KAVE: A Knowledge-Based Multi-Agent System for Web Vulnerability Detection

Rafael Ramires
LASIGE, DI, Faculdade de Ciências
Universidade de Lisboa, Portugal
rframires@fc.ul.pt

Ana Respício
LASIGE, DI, Faculdade de Ciências
Universidade de Lisboa, Portugal
alrespicio@fc.ul.pt

Ibéria Medeiros
LASIGE, DI, Faculdade de Ciências
Universidade de Lisboa, Portugal
imedeiros@di.fc.ul.pt

***Abstract*—The growing use of the web has led to a rise in cyber attacks exploiting software vulnerabilities, thereby causing significant damage to companies and individuals. Static analysis tools can assist programmers in identifying vulnerabilities within their code. However, these tools are prone to producing false positives and lack precision, which relegates them to a somewhat marginalised role in software development. This paper proposes a new and more effective static analysis approach for assessing and evaluating web applications against vulnerabilities by using a knowledge-based multi-agent system web vulnerability detector called KAVE. The multi-agent system performs static taint analysis over a specially designed multi-layer knowledge graph, whereas this graph aggregates diverse interconnected representations of the lexical and semantic features of the application's source code, their data and control flows, and function calls. Additionally, this graph integrates security properties associated with vulnerabilities. The evaluation results of KAVE and comparison with existing tools showed that KAVE employs an effective and efficient method to detect vulnerabilities in web applications, finding 235 vulnerabilities with a precision of 95.9% over 12 open-source PHP web applications.**

***Index Terms*—Web Application Vulnerabilities, Static Analysis, Multi-Layer Knowledge Graph, Multi-Agent System, Software Security**

I. INTRODUCTION

The digital transformation and increased accessibility to devices (e.g., computers, smartphones, and tablets) have leveraged the World Wide Web's endless growth. However, the urge for innovation has led developers to adopt automatic code-generation platforms to create program functionalities, resulting in a shortfall in pre-release software testing, thus causing a decline in overall quality [1]. These factors compromise the integrity of the released software and amplify the risk of potential attacks [2]. A higher number of insecure web applications represents a greater risk of vulnerability exploitation, and inadequate testing makes them an easy target [3]. Despite the prevalence of insecure software, the lack of knowledge on detecting and effectively solving its vulnerabilities continues to be a topic of much discussion [4]–[7]. In web applications, the two most common and exploited vulnerabilities are SQL injection (SQLi) and Cross-site scripting (XSS) [3], [8]. Their exploitation can cause huge harm to users and organisations, depending on the attacker's experience, the vulnerability's severity [9], and the application's business value. PHP is the most widely used language in building such applications [10]. Moreover, it is a loosely-typed language, meaning variables are not explicitly declared with a specific data type and assume the data type of their value. All these factors make PHP web applications particularly attractive to attacks, thereby needing rigorous validation before market release.

Static analysis systematically scans for vulnerabilities in applications, offering developers the advantage of identifying code vulnerabilities without executing the code or requiring a complete application version. This allows the integration of static analysis tools (SASTs) into the development lifecycle, aiding in creating secure software. By analysing source code representations, SASTs facilitate the detection of potential security issues. However, the challenge of false positives, where non-issues are mistakenly flagged as vulnerabilities, can hinder their effective use and adoption by developers despite their potential benefits [11]. Most SASTs perform taint analysis relying on the Abstract Syntax Tree (AST) [6], [12]–[15], while others rely on the Control Flow Graph (CFG) [16], [17]. These code representation structures have limitations for program analysis. While ASTs, though complete, result in substantial complexity and size, CFGs do not reveal data dependencies, which is fundamental for vulnerability detection. To overcome these issues, Yamaguchi et al. [7] introduced the Code Property Graph (CPG), a combination of AST, CFG, and PDG (Program Dependence Graph [18]–[21], which by itself combines CFG and Dependence Variable Graph (DVG) [18]) to enhance the analysis's precision of C programs. This is achieved by making graph traversals considering the underlying structures' code properties. The CPG relies on a PDG with nodes representing statements and predicates, and connected by the data and control flow dependencies between them. Each PDG's node is defined by its AST, which allows the breakdown of statements and predicates in code elements.

Based on the CPG concept, SAST approaches for web applications have emerged [21]–[24], but most of them do not indicate whether vulnerabilities have been found; instead, they present the resulting traversals found (i.e. the data flow paths), and the user must analyse them, looking for vulnerabilities. Furthermore, they do not consider the Function Call Graph (FCG) [18], [21], determinant for the inter-procedure analysis and propagation of taint analysis in vulnerability discovery.

