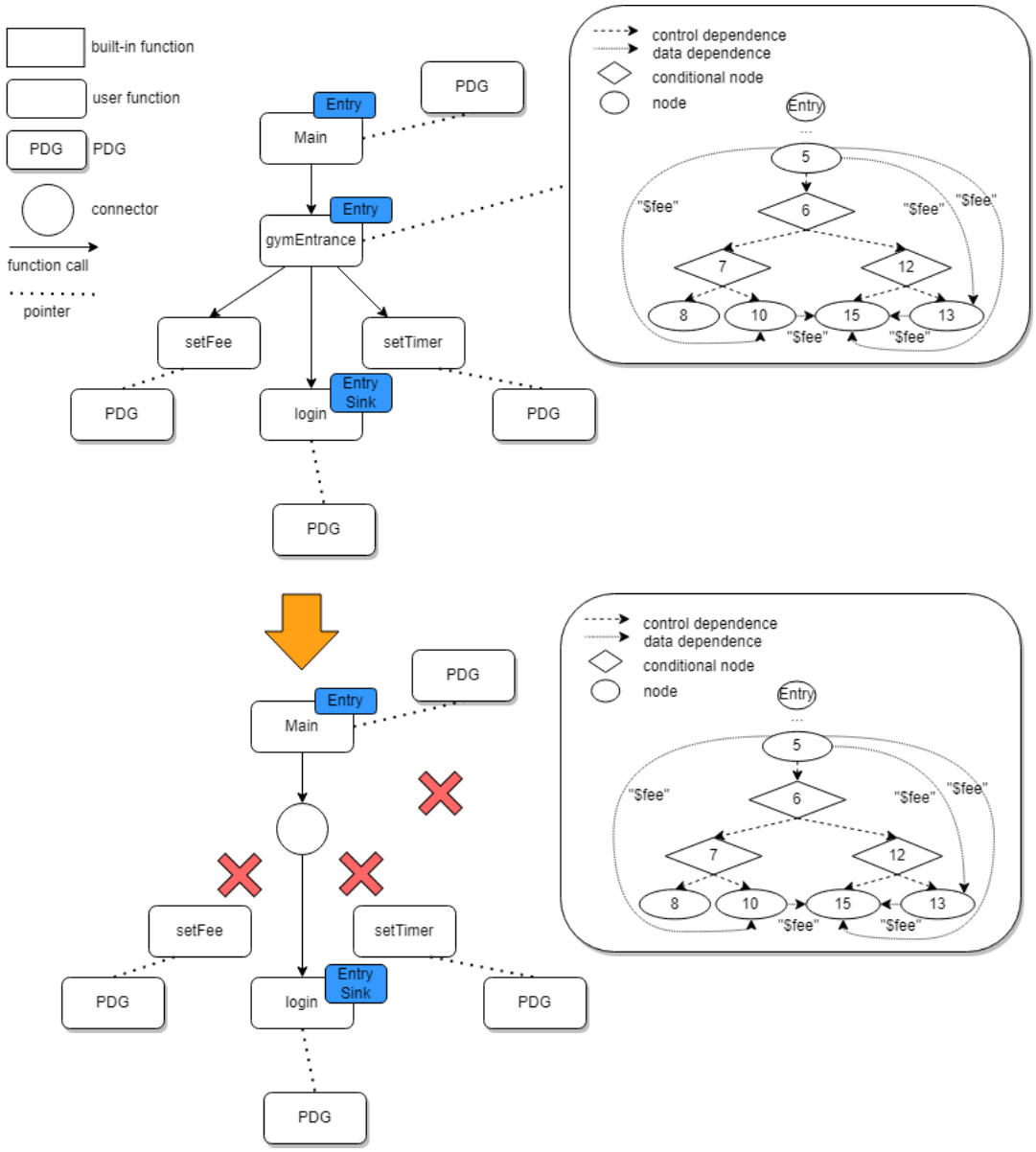

Figure 3.6: Pruning example for Fig. 2.3

As a result of the pruning, we get a much lighter version of the knowledge graph, which will save us a lot of running time in scenarios where we would need to otherwise

to evaluate every single program dependency graph. As we can observe in the example in Fig. 3.6 (based on code from Fig. 2.3 excluding line 15 to remove the sink from the function gymEntrance), many nodes are subject to being removed or adapted, allowing us to skip the ones turned into connectors and those corresponding to Void functions.

The pruning is a very significant improvement in efficiency, even if we only consider a single passage through the knowledge graph. However, knowing that for a complete analysis, we have to do multiple passages, it will be of even greater importance and only at the cost of superficially accessing the labels previously associated with the nodes.

3.4 Vulnerability Detection

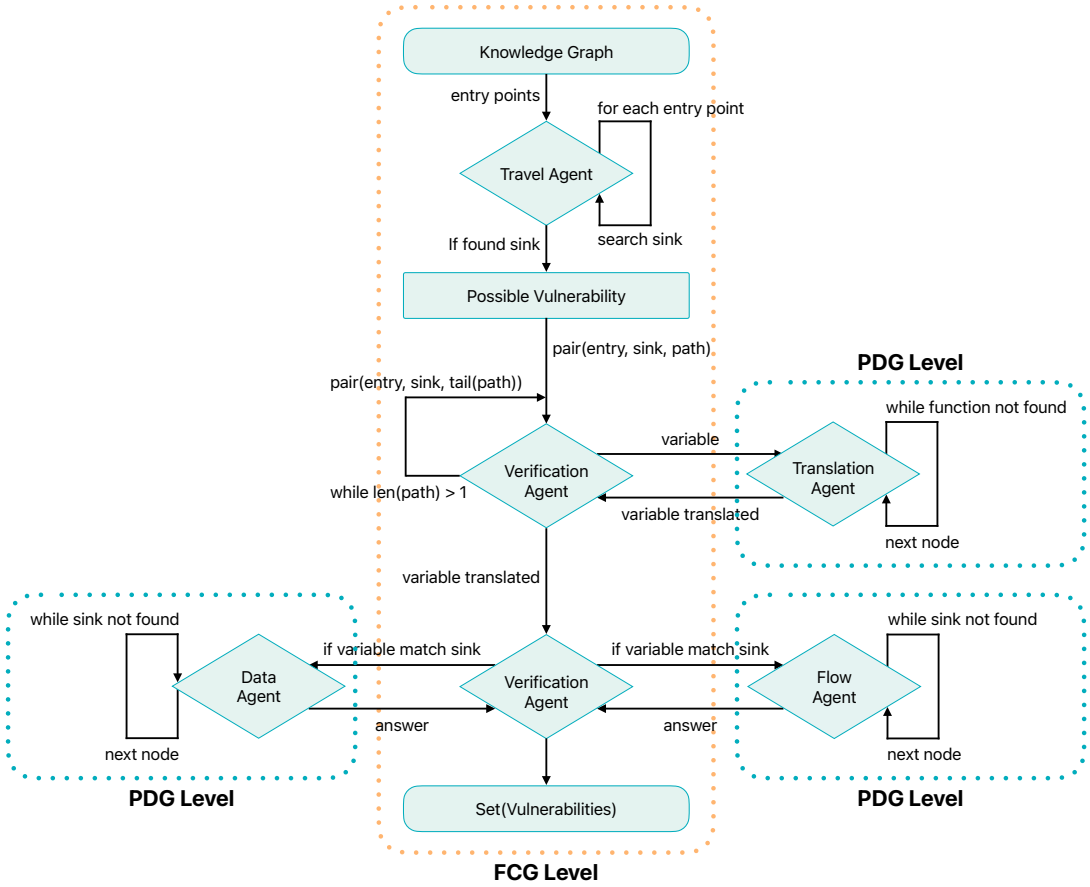

Figure 3.7: Multi-Agent System

In our approach, vulnerability detection is composed of an intelligent multi-agent system as a way to enhance overall navigation performance through the knowledge graph, specifically along the dimensions of computational efficiency. The agents in the system have several essential characteristics, such as being partially independent, self-aware, and autonomous. No agent has a global view of the system to keep it simple for an agent to exploit the knowledge of the graph. Since they are independent entities, communication

between them is necessary to allow collaboration, negotiation, and cooperation, all crucial features to keep the system robust.