This paper presents a new, more effective static analysis approach for assessing web applications against vulnerabili-

ties using a *Knowledge-based multi-Agent system web Vulnerability detector* (KAVE). Inspired by the *Code Property Graph* concept and the *Multi-Agent System* (MAS) domain, we propose, respectively, (i) a *Multi-Layer Knowledge Graph* (MLKG) to represent the program's code properties, with different code structure representations (FCG, CFG, DVG and PDG), and enriched with security properties of vulnerabilities, thus enclosing knowledge about all the code and vulnerabilities, and (ii) a MAS to perform static taint analysis over the MLKG to detect potential vulnerabilities more effectively. The approach also integrates a *pruning* method to strategically discard irrelevant MLKG's nodes for vulnerability analysis purposes effectively and efficiently, and minimising the false positive occurrence. A MAS [25], [26] consists of multiple autonomous agents interacting or working with their fellow agents and the surrounding environment. Each agent is capable of autonomous problem-solving, which operates asynchronously concerning other agents and influences different parts of the environment, depending on the relationships between agents [27]. In turn, static taint analysis [6], [16], [28] is a technique used to track input data (i.e., entry points like `$_POST` in PHP) in a program and so identify potentially dangerous code places (i.e., sensitive sinks like `echo` in PHP) where this data is used. In our context, our MAS comprises different agents that operate in the MLKG's nodes autonomously but depend on the other agents to make decisions about taint propagation and discover vulnerabilities.

We implemented the approach in the KAVE tool. KAVE creates the MLKG, connecting the FCG, CFG, DVG, and PDG graphs to each other, representing the lexical and semantic features, data and control flows, and function calls in the application's source code. Then, it enriches the graph with security properties associated with vulnerabilities (e.g., entry points and sensitive sinks) not present in these structures as such, and applies the pruning method. Afterwards, the MAS traverses the MLKG to look for potential vulnerabilities in a computational effort-saving way. We evaluated KAVE's effectiveness for SQLi and XSS vulnerabilities with synthetic and real applications and compared its results with standard SAST and CPG tools. The results showed that KAVE was able to detect vulnerabilities in web applications efficiently, finding 235 vulnerabilities in 12 open-source PHP web applications, with a precision of 95.9%.

The main contributions of the paper are: (1) a new static analysis approach for improving web application security. The approach relies on $i)$ an MLKG graph combining the strengths of different code representations for vulnerability detection, $ii)$ a pruning method eliminating from the MLKG irrelevant information for vulnerability detection, resulting in a more manageable graph enhancing the analysis, minimising false positives, and $iii)$ a MAS that identifies potential vulnerabilities by coordinating MLKG traversals and validations; (2) the KAVE tool that implements the approach and an experimental evaluation for XSS and SQLi detection in PHP applications. The tool is open source and available at [29].

II. Background

This section presents the background on web vulnerabilities in this work and provides an overview of code representation graphs, multi-agent systems, and multi-layer graphs.

A. Web Vulnerabilities

Vulnerabilities primarily arise from errors in software design, implementation, or configuration. When attackers find them, they can exploit them to breach the application's security and cause harm. Organisations like OWASP [8], WASC [30], and CWE [9] have been established to help identify and mitigate vulnerabilities, being OWASP the most well-known for web applications.

```
1  function gymEntrance(int $age, bool $ismember){
2     if (hasError()){ //function error
3        return;
4     }
5     $fee = $_GET['fee'];
6     if(htmlentities($age) <= 18){ //is junior
7        if($ismember){
8           return;
9        }
10       $fee = $fee - 10; //discount 10
11    }
12    else if($age >= 60){ //is senior
13       $fee = $fee - 5; //discount 5
14    }
15    echo $fee;
16    $fee = 2;
17    setFee(htmlentities($fee)); //set entrance fee
18    setTimer(1); //set time inside
19    if($ismember){
20       setTimer(2);
21       login();
22    }
23    return;
24 }
```

Listing 1. A user function definition in PHP for the graph generation and that contains an XSS vulnerability.

The two most prevalent vulnerabilities present in web applications are SQLi and XSS. SQLi [31], [32] is a vulnerability where an attacker can manipulate the data input sent to a web application to execute unauthorised SQL commands in the database. This can allow the attacker to access or modify sensitive and private data stored in the database, such as user passwords or financial data. XSS [33], [34] allows an attacker to inject malicious scripts (e.g., JavaScript) into a web page viewed by other users. These scripts can be used to steal sensitive data, such as user cookies or session tokens, or to perform actions on behalf of the user without their knowledge or consent. The PHP code in Listing 1 contains an XSS vulnerability in line 15. If the entry point `$_GET['fee']` (in line 5) receives a malicious script (e.g., `<script>alert("XSS")</script>`) when the `$fee` variable is used in the `echo` sensitive sink (in line 15), the script will be executed in the victim's browser.