As usual in a system like this, we propose different types of agents, each specialized in a distinct task so that they are objective and focused on only one goal each (travel, verification, a single translation, flow and data). The following sections describe each agent, how they communicate with each other, and how vulnerabilities are detected by them.

Figure 3.7 illustrates the multi-agent system with the four agents we created (Travel Agent, Verification Agent, Translation Agent, Flow Agent, and Data Agent) and the connections and decisions they make.

3.4.1 Travel Agent

The Travel Agent is the first agent that will be used in the system. Its objective is to make an initial superficial traverse through the graph in order to identify possible dangerous pieces of the program where a vulnerability might exist, e.g., a statement related to XSS or SQLi. This agent will only travel over the first layer of the knowledge graph (correspondent to the function call graph), never getting into the details of the program dependency graphs to keep it simple.

There will be one default agent of this kind for each node of the knowledge graph (used for communication) and one for each entry point label in each node (that will work as a trigger to start the communication). Following the example in Fig. 3.7, suppose we have a function that has three entry points labels, this means that we will have three travel agents in that function, one for each entry, plus one default agent that will be used for travel agents from other functions to communicate with.

The agent system will start navigating from these entry points marked positions with an initial travel agent that will communicate to the node successors' corresponding agent, requesting him to check whether he is labelled as having any sensitive sinks. This process will be repeated through all the possible function call paths in the knowledge graph, starting from the corresponding entry point node. Suppose he is able to find some node with a sensitive sink. In that case, it means that a path exists going from the entry to the sink (at least at the function call level), so it is also possible that it might represent a vulnerability. For now, we just talk about the possibility because being at a superficial level. It is still left to check by another kind of agent if it also exists a path connecting the entry and the sink via the program dependency graphs.

At the end of its life cycle, the travel agent will leave us with marked-out entry to sink pointers as well as giving information about the path of the functions that he went across to discover this connection. Having this path pointed out will allow the further travelling required to verify this possible vulnerability to be a directed search.

3.4.2 Verification Agent

The Verification Agent is the agent responsible for inferring if there is an actual flow and data path from the pairs of entries and sinks pointed out as possibly vulnerable by the travel agent.

With this in mind, one verification agent will exist for each node of the knowledge graph that is included in the path previously designed by the travel agent. Based on this path, there is the necessity of having agents for every single node.

Several navigation cycles will then occur, one for each possible vulnerability, starting with the agent indicated as containing the entry point that will generate a translation agent (better detailed below in Section 3.4.3), which will check how the variable associated with the entry point is referred to in the succeeding function of the path and give this information back to the verification agent. Once more, having previously recorded the path will allow us only to generate one translation agent for the direct successor of the path instead of all node successors of the graph.

The verification agent will then pass the acquired information to the corresponding verification agent successor in the path. This process will be repeated with the newly referenced translation until it reaches the agent in the last node of the path. It will then generate both a flow and a data agent (better detailed below in Section 3.4.4) to give us the final verdict about the possible vulnerability. The flow and data agents confirm if it exists flow and data dependencies accordingly. If neither or only one of them is inferred, then the variable is given as clear. Instead, if both exist, it is concluded that we are in the presence of a vulnerability.

3.4.3 Translation Agent

The Translation Agent is responsible for finding out how a variable referred to as coming from an entry point is translated on the following function of the designed path. This will require navigation at the program dependency graphs level, so we will have a translation agent for each node of the subgraphs included in the path.

The navigation starts with the agent associated with the variable previously passed from the verification agent. It communicates with all the successors that are both data and flow dependent from him (to ensure that exists a flow connection and the variable still lives in that path) successively until he reaches the agent where the desired function that he had been pointed on is called. When reaching this agent, the translation is performed, and the data is given to the verification agent. If a sanitization function is reached instead, the agent verifies if the variable corresponds to the one being sanitized and if so, he terminates the callings and gives immediate feedback to the verification agent, stating that the variable is clear.

3.4.4 Flow & Data Agents

The Flow Agent and the Data Agent functions are identical in every way, excluding the objective. They are responsible for the last step of the evaluation, the flow agent verifies if it exists a flow dependence between the entry and sink, while the data agent checks the data dependency.

Similarly to the translation agent, they perform at a program dependency graph level. However, instead of running in every subgraph included in the path, they only operate in the last node (function) of the path, which is also the node where the sensitive sink is contained, so that we will have flow and data agents for each node of this subgraph.
The navigation starts with the agent associated with the variable previously passed from the verification agent, which coincides with the last translation made by the translation agent. They communicate with all the successors that are flow-dependent (for the flow agent) and data-dependent (for the data agent) successively until they reach the agent where the sensitive sink is on.

The agents then verify if the entry variable corresponds to the one present in the sink. If so, they give feedback to the verification agent that the pair of entry-sink is flow/data dependent (for the flow and data agents accordingly) so that he can infer the vulnerability exists in case it is both flow and data dependent. Otherwise, they give it as non-dependent, meaning that the variable is clear.

If a sanitization function is reached instead, he makes the same procedure as the translation agent giving feedback to the verification agent stating that the variable is clear.

Chapter 4

Implementation

This chapter presents the implementation of our proposed solution that sustains the approach described in Chapter 3. For the solution to be as simple and efficient as possible, we chose to use Python due to its already well-developed graph frameworks.

Section 4.1 describes the tools and packages we used for graph creation and manipulation. Section 4.2 presents our algorithm implementation of the property graphs needed for creating the knowledge graph. Section 4.3 explains how we assembled all the property graphs into the unique knowledge graph. Section 4.4 details how we performed the pruning over the graph for efficiency purposes. Section 4.5 illustrates how the intelligent multi-agent system was implemented for the agents to cooperate in giving the final result of the evaluation.

4.1 Tools and Packages

4.1.1 Function Call Graph

In contrast to the other types of graphs, where we had to create our own methods due to the absence of pre-existing tools or their inadequacy, there are some available tools for generating the PHP function call graph. These tools' existence, combined with the fact that we do not need to add any extra information to the function call graph, led us to decide that its implementation was unnecessary and that we could take advantage of the already existing software.

Thus we choose to use the phpCallGraph [38] tool to generate the static FCG from the PHP source code. This resulting graph results from parsing call functions instructions into a DOT file, which is a widespread format to represent graphs, although not the easiest to manipulate, so we transform it using the NetworkX [36] package for Python, allowing more extensive control over the data. As we will explain in the following sections, NetworkX will be used as a base for all our graph creation algorithms, so it is ideal that we can use this to deal with the FCG.

4.1.2 NetworkX

NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks [36]. Since every graph we create is enriched with extra information, we need a highly adaptable structure to host them. NetworkX is one of the most flexible packages for graphs in Python, so it was an optimal choice for our approach implementation.

Some of the advantages we look for include:

- More than just simple graphs, allowing the creation of directed graphs and multigraph systems, which is ideal since we need graph orientation to represent the dependencies and also our graphs are not always connected, so we need a structure that allows multiple graphs associated with the same variable, e.g., in the dependency variable graphs when a chain of dependencies is broken (Figure 3.2);
- Nodes can be virtually anything so we can associate as much information given in any format with them;
- Edges can hold arbitrary data so we can differentiate them, e.g. in the program dependency graph, we can distinguish flow from data dependencies as well as tag the data dependencies with the respective dependent variable.
- It can import data from DOT format, the format in which the function call graph is generated, so we can convert it to include it in the knowledge graph;
- It allows fast prototyping as well as a possibility to visualize the graphs, which is important to expedite the development process.

4.2 Property Graphs Algorithms

This section presents the algorithms that describe the implementation of the property graphs functions and explains how they work.

4.2.1 Dependence Variable Graph

Algorithm 1 describes the main highest-level steps performed by the DVG parser. It receives the source code relative to the function that will be parsed into the multiple DVGs', one for each variable contained in the code.

The algorithm starts by initializing an empty dictionary *variables* that will store the information necessary for the DVG construction for each variable. Also, it initializes a list *ramifications* only with zeros (0's) representing the current ramification index of a conditional statement and, finally, a numeric variable *layer* that represents the index of the current relative depth of the statement we are analyzing.

Algorithm 1 Dependence Variable Graph Main Algorithm

Input:
file: source code of the function to get the DVGs

Output:
graphs: list of DVGs of each variable

```
1: variables ← {}
2: ramifications ← [0]
3: layer ← 0
4: for each line ∈ file do
5:     line ← line.rstrip()
6:     if "}" ∈ line then
7:         layer -= line.count("}")
8:     end if
9:     if "else" ∈ line then
10:        ramification[layer] += 1
11:    end if
12:    if "$" ∈ line then
13:        i = line.get_index()
14:        varss ← get_vars(line)
15:        label ← get_labels(line, i)
16:        for var ∈ varss do
17:            if var ∉ variables then
18:                variables[var] ← [(i+1, 0, layer, ramification, label)]
19:            else
20:                version = variables[var].get_version()
21:                if redifined(var) then
22:                    variables[var].append((i+1, version+1, layer, ramification,
   label))
23:                else
24:                    variables[var].append((i+1, version, layer, ramification,label))
25:                end if
26:            end if
27:        end for
28:    end if
29:    if "{" ∈ line then
30:        layer += line.count("{")
31:    end if
32: end for
33: graphs ← []
34: for each var ∈ variables do
35:    g ← nx.MultiDiGraph()
36:    nodes ← var.get_nodes()
37:    for n ∈ nodes do
38:        for k ∈ nodes[n:] do
39:            if k.is_dependent(n) then
40:                g.add_edge(n, k)
41:            end if
42:        end for
43:    end for
44: end for
```

After the initialization, for each line of code in the function we are analyzing, we clear the line of blank spaces on the right side and then check if it contains closing braces (}), meaning that the depth has decreased and so we decrease the *layer* variable, we also check if it contains an $else$ statement so that we increase the conditional *ramifications* corresponding to the layer we are at the moment (lines 5 to 11).

After updating these variables, we move to the more important step, checking if the line contains a variable by checking if it contains the $ char. If so, we start collecting the variables information necessary for the graph creation, starting with the index inside the line it belongs to, then we call the method *get_vars* that gives us a list of all the variables in the line and assign them to a variable *varss*, we also call the method *get_label* that checks if the line contains any "entry_point", "sqli_sanitization", "xss_sanitization", "sqli_sink" or "xss_sink" so it assigns the corresponding designation to a variable *label* if any of this is present (lines 13 to 15).

Having the variables from the respective line, we check for each of them if it is already present in our *variables* dictionary. If so, we add at the end of the respective list the information for this specific occurrence of the variable, namely, we add a tuple that contains the index of the line where the variable appears, 0 as being the first instance where the variable is used, the corresponding conditional ramification and the label previously assigned. If it is not the first instance of the variable, we check if it is being reassigned or just used for some other purpose; if it was reassigned, we save its information similarly as previously described, but instead of 0 in the second element of the tuple we increment the version of the variable; otherwise, we keep the same version of its last instantiation (lines 16 to 24).

At this point, we already have all the information necessary about all the variables in the code. So we proceeded to create the corresponding DVGs', starting by creating an empty list *graphs* where we will save them (line 33).

For each variable we saved in *variables*, we create a list *repeated* where we save all the line indexes where the variable has been used. Then we create an empty *MultiDiGraph* (Multiple Directed Graph) from the networkx package that we assign to a variable *g*. Also, we create a variable *nodes* that contain where we assign all the instances of information previously acquired of the variables (lines 34 to 36).
Finally, each of these instances are converted into nodes and connected by edges in the graph *g* if they represent data dependencies between each other. To verify this, we use the conditions (represented by the function *is_dependent* on line 39) so that it does not create edges for nodes that are on the same layer and are in different conditional ramifications or have different versions. Also, they cannot have the same version or be twice on the same line, meaning that it was not redefined and, since it is on the same line, do not depend on one another (lines 37 to 39).

In the end, we add the *g* graph representing the DVG to the current variable to the

graphs list (line 40). When all the variables are analyzed, *graphs* will contain all the DVGs associated with the variables in the respective source code.

4.2.2 Control Flow Graph

Algorithm 2 describes the main highest-level steps performed by the CFG parser. It receives the source code relative to the function that will be parsed into the corresponding CFG.

The Algorithm starts by initializing an empty list *nodes* that will store all the graph nodes and a variable *depth* that will represent the relative depth at each moment, starting with 0 (lines 1 and 2).

After the initialization, for each line of code in the function we are analyzing, we clear the line of blank spaces on the right side and save the result in a variable *line*. Then we call the *get_label* function (previously described in the DVG section) that will return us the labelling of the line, and we save it on a variable *label* (lines 3 to 5).

Finishing these initial steps, we check if the line is not empty or if it only contains braces. If it is not empty, it contains essential information for the CFG, and we turn the line into a node and add it to the *nodes* list. Each node will contain the index of the line, its relative depth, if it is a return statement, if it is a conditional statement and its label. If it contains closing or opening braces, we decrease or increase the depth accordingly (lines 6 to 13).

Then we create an empty *MultiDiGraph* where we will build the CFG and assign it to a variable *g*. We will compare each node in the *nodes* list and check if the depth is the same, meaning that it is the only flow possibility, and if so, we will add a new edge to the CFG between them, otherwise it means we got inside a conditional statement or a cycle, and so we have multiple flow possibilities that we have to review (lines 16 to 20).

So instead of comparing with the immediate next node, we inspect all the following nodes for flow dependencies until we reach either the end of the *nodes* list, a node with a smaller depth than the current one or a node with a return statement. For the others, we define different criteria for the flow dependencies (lines 22 to 41):

- if the current node depth is -1, then the forward one got inside the condition or the cycle, we add an edge between them, and we do not add any more edges considering this criterion, but keep checking the others;
- if the node is the same depth as the forward one it got outside the condition or the cycle, so we add an edge between them, but we do not add any more edges to the current node;
- if the current node depth is +1 than the forward one it got outside the condition or the cycle and the forward statement is not a conditional statement we add an edge between them, but we do not add any more edges to the current node;

Algorithm 2 Control Flow Graph Main Algorithm

Input:
file: source code of the function to get the CFG

Output:
g: CFG correspondent to the source code

```
1:  nodes ← []
2:  depth ← 0
3:  for each line ∈ file do
4:      i ← line.get_index()
5:      label ← dvg.get_label(line, i)
6:      if not line.is_empty() then
7:          nodes.append(i+1, depth, returnStatment, conditionalStatment, label)
8:      end if
9:      if "}" ∈ line then
10:         depth -= line.count("}")
11:     end if
12:     if "{" ∈ line then
13:         depth += line.count("{")
14:     end if
15: end for
16: g ← nx.MultiDiGraph()
17: for each node ∈ nodes do
18:     nextNode ← node.get_next()
19:     if node.depth() == nextNode.depth() then
20:         g.add_edge(node, nextNode)
21:     else
22:         while nextNode != null  &  node.depth() ≤ nextNode.depth() do
23:             if node.is_return() then
24:                 break
25:             end if
26:             if node.depth()+1 == nextNode.depth() then
27:                 g.add_edge(node, nextNode)
28:                 stop_criteria()
29:             end if
30:             if node.depth() == nextNode.depth() then
31:                 g.add_edge(node, nextNode)
32:                 break
33:             end if
34:             if node.depth() == nextNode.depth()+1  &
    nextNode.is_conditional() then
35:                 g.add_edge(node, nextNode)
36:                 break
37:             end if
38:             if node.depth() == nextNode.depth()+1 then
39:                 stop_contitionalCriteria
40:             end if
41:             nextNode = nextNode.next()
42:         end while
43:     end if
44: end for
```

- last if the forward node has a conditional statement it breaks the conditional chain so from now on it only links to nodes outside the chain (nodes with -1 depth).

After these criteria are applied, we end up with the full CFG in the *g* variable and return it.

4.2.3 Program Dependency Graph

Algorithm 3 Program Dependency Graph Main Algorithm

Input:
file: source code of the function to get the PDG

Output:
pdg: PDG correspondent to the source code
labels: list of the labels in the PDG
ret: list of return statements in the PDG

```
1:  pdg ← cfg.to_cfg(file)
2:  dvgs ← dvg.to_dvg(file)
3:  for each dvg ∈ dvgs do
4:      for each edge ∈ dvg.edges() do
5:          if pdg.has_edge(edge) then
6:              new_label ← edge.getLabel().add(dvg.getVariable())
7:              pdg.remove_edge(edge)
8:              pdg.add_edge(edge, "green", new_label)
9:          else
10:             label ← dvg.getVariable()
11:             pdg.add_edge(edge, "red", label)
12:         end if
13:     end for
14: end for
15: labels ← pdg.get_labels()
16: ret ← pdg.get_returnStatements()
```

Algorithm 3 describes the PDG parser as connecting the previously implemented CFG and DVGs'. Starting by creating a variable pdg, where we will build over the PDG, and assign it the CFG as a base structure by calling the previously described (Section 4.2.2) function to_cfg over the function to be analyzed. Also, we call the function to_dvg (Section 4.2.1) to get a list of all the DVGs associated with the function variables and assign it to a variable dvgs (lines 1 and 2).