B. Code Representation Graphs

Function Call Graph. A FCG [18], [21] represents the relationships between the functions in a program, showing how functions are called and how they call other functions, creating

a hierarchy of function calls. Each node in the graph represents a function, and the edges represent the data flow between the functions. An FCG can be used to understand the program's structure, identify areas prone to errors or performance issues, and enhance code efficiency by recognising frequently invoked or resource-intensive functions.

Control Flow Graph. A CFG [21], [35], [36] represents a program's execution sequence or a code snippet (e.g., a function). It depicts the order in which the instructions of a program are executed and how they are related to each other, considering the predicates existing in the program. The CFG's nodes represent individual or blocks of (sequential) instructions, and the edges represent the possible execution paths between them. The edges are usually labelled with the predicate or statement determining the flow direction.

Dependence Variable Graph. A DVG [18] is a directed graph representing the causal relationships (edges depict dependencies) between a program's variables (represented by nodes).

Program Dependence Graph. A PDG [18]–[21] is a directed graph representing the dependencies between statements and predicates in a program (or function), as well as data and control dependencies. Nodes depict statements, variables, and predicates, while arcs represent the control and data flow between the elements in the nodes. Generally, one can see a PDG as a merging of the CFG and DVG into a single graph. The PDG is very useful in program analysis and optimisation, as it provides a structured way to understand the interactions between different program parts.

C. Multi-Layer Graphs

Multi-layer graphs [37] represent relationships of heterogeneous objects, going beyond traditional graphs into a richer framework capable of hosting objects and relations of different types. Thus, a multi-layer graph can be seen as a multi-layer object in which each layer encloses a uni-dimensional graph. Each node in a given layer can connect to any other node or sub-graph of a graph in a different layer.

D. Multi-Agent Systems

A Multi-Agent System (MAS) [25] consists of several agents, each capable of perceiving and interacting dynamically with their surroundings to make decisions and execute actions towards fulfilling their individual objectives. Within this framework, agents typically operate with a degree of independence, driven by their interests, which might diverge or converge with those of their peers. The definition of an agent is contingent on its operational context and application domain [38]–[41], embodying a computational unit that autonomously functions within a bounded setting, potentially alongside other agents possessing similar or distinct attributes. The intelligence embedded in each agent within a MAS is tailored to meet the system's needs and their specific roles, ranging from straightforward, rule-based algorithms to complex adaptive systems. This diversity is crucial for the MAS's ability to efficiently meet its overarching objectives while maintaining operational effectiveness in its designated environment.

III. KAVE System Approach

Our approach relies on a MAS system that, in an intelligent manner, navigates an MLKG graph to effectively and efficiently discover vulnerabilities in web applications. To obtain such effectiveness, the approach extracts diverse essential code representation graphs (FCG, CFG, DVG, and PDG) to depict code, control and flows in interconnected ways, which are enriched with properties associated with vulnerabilities (e.g., entry points, sanitisation and sink information). These enriched graphs are merged into an MLKG. The MLKG settles in an FCG (the first MLKG layer), where the nodes represent user functions and the edges represent calls between these functions. As a second layer, the FCG's nodes link to the PDG of the function they represent. In essence, each PDG encapsulates and connects both the CFG and DVG for its corresponding function. The security properties are reflected in the third layer of the MLKG, which segregates the properties existing in the PDGs and are transmitted and aggregated to the nodes of the FCG.

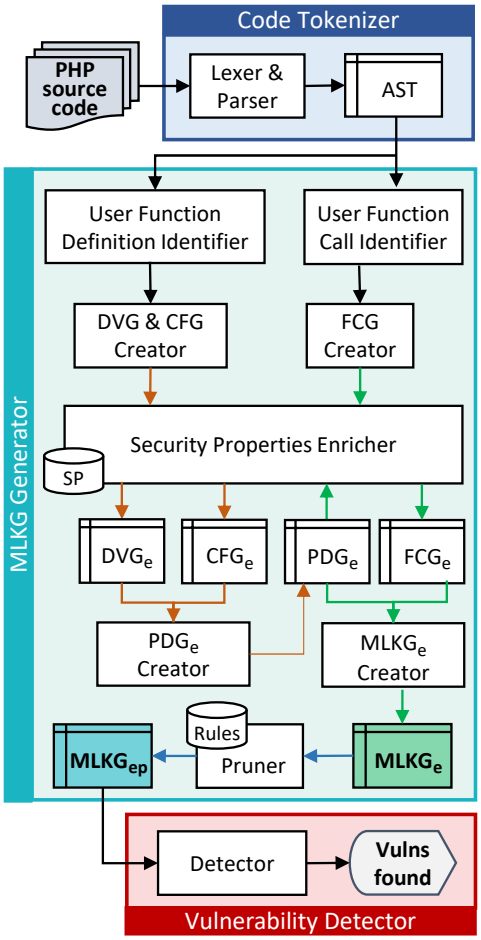

Fig. 1. KAVE system architecture.