Our PDG will have three different kinds of edges that we will differentiate by associating different colors to each of them: flow dependency edges will be black by predefinition, data dependency edges will be red, and edges with both data and flow dependencies will be green. Each edge that contains data dependencies will be labeled with the corresponding variables that are dependent.

As we are taking the CFG as a base graph, we already have the flow dependency edges, so we will have to iterate over each of the DVGs' and add its edges to the PDG. For each edge of each DVG, we will check if it already exists in the PDG, meaning that it has both flow and data dependencies, so we remove the current flow edge and replace it with a new edge with both dependencies while also updating the label in the edge to contain the new variable correspondent to the current DVG (lines 3 to 8). If it does not already exist, we add it as a data dependency edge (lines 10 and 11).

After the PDG is fully built, we create two new variables labels to save all the labels in the function to ease the development of the knowledge graph and ret that will be a boolean stating if the function has return statements to distinguish flow void functions (lines 15 and 16).

4.3 Knowledge Graph Assemble

Algorithm 4 represents the program's highest-level algorithm, as it implements the Knowledge Graph and takes advantage of all the previously implemented graph algorithms. The KG itself is a combination of the FCG and a program's multiple PDGs.

To get the FCG, we use the phpCallGraph tool to generate the static FCG from the PHP source code. As we explained in Section 4.1.1 we use the Networkx package to transform the dot file in a more understandable and easier format, and we assign it to a variable fcg (line 1).

Getting the PDGs is a more complex task as there will be multiple PDGs for a single source file, so we need to isolate the distinct function's code to convert it into the corresponding PDG. To do this, we start by creating two lists: file, where we store the source code split by lines, and nodes, where we store the names of all the functions in the code that we can take from the FCG nodes. We also create two empty dictionaries: kg, where we will further add the PDGs and functions, where we will add the already isolated function code (line 2 to 5).

So for each function name in nodes, we create a numeric variable stack that will be used to measure the relative depth inside the function and a list function where we will store the function lines. Then for each line in the file, we will search if it contains the string "function" succeeded by the function name, meaning that it is the beginning of its definition, and so we set stack to 1 and append the line to function. We proceed to append the following lines until we reach a point where stack is equal to 0, indicating that we reach an inferior depth in the program compared to the function definition, and so we get out of it. The stack is updated at each line by adding the number of opening braces and deducting the number of closing braces (lines 6 to 17).

When reaching the 0 stack point, we have all the function lines inside function, so we add it to the functions dictionary and, at the same time, call to_pdg function to convert it

into a PDG and add it to the kg (lines 19 to 22).

Algorithm 4 Knowledge Graph Main Algorithm

Input:
file: source code of the file to get the KG

Output:
kg: KG correspondent to the source code

```
1:  fcg ← networkx.graph_from_source(source_file)
2:  file ← open(source_file)
3:  nodes ← cfg.get_nodes()
4:  kg ← {}
5:  functions ← {}
6:  for each node ∈ nodes do
7:      stack ← -1
8:      function ← []
9:      for each line ∈ file do
10:         if line.has_functionDefinition() then
11:             stack ← 1
12:             function.append(line)
13:         end if
14:         if stack ≥ 0 then
15:             stack += line.count("{")
16:             stack -= line.count("}")
17:             function.append(line)
18:         end if
19:         if stack == 0 then
20:             functions[node] ← function
21:             pdg ← pdg.to_pdg(function)
22:             kg[node] ← pdg
23:         end if
24:     end for
25: end for
```

4.4 Pruning

The pruning implementation is spread throughout our solution for efficiency purposes instead of having its own algorithm. Its highest level of verification is at KG level, where even before adding a PDG to the KG, we verify if it has any entry point, sensitive sink, sanitization function or if it is void as not returning anything.

Although the verification is made at this level, we use the values of the labels and the boolean saying if it is void returned from the PDG level that, in its turn, received the labels from the CFG and DVG at its creation time. Handing all this information at

creating allows us to avoid further passages over any of the previously created structures to acquire it, saving us time and computational power.

Having all this information available, we follow the approach of Table 3.1 to do the pruning, so if we decide to trim it, we remove the function node from the FCG and do not even add his PDG to the KG, if we decide it will be a connector we do not add its PDG to the KG but instead we add a connector node, or if we decide to keep it we add its PDG to the KG.

4.5 Multi-Agent System

Algorithm 5 Multi-Agent System Main Algorithm

Input:
kg: KG to be analyzed
Output:
vulnerabilities: the list containing all vulnerabilities found

```
1:  possible_vulnerabilities ← []
2:  for each function ∈ kg do
3:      if function.has_labels() then
4:          for each label ∈ function.get_labels() do
5:              if label.is_entryPoint() then
6:                  possible_vulnerabilities += travel_agent(fcg, kg, function, entry_point)
7:              end if
8:          end for
9:      end if
10: end for
11: vulnerabilities ← []
12: for each trial_vul ∈ possible_vulnerabilities do
13:     vulnerabilities.append(verification_agent(trial_vul))
14: end for
```

Concluding the construction of the KG already trimmed, we are positioned to find vulnerabilities, which is the main objective of the multi-agent system we implemented. Starting with its highest level, we create an empty list possible_vulnerabilities where we are going to store pairs of entry points and sensitive sinks that possibly represent vulnerabilities.

Then for each function of the KG, we check if it has any associated labels. If so, we isolate the entry points, and for each one of them, we create a travel agent to return as a list of sensitive sinks it can possibly reach and append it to possible_vulnerabilities. By separating possible vulnerabilities, we avoid travelling through all the graph's possible paths but instead, we focus on just these specific ones.

Having already the possible vulnerabilities, we create another empty list vulnerabilities where we will store the actual vulnerabilities we find. Next for each possible vulnerability, we will create a verification agent that will verify and confirm if it represents, in fact, a vulnerability, and we append it to vulnerabilities. Finally, we end up with a list of all the vulnerabilities found.

4.5.1 Travel Agent

Algorithm 6 Travel Agent Main Algorithm

```
1:  function TRAVEL_AGENT(fcg, kg, path, entry_point)
2:      var ← []
3:      current_function ← path.last()
4:      var.append(entry_point, kg[current_function].get_sinks(), path.copy())
5:      for each successor ∈ fcg.successors(current_function) do
6:          if successor ∉ path then
7:              path.append(successor)
8:              var.extend(travel_agent(fcg, kg, path, entry_point))
9:          end if
10:     end for
11:     return var
12: end function
```

The travel agents are responsible for finding all the paths between an entry point and sensitive sinks while keeping track of the path between them.

To do so, we create an empty list var and start by appending a tuple with the entry point, all the sensitive sinks in the function corresponding to the last element of the current path, having in mind that in the first iteration, the travel agent receives as the path a list with only one element correspondent to the same function where the entry point is, so the last element will also be the same function, at last, we add a copy of the current path, so we know the path between the entry and sinks.

Then it needs to check the other possible paths, so we get the current function successors, append each one of them to the path, and create a new travel agent with the updated path, so it has the same behaviour, but when getting the last element of the path, he will get the sinks from the new function added to it.

This will happen until we reach a function with no successors, so the last travel agents will return the updated lists of possible vulnerabilities to the ones that created them and finally reach the initial agent that will return the complete list of sinks and paths for that specific entry.

4.5.2 Verification Agent

The verification agent's main objective is to verify whether pairs of entry points and sensitive sinks represent concrete vulnerabilities. It mainly operates as a higher-level manager for translation, flow, and data agents. It receives the entry-sink pairs previously established by the travel agent, including the path it should follow to connect them, this is optimal since it makes the search oriented.

A verification agent will contact other verification agents along the path it received while calling translation agents between each function (lines 18 to 23) to translate the different nominations of the same variable along the way until he reaches the last element of the path ending up with a chain of translations that will allow tracking all the variable trail between entry and sink.

When reaching the last function in the path (lines 7 to 15), it will invoke both a flow and a data agent that provide information about whether the entry and sink are flow and data dependent, respectively. Ultimately the verification agent will return whether the pair corresponds to a vulnerability or not based on this information.

Algorithm 7 Verification Agent Main Algorithm

```
1:  function VERIFICATION_AGENT(entry_point, sink, path, current_var)
2:      if not path then
3:          return False
4:      end if
5:      current_function ← path[0]
6:      pdg ← kg.getPDG(current_function)
7:      if path.length() == 1 then
8:          if entry_point.getFunction() == current_function then
9:              start_node ← pdg.getNode(entry_point.getIndex())
10:             target_var ← entry_point.getVariable()
11:         else
12:             start_node ← pdg.getNode(0)
13:             target_var ← current_var
14:         end if
15:         return data_agent(pdg, target_var, sink, start_node) &
    flow_agent(pdg, target_var, sink, start_node)
16:     end if
17:     next_function ← path[1]
18:     if entry_point.getFunction() == current_function then
19:         start_node ← pdg.getNode(entry_point.getIndex())
20:         translations ← translation_agent(pdg, start_node, current_var, )
21:     else
22:         start_node ← pdg.getNode(0)
23:         translations ← translation_agent(pdg, start_node, current_var)
24:     end if
25:     return verification_agent(entry_point, sinks, path.tail(), translations)
```

4.5.3 Translation Agent

Algorithm 8 Translation Agent Main Algorithm

```
1: function TRANSLATION_AGENT(pdg, node, variable, function, next_func,
   translations ← [], visited ← []):
2:     successors ← current_pdg.successors(current_node)
3:     if not successors then
4:         return clean
5:     end if
6:     for each successor_node ∈ successors do
7:         line ← function.get_line()
8:         if next_func ∈ line & successor_node ∉ visited then
9:             possible_vars ← dvg.get_vars_func(line, next_func)
10:            if variable ∈ possible_vars then
11:                translations.append(next_func.get_variableName(variable.index()))
12:            end if
13:            visited.append(successor_node)
14:            translation_agent(pdg, node, variable, function, next_func,
   translations, visited)
15:        end if
16:    end for
17:    return translations
```

The translation agent has the role of checking how a variable denomination is translated between two functions. To do so, it needs to check the flow of the variable through the PDG of the callee function until it reaches the called function and checks all the transformations it might suffer along the way.

Therefore, starting from the node where the variable was created, for each of its successors that are not return statements, we check if it contains the callee function (we know this function since it is on the path sent by the verification agent). If the callee function is in the line, we get all the vars nominations passed to it by calling the function *get_vars_func* from the *dvg* class and saving the result in a variable *possible_vars*. We check if the variable that we want to translate is in *possible_vars*, and if so, we call again the *get_vars_func* this time on the node corresponding to the callee function definition to get its variables nominations and get the one with the same index as the variable we are currently translating to know how it will be called next in the chain of translations. So we append it in list *trans*, which will save the chain of translations of that specific variable.

After visiting a node, we save it on a list *visited*, so we do not revisit it in case there are cycles on the PDG.

Finally, we call a new translation agent over the following function to repeat the process until reaching a function that does not pass the variable anymore, so we end up with a complete chain of translations of that variable to further check if it reaches a sink with the evaluation agent.

4.5.4 Flow Agent

Algorithm 9 Flow Agent Main Algorithm

```
1:  function FLOW_AGENT(current_pdg, sink, current_node, visited ← []):
2:      successors ← current_pdg.successors(current_node)
3:      if not successors then
4:          return False
5:      end if
6:      for each successor_node ∈ successors do
7:          if edge_type(current_node, successor_node) == "flow_edge" &
    successor_node ∉ visited then
8:              if successor_node.get_label() == sink then
9:                  return True
10:             end if
11:             visited.append(successor_node)
12:             return flow_agent(current_pdg, sink, successor_node, visited)
13:         end if
14:     end for
15:     return False
```

The flow agent is responsible for determining whether a flow dependence relation exists between the entry and the sink. Since it is only called in the last iteration of the verification agent, it only operates over the PDG associated with the last function of the path (*current_pdg*), which correspondents do the function where the sink is located.

Depending if the entry is also on the same PDG, it will start the navigation by its corresponding node (*current_node*); otherwise, it will start by the node corresponding to the function definition, which we previously know is reachable from the entry since it was led to us by the verification agent enquiry of the translation agent communication chain.

Having this set, we start going throw the node *successors* (line 2) consecutively until we either reach the node corresponding to the *sink* (line 8) or we reach a point where there are no more *successors* (line 3), meaning that there is no flow dependency, and so returns *False* to the verification agent.

To achieve this, for each *successor_node* (line 6), the agent will verify if the edge connecting it to the *current_node* is a *"flow_edge"* (that includes edges that are both data and flow as we represent them differently on our model), implying that there exists a flow dependency between them. It also checks if we have not visited that node before to avoid loops (line 7). If these conditions are not met, it will stop that branch's navigation as it means an interruption in the dependency chain.

Suppose the flow dependency between the nodes is present. In that case, it checks if the *successor_node* corresponds to the *sink*, which would indicate that there is, in fact, a flow dependency between the entry and sink, so the flow agent returns *True* to the verification agent. Otherwise, it will append the *successor_node* to the *visited* list and call

a new flow agent to proceed with the navigation from the *successor_node* (lines 8 to 12).

4.5.5 Data Agent

Algorithm 10 Data Agent Main Algorithm

```
1:  function DATA_AGENT(current_pdg, target_var, sink, current_node, visited ← []):
2:      successors ← current_pdg.successors(current_node)
3:      if not successors then
4:          return False
5:      end if
6:      for each successor_node ∈ successors do
7:          if target_var ∈ edge_data(current_node, successor_node) & successor_node ∉
    visited & edge_type(current_node, successor_node) == "data_edge" then
8:              if successor_node.get_label() == sink then
9:                  return True
10:             end if
11:             visited.append(successor_node)
12:             return data_agent(current_pdg, target_var, sink, successor_node, visited)
13:         end if
14:     end for
15:     return False
```

The data agent is responsible for determining whether it exists a data dependency relation between the entry and the sink. Its algorithm is very similar to the flow agent algorithm as they both operate over the PDG corresponding to the last function of the path (*current_pdg*). The main difference remains on the edges that they go throw. While the flow agent travels throw the *"flow_edges"*, the data agent travels throw the *"data_edges"*.

So it will start from the *current_node* and proceed to its successors (line 2) until it reaches a node that has no successors, in that case returning *False* to the verification agent as he did not find a data dependency to the *sink* (line 3), or it reaches the node corresponding to the *sink* (line 8).

To be able to find the path to the *sink* while ensuring that the data dependency has been kept, for each *successor_node*, it will be checked if the edge that connects it to the *current_node* is a *"data_edge"* which included the *target_var* (the tainted variable) in its label, this extra verification is necessary since we have to make sure that the dependency is related to a variable associated with the entry point, or to the entry point itself (in this case, the *target_var* would be the entry function instead of a variable). It also checks if the *successor_node* has not been visited before to avoid loops (lines 6 and 7).

If these conditions are met, it will check if the *successor_node* corresponds to the node containing the *sink*, confirming that the data dependency exists, and so it returns *True* to the verification agent (lines 8 and 9); otherwise, it will append the *successor_node* to the *visited* list and call the data agent in the *successor_node* (lines 11 and 12).

If the conditions are not met, it means that there is no more dependency, and so it will kill that branch by returning *False* (line 15).

Chapter 5

Evaluation and Validation

In this chapter, we will delve into the various processes used for evaluating and validating KAVe and the key considerations and challenges involved in these processes.

Both tool validation and evaluation are crucial steps in ensuring the reliability and validity of the results. Tool validation implicates verifying that the tool meets the requirements and specifications for its intended use, while evaluation concerns assessing the performance and effectiveness of the tool.

To properly evaluate KAVe, it is important to consider the challenges outlined in Section 3.1 and the solutions proposed to address them. With them in mind, we defined the following questions which we will reason about:

1. Is it able to generate the property graphs?
2. Can it merge the property graphs into the knowledge graph?
3. Is it able to do the pruning over the knowledge graph?
4. Is the multi-agent system able to navigate efficiently over the knowledge graph?
5. Can KAVe in fact find vulnerabilities?
6. Is the KG resilient enough to represent web applications?
7. How does it perform compared to other existing static analysis tools?

The chapter will be divided as follows: Section 5.1 describes the evaluation setup. Section 5.2 characterizes the web applications we analyzed. Section 5.3 presents and discusses the experiment results and answers the previously established questions (numbered from 1 to 5). Section 5.4 makes a comparison of the KAVe performance with other existing tools answering questions number 6 and 7.

5.1 Evaluation Setup

The purpose of the evaluation setup section is to provide a detailed description of how to run the tool to be able to detect vulnerabilities in web applications and validate them. So we follow the subsequent procedure:

1. Select the web applications which we will use to test KAVe. A good candidate application to test is an application which we already know contains vulnerabilities. This will allow us to verify if the tool can accurately detect the vulnerabilities that we know already exist. Therefore, we can validate KAVe for the functionalities it was developed.

   It can also be helpful to select web applications that other tools have already tested. This will allow us to compare the results of our tool with the results of other tools, which can give us a sense of how effective our tool is compared to others, as well as help us identify any unique vulnerabilities that our tool can detect that other tools may have missed.

   Ultimately the goal is to select web applications that will provide a thorough and accurate test of our tool's capabilities while also allowing us to compare our results with those of other tools.

2. Generate the .dot files containing the FCGs' utilizing the phpCallGraph tool [38]. Unfortunately, this tool can not be integrated automatically into our system, so the FCGs' must be created prior to the analysis. In the future, we can create our own tool to facilitate this process.

3. Run the tool to detect the vulnerabilities over the chosen web applications and the correspondent FCGs. More specifically, this includes all the processes of generating the CFGs and DVGs and merging them into PDGs to further link with the FCG and additional information, such as labels and differentiated edges, to create a final KG. It also includes KG pruning and running the multi-agent system over the KG to end up with the vulnerabilities.

   This process is automatic and represents the primary usage of the tool, which must be repeated to each PHP file of each web application. In the future, the tool can be improved to allow analyzing all the PHP files at once.

4. Gathered the information relative to the vulnerabilities, we need to validate them to verify the existence of false positives in order to measure the accuracy of our tool.

   Since there is no data available on all the existing vulnerabilities in the applications, also considering that we are very likely to find new vulnerabilities that have not been exploited yet, it is not possible to build a precise validation tool that contemplates all the vulnerabilities we found.

We had then to manually review the vulnerabilities and check for any inconsistencies or errors, and when possible, correct them (it helped a lot in minimizing the number of false positives). This can be time-consuming, but it is a thorough way to ensure the accuracy of our tool and minimize false positives.

5. Compare our results to other vulnerability detection tools in terms of the number of SQLI and XSS vulnerabilities found, accuracy percentage, false positives and time of analysis.

5.2 Web Applications Data Characterization

In order to conduct an effective evaluation of our model, we selected 12 open-source PHP web applications that met our criteria and had also been previously tested by other tools. Our goal was to compare the results of our model against the most recent and advanced vulnerability detection tool for PHP code (to the best of our knowledge), WAP [1], which was already proven to have a better performance than some other very cited PHP static analysis tools such as Pixy [39] and PhpMinerII [40].

To ensure a fair comparison, we chose to work with applications that had already been tested by WAP. However, because our solution analyzes files individually, while WAP also contemplates vulnerabilities between files, and because we had the limitation of using the phpCallGraph [38] tool, which creates FCG .dot files separately, we chose to select applications with fewer files to minimize any discrepancies in the results. The final list of selected applications is as follows in Table 5.1.

Web Application	Files	Lines of code	Analysis time (s)
Butterfly insecure	16	2364	6
Butterfly secure	15	2678	7
currentcost	3	270	1
Ghost	14	398	1
gilbitron-PIP	14	328	1
Measureit 1.14	2	967	5
Mfm 0.13	7	5859	78
OWASP Vicnum	22	814	2
Peruggia	10	988	2
PHP X Template 0.4	10	3009	8
Webchess 1.0	37	7704	5
ZiPEC 0.32	10	765	2
Total	160	26144	118

Table 5.1: Chosen Web Applications.

In total, the applications that we analyzed consisted of more than 26.000 lines of code distributed in 160 files. This procedure resulted in 12 knowledge graphs (one for each

web application) that represented multiple functions and variables and were composed of numerous sub-graphs, nodes and edges. All these statistics are important for a better understanding of the solution complexity and can be found below in Table 5.2.

Web Application	No. functions	No. variables	No. graphs	No. nodes	No. edges
Butterfly insecure	25	214	279	1264	2905
Butterfly secure	25	226	291	1437	3113
currentcost	5	67	80	163	319
Ghost	14	34	76	339	422
gilbitron-PIP	14	28	70	171	168
Measureit 1.14	51	266	370	736	2249
Mfm 0.13	20	387	434	2020	61585
OWASP Vicnum	22	109	175	310	579
Peruggia	10	87	117	673	967
PHP X Template 0.4	10	124	154	663	5811
Webchess 1.0	49	216	343	1351	1501
ZiPEC 0.32	12	76	108	391	868
Total	257	1834	2497	9518	80487

Table 5.2: Knowledge graph data characterization.

In total, the KGs represent 257 functions with 1834 distinct variables and assemble 2497 graphs (DVGs, CFGs, and FCGs) with 9518 nodes and 80487 edges. From this characterization, we can notice that the number of lines of code does not directly impact the analysis time. Instead, there is a strong correlation between time and the number of nodes and edges, which was expected since the more nodes it has, the more agents are needed for the system to operate, and the more edges it has, the more paths these agents have to navigate. It is also noticeable that by having a higher number of variables, it is more likely to end up with more nodes and edges, but it has no correlation with the number of files, functions or graphs.

5.3 Vulnerability Detection with KAVe

After thoroughly selecting the applications we wanted to analyze, we proceeded to execute our tool over them, following steps 1 to 4 described in Section 5.1 of our evaluation methodology.

As a result of running our tool over these applications, we found a total of 169 vulnerabilities. Out of these vulnerabilities, 30 were identified as SQLi vulnerabilities, and 139 were identified as XSS vulnerabilities. However, it is important to note that out of these, 2 were false positives, one of each kind, respectively.

Having the tool prove to be able to find vulnerabilities in the applications was a key achievement for our project. Furthermore, we were able to answer affirmatively to question number 5, that the tool is able to find vulnerabilities. Additionally, to be able to find

vulnerabilities, the multi-agent system needs a KG to run through that for itself is composed of multiple property graphs so we can deduct the tool is actually able to generate the property graphs and merge them into the KG and so answering both questions number 1 and 2.

The answer to question number 3 is not as linear, as it does not reflect concrete results. However, we were able to visualize that during the analysis, many function nodes that were not necessary for the analysis were not present. The remaining ones were converted into connectors that the agents were able to navigate through. This resulted in an improvement in the analysis time. Based on these observations, we can confidently answer affirmatively that the tool can perform the pruning and so give a positive response to question 3.

After the analysis, we gathered all the collected information and created the Table 5.3 that follows. This table provides a comprehensive summary of the results of our analysis and serves as a useful reference for our further analysis and conclusions.

Web application	**Vulnerable files**	**Vulnerabilities found**	**False Positives**	**Real Vulnerabilities**
Butterfly insecure	4	5	0	5
Butterfly secure	4	5	0	5
currentcost	2	9	0	9
Ghost	5	10	0	10
gilbitron-PIP	0	0	0	0
Measureit 1.14	1	39	0	39
Mfm 0.13	1	6	0	6
OWASP Vicnum	10	15	1	14
Peruggia	6	26	0	26
PHP X Template 0.4	0	0	0	0
Webchess 1.0	9	52	1	51
ZiPEC 0.32	1	2	0	2
Total	43	169	2	167

Table 5.3: Analysis results.

From the results obtained through the execution of our tool, we were able to estimate a precision of 98.81%. These values are auspicious and indicate the potential of our solution. By combining these results with the fact that the tool performed well in terms of time analysis, mostly below 8 seconds (with the exception of Mfm 0.13, which can be explained by the high number of nodes and edges, see Table 5.2), we can confidently conclude that the multi-agent system can effectively navigate the KG both in terms of time and quality. This means that the tool can find vulnerabilities in the applications quickly and accurately. Therefore, we can answer question number 4 affirmatively.

Furthermore, the precision value has significant implications in the evaluation of our solution and indicates that the tool can effectively identify vulnerabilities with a high degree of exactness while also minimizing the number of false positives. This is a crucial aspect of security testing tools and highlights the potential of our solution in providing

valuable results for real-world scenarios.

5.4 Performance Comparison

In this section, we compared KAVe to both WAP and Pixy tools. All the 12 web applications we analyzed are contemplated by WAP but not in Pixy. Nevertheless, we can already guess by the ones in common that WAP performs better than Pixy, so we are more interested in WAP comparison. So we will split our analysis into two phases. First between all three tools and second just between KAVe and WAP. Furthermore, in the first comparison, we have concrete data about the type of vulnerabilities that were discovered, whereas, in the second, we only have access to the total number that was found, so we can not be precise about the types found which may lead to a slightly more biased comparison (more details further ahead).

5.4.1 First Phase Comparison

Starting with comparing KAVe, WAP and Pixy, we tested five web applications in common, obtaining the following results in Table 5.4:

Web application	**KAVe**				**WAP**				**Pixy**			
	SQLI	XSS	FP	RV	SQLI	XSS	FP	RV	SQLI	XSS	FP	RV
currentcost	3	6	0	9	3	4	2	5	3	5	3	5
Measureit 1.14	0	39	0	39	1	7	7	1	1	16	16	1
Mfm 0.13	0	6	0	6	0	8	3	5	0	10	8	2
OWASP Vicnum	1	14	1	14	3	1	3	1	3	1	3	1
ZiPEC 0.32	2	0	0	2	3	0	1	2	3	7	8	2
Total	6	65	1	70	10	20	16	14	10	39	38	11

Table 5.4: Comparison between KAVe, WAP and Pixy.

KAVe was able to detect 70 real vulnerabilities (RV) and had only 1 false positive (FP), estimating a precision of 98.59%; in its turn WAP, could detect 19 real vulnerabilities and had 21 false positives, ending at 47.5% precision; at last, Pixy could detect only 11 real vulnerabilities and had 38 false positives, finishing with 26.8% precision.

These results seem very promising to KAVe, but it is also important to keep in mind that they are related to these five specific applications and could vary depending on which applications we are testing, so it is important to dive deeper into the results by application. We also have to take into consideration that the results for WAP are relative to the WAP-TA (WAP Taint Analyzer), which is the first step of its analysis before applying machine learning to correct the vulnerabilities so it might not be worried so much about the false positives as they will be filtered later on (the reported global precision by WAP authors was 92.5%).

Going throw the applications, in Measureit 1.14 and OWASP vicnum, we can see that KAVe could find many more XSS vulnerabilities without false positives while Pixy found a considerable amount of false positives, and even WAP found some less but still took a big hit on precision. In Mfm 0.13, KAVe could only find one more vulnerability than WAP but four more than Pixy. In currentcost, all of them found the same amount of SQLi, but KAVe was able to find more XSS and avoid false positives. Finally, in ZiPEC 0.32, all of the tools detected two SQLi correctly, although WAP had one false positive, and Pixy found eight of them.

After this deeper analysis, we can conclude that the most significant advantage we can notice in KAVe was in terms of XSS vulnerabilities found and the capability of avoiding false positives (as its evaluation process included correcting errors that lead to them), the number of SQLi was similar across the applications (if we do not consider the false positives).

5.4.2 Second Phase Comparison

For this comparison, we do not have access to the full details of the vulnerabilities found by WAP, so comparing the two tools side by side is more ambiguous. Still, we can draw some conclusions from the data, enriching our evaluation. Pixy did not analyze these specific applications, so we will not include them in the comparison.

We have to take into consideration as well that this comparison is made by making use of results by the complete version of WAP and not just the taint analyzer, so it includes already code correction that our tool does not make, and so we cannot compare them in terms of precision, as WAP complete eliminates the vast majority of false positives found by its taint analyzer. Also, it is important to know that WAP detects more types of vulnerabilities other than SQLi and XSS that KAVe focuses on, so the number of vulnerabilities may only sometimes match the same kinds of vulnerability.

We selected the remaining 7 web applications we analyzed and gathered the comparison results in the following Table 5.5.

Web application	**KAVe**				**WAP**	
	SQLI	XSS	FP	RV	SQLI	XSS
Butterfly insecure	0	5	0	5	0	10
Butterfly secure	0	5	0	5	0	4
Ghost	0	10	0	10	0	3
gilbitron-PIP	0	0	0	0	0	0
Peruggia	18	8	0	26	0	22
PHP X Template 0.4	0	0	0	0	0	0
Webchess 1.0	6	45	1	51	0	13
Total	24	73	1	97	0	52

Table 5.5: Comparison between KAVe and WAP.

In total, KAVe encountered 97 real vulnerabilities (RV) with 1 false positive (FP), while WAP found 52 vulnerabilities with no false positives. Going throw the results, we can notice that KAVe was able to find more vulnerabilities in Ghost (10 vs 3), Peruggia (26 vs 22) and Webchess 1.0 (51 vs 13).

Neither of them could find vulnerabilities in PHP X Template 0.4 or gilbitron application. We tested these specific applications as we already knew WAP did not find any vulnerability to test if our tool could detect any unknown one.

In the application Butterfly, we tested two versions, one insecure and one allegedly secure. Comparing the results, we notice that WAP could find 10 vulnerabilities in the insecure version while KAVe only found 5. This may be due to one of the reasons previously mentioned, such as WAP englobing more kinds of vulnerabilities or being able to find vulnerabilities between files, however, we have no way to be sure.

KAVe still found the same 5 vulnerabilities for the secured version, while WAP only found 4. This led us to speculate that only 4 of the initial ones matched, and the secured version could not correct these particular kinds of vulnerabilities, therefore strengthening our previous motive for the difference in the insecure version. Either way, we can only be sure that KAVe did not match at least 1 of the vulnerabilities found by WAP.

Ultimately, to answer questions 6 and 7, this comparison made us more confident in KAVe's capabilities of finding a significant amount of vulnerabilities while minimizing the false positives, having virtually better results than the top already existing tools for vulnerability detection in PHP web applications.

Chapter 6

Conclusion

This dissertation has aimed to investigate a novel approach for vulnerability detection in web applications written in PHP. Through a thorough analysis of code property graphs and static analysis methodologies, it was found that we are able to assemble multiple property graphs taking each one's advantages into a knowledge graph. This knowledge graph is then enrichened with important information (e.g., entry points and sensitive sinks information) and pruned to remove irrelevant information for the search of vulnerabilities and get an optimized search. Also, this graph is further analyzed by a multi-agent system that can detect vulnerabilities in the source code it accurately represents, employing taint analysis.

These findings have significant implications for establishing a guideline for future works that want to take advantage of knowledge graphs and multi-agent systems in vulnerability detection, as it has never been done to combine these fields of knowledge. They indicate that there are still new techniques that can be leveraged and suggest that there is a need for further research in this area.

The research conducted has added to the existing knowledge in the field by providing the KAVe tool that implements the presented approach and was capable of succeeding in finding vulnerabilities, more importantly, with significant results that are in many situations better or at least comparable with already existing tools with similar purposes. The limitations and strengths of the tool have also been discussed, and suggestions for future research will be made in the following Section 6.1.

In conclusion, this study has contributed to our better understanding of vulnerability detection and has implications for future research to be developed. The findings and recommendations presented in this dissertation provide valuable insights that can inform programmers and entities if they have vulnerabilities and where in their web applications which can help improve code quality and software security.

6.1 Future Work

KAVe has shown promising results in detecting vulnerabilities, but there is always room for improvement. Especially as it is a very complex tool, there are a lot of potential upgrades that were already thought of but were not viable time-wise to implement, including:

- Create our own tool for generating the function call graphs: we are currently using phpCallGraph, which limits us in terms of not being able to generate multiple graphs at once, and also, it is not possible to integrate into our algorithms.
- Develop between file analysis: for now, we can only discover vulnerabilities that are present in the same file, but as it is possible to propagate tainted data between different files and so discover vulnerabilities, we want to integrate also this analysis.
- Optimize the algorithms: the current algorithms used for detecting vulnerabilities have room for improvement in terms of efficiency. One meaningful improvement already thought is to save the state of the analysis of specific functions (e.g., if a function parameter reaches any sensitive sink), so whenever we reach it again, we do not repeat the complete analysis. Further research can be conducted to develop new algorithms that are better suited for detecting different types of vulnerabilities.
- Analysis expansion: taint analysis covers most of the scenarios needed for vulnerability detection, although a current challenge is that sanitization does not cover input alteration with, for example, string manipulation functions that can lead to a ”virtual” sanitization that should be contemplated.
- Detect different types of vulnerabilities: The tool will be improved to detect additional vulnerabilities other than SQLi and XSS.

Bibliography

[1] I. Medeiros, N. Neves, and M. Correia, "Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining," *In Proceedings of IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 54–69, 2016.

[2] O. Foundation, "Open Web Application Security Project." [Online]. Available: https://www.owasp.org/

[3] A. C. for Schools, "Western Association of Schools and Colleges." [Online]. Available: https://www.acswasc.org/

[4] T. M. Corporation, "Common Weakness Enumeration." [Online]. Available: https://cwe.mitre.org/

[5] Kingthorin, "Sql injection." [Online]. Available: https://owasp.org/www-community/attacks/SQL_Injection

[6] S. Kost, "An Introduction to SQL Injection Attacks for Oracle Developers," 2004.

[7] A. Klein, "Dom based cross site scripting or xss of the third kind," 2005. [Online]. Available: http://www.webappsec.org/projects/articles/071105.shtml

[8] S. D. P. M. H. E. A. V. N. J. W. Dave Wichers, Arshan Dabirsiaghi, "Types of xss." [Online]. Available: https://owasp.org/www-community/Types_of_Cross-Site_Scripting

[9] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *Proceedings of the 13th International Conference on World Wide Web*, ser. WWW '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 40–52. [Online]. Available: https://doi.org/10.1145/988672.988679

[10] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Kuo, "Verifying web applications using bounded model checking," in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, ser. DSN '04. USA: IEEE Computer Society, 2004, p. 199.

[11] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Software*, vol. 19, no. 1, pp. 42–51, 2002.

[12] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: effective taint analysis of web applications," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 87–97, 2009.

[13] A. Figueiredo, T. Lide, D. Matos, and M. Correia, "Merlin: Multi-language web vulnerability detection," in *2020 IEEE 19th International Symposium on Network Computing and Applications (NCA)*, 2020, pp. 1–9.

[14] T. Ishio, "Dependency-based extraction of conditional statements for understanding business rules," *IEICE Transactions on Information and Systems*, vol. E99-D, pp. 1117–1126, 04 2016.

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

[16] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and flexible discovery of php application vulnerabilities," in *Proceedings of the IEEE European Symposium on Security and Privacy*, Paris, France, 2017, pp. 334–349.

[17] M. Wijngaard, "Dependence Analysis in PHP," Amsterdam, The Netherlands, Aug. 2016. [Online]. Available: http://www.scriptiesonline.uba.uva.nl/618176

[18] R. Johnson, D. Pearson, and K. Pingali, "The program structure tree: Computing control regions in linear time," ser. PLDI '94. New York, NY, USA: Association for Computing Machinery, 1994, p. 171–185. [Online]. Available: https://doi.org/10.1145/178243.178258

[19] F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, p. 1–19, jul 1970. [Online]. Available: https://doi.org/10.1145/390013.808479

[20] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, p. 319–349, jul 1987. [Online]. Available: https://doi.org/10.1145/24039.24041

[21] A. Johnson, L. Waye, S. Moore, and S. Chong, "Exploring and enforcing security guarantees via program dependence graphs," *SIGPLAN Not.*, vol. 50, no. 6, p. 291–302, jun 2015. [Online]. Available: https://doi.org/10.1145/2813885.2737957

[22] L. Ehrlinger and W. Wöß, "Towards a definition of knowledge graphs." *SEMANTiCS (Posters, Demos, SuCCESS)*, vol. 48, no. 1-4, p. 2, 2016.

[23] D. Fensel, U. Şimşek, K. Angele, E. Huaman, E. Kärle, O. Panasiuk, I. Toma, J. Umbrich, and A. Wahler, *Introduction: What Is a Knowledge Graph?* Cham: Springer International Publishing, 2020, pp. 1–10. [Online]. Available: https://doi.org/10.1007/978-3-030-37439-6_1

[24] G. Weiss, *Multiagent Systems, A Modern Approach to Distributed Artificial Intelligence*. Cambridge, Massachussetts, London, England: The MIT Press, 1999.

[25] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, 2010.

[26] P. Maes, "Pattie maes on sofware agents: Humanizing the global computer," *IEEE Internet Computing*, vol. 1, no. 4, pp. 10–19, 1997.

[27] D. C. Smith, A. Cypher, and J. Spohrer, "Kidsim: Programming agents without a programming language," *Commun. ACM*, vol. 37, no. 7, p. 54–67, jul 1994. [Online]. Available: https://doi.org/10.1145/176789.176795

[28] N. Gilbert and R. Conte, *Artificial societies*. Taylor & Francis, 1995.

[29] G. Weiss, "A modern approach to distributed artificial intelligence," *IEEE Trans. Syst. Man Cybern. Part C Appl. Rev*, vol. 22, no. 2, 1999.

[30] V. R. Lesser, "Cooperative multiagent systems: A personal view of the state of the art," *IEEE Transactions on knowledge and data engineering*, vol. 11, no. 1, pp. 133–142, 1999.

[31] N. R. Jennings, "On agent-based software engineering," *Artificial intelligence*, vol. 117, no. 2, pp. 277–296, 2000.

[32] S. Kim, Y. M. Kim, J. Hur, S. Song, G. Lee, and B. Lee, "FuzzOrigin: Detecting UXSS vulnerabilities in browsers through origin fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1008–1023. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/kim

[33] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "Beacon: Directed grey-box fuzzing with provable path pruning," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 36–50.

[34] S. Park, D. Kim, S. Jana, and S. Son, "FUGIO: Automatic exploit generation for PHP object injection vulnerabilities," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 197–214. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/park-sunnyeo

[35] C. Lin, Y. Xu, Y. Fang, and Z. Liu, "Vuleye: A novel graph neural network vulnerability detection approach for php application," *Applied Sciences*, vol. 13, no. 2, 2023. [Online]. Available: https://www.mdpi.com/2076-3417/13/2/825

[36] P. J. S. Aric A. Hagberg, Daniel A. Schult, "Exploring network structure, dynamics, and function using networkx," 2008. [Online]. Available: https://networkx.org/

[37] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, "Graphviz—open source graph drawing tools," in *Graph Drawing*, P. Mutzel, M. Jünger, and S. Leipert, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 483–484.

[38] M. R. C. R. N. S. Y. S. Falko, Johannes Nicolai, "Phpcallgraph," 2015. [Online]. Available: http://phpcallgraph.sourceforge.net

[39] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (SP'06)*, 2006, pp. 6 pp.–263.

[40] L. K. Shar and H. B. K. Tan, "Mining input sanitization patterns for predicting sql injection and cross site scripting vulnerabilities," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 1293–1296.

[41] L. Lamport, *LaTeX - A Document Preparation System: User's Guide and Reference Manual, Second Edition*. New York: Pearson / Prentice Hall, 1994.

[42] A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 01 1937. [Online]. Available: https://doi.org/10.1112/plms/s2-42.1.230

[43] K. J. Q. Y. A. Dave Benjamin, Dan Colish, "Phply," 2018. [Online]. Available: https://github.com/viraptor/phply

[44] M. W. Anthony Ferrara, Nikita Popov, "Php-cfg," 2016. [Online]. Available: https://github.com/ircmaxell/php-cfg

[45] M. Muths, "Phpdependencyanalysistool," 2015. [Online]. Available: https://github.com/mamuz/PhpDependencyAnalysis

[46] M. Wijngaard, "Php-pdg," 2016. [Online]. Available: https://github.com/mwijngaard/php-pdg

[47] F. Y., "The programming legend single-entry single-exit (sese) – a test perspective," 2017. [Online]. Available: https://sysadminpp.wordpress.com/2017/08/18/the-programming-legend-single-entry-single-exit-sese-a-test-perspective/

[48] KirstenS, "Cross site scripting (xss)." [Online]. Available: https://owasp.org/www-community/attacks/xss/