We propose a MAS to navigate the MLKG to detect vulnerabilities, thus creating an intelligent and guided system that performs static taint analysis to locate vulnerabilities dynamically and autonomously. Unlike traditional SASTs that inspect the entire application code, following all application

entry points, regardless of where in the application point they end (i.e. a sensitive sink or any other unrelated vulnerability point), our MAS analyses the first layer of the MLKG and, together with information from the third layer, decides whether an in-depth inspection of the second layer is necessary. This approach saves computational effort by streamlining the analysis process and focusing only on relevant parts of the code. Moreover, this hierarchical layer organisation allows us to subsequently prune the MLKG, reducing the traversal effort and enabling the exclusion of MLKG sub-graphs that do not require analysis. This strategy reduces the overall search space, hence improving efficiency.

Figure 1 depicts the approach architecture, which comprises three modules: *Code Tokenizer*, *MLKG Generator*, and *Vulnerability Detector*. The following sections describe these modules: Section IV, the first two, and Section V, the last.

IV. Multi-Layer Knowledge Graph Generator

To create the MLKG, the code must first be split into tokens, and the *Code Tokenizer* module is responsible for this action. Using a *Lexer & Parser*, it breaks the code into tokens based on language syntax (e.g., PHP syntax), cleans irrelevant data (e.g., comments, semi-colons), and then creates an abstract syntax tree (AST). In the AST, each code instruction is an AST's branch, which is expressed as a list containing the relevant code needed to construct the graphs. For example, the instruction `$fee = $_GET['fee'];` (line 5 of Listing 1) is broken into three tokens and results in the following list in the AST: `[$fee, =, $_GET['fee']]`.

Contrary to SASTs [6], [12], [14] that resort to an AST to search for bugs, the purpose of AST in our approach is to extract the essential graphs needed to build the MLKG. Hence, once the AST has been obtained, the *MLKG Generator* module processes it to obtain these graphs. The module employs a five-step pipeline that we describe in the next sections.

A. User Function Identification

As the MLKG's first layer relies on an FCG, one uses the AST to identify the user functions' definitions and where they are called. While the latter allows for the construction of the FCG, the former gives place to the building of the PDG of each user function identified and called. To identify user function definitions, this step searches for `function` as the initial token in the AST's branches (i.e., lists), retrieves the sub-AST associated with each function found (i.e., the lists that represent the instructions of the function), for then to be delivered to the *Enriched PDG Creation* step (see Section IV-B). On the other hand, to identify user function calls, for each call function found in the AST, this step checks whether it is a user function whose definition has been previously identified. Later, these function calls are delivered to the *Enriched FCG Construction* step (see Section IV-C).

B. Enriched PDG Creation

We inspect each function's sub-AST to extract its Dependency Variable Graph (DVG) and Control Flow Graph (CFG), and then the Program Dependence Graph (PDG).

DVG construction. The DVG is created by tracking all the variable dependencies in a function and defining which variables depend on others. A DVG is created for each variable. For example, if a function has three variables, three DVGs are created. Each DVG is a digraph, whose nodes represent lines of code and edges represent the variable dependency between those lines. Afterwards, the DVG's nodes are enriched with security properties associated with vulnerability information and useful for its discovery, such as whether the node (statement) contains an entry point (e.g., `$_GET`), sensitive sink (e.g., `echo`), and a sanitisation function (e.g., `htmlentities`). Each node of the resulting *enriched-DVG* (DVG_e) is defined by a tuple *<i, d, v, b, l>*, where:

- *i* is the code line corresponding to that node;
- *d* is the relative depth of the statement within this function, indicating its nesting level relative to the function's top-level scope. It increases as the control flow enters nested structures (conditional clauses or loops), and decreases upon exiting these structures;
- *v* is an abstract version of the represented variable; it 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* determines the branch of a sequential conditional clause in which the code line is;
- *l* is a label resulting from the statement evaluation concerning the node's security properties (e.g., entry point, sensitive sink, and sanitisation function).

CFG construction. The CFG contains the control flow information of a function, describing all the possible paths that might be traversed via the program's execution, considering its conditional instructions and branches. The CFG is a digraph whose nodes and edges represent lines of code and control flow between them. We also enrich the CFG's nodes with security properties, thus resulting in an *enriched-CFG* (CFG_e), in which each node is a tuple *<i, d, r, c, l>*, where:

- *i* is the code line corresponding to that node;
- *d* is the relative depth of the corresponding statement in the function (as parameter *d* in the DVG_e);
- *r* is *True* if the node contains a return statement, and *False*, otherwise;
- *c* is *True* if the node contains a conditional statement, and *False*, otherwise;
- *l* is a label enclosing the node security properties, similarly to *l* in the DVG_e.

PDG_e construction. We merge the DVG and CFG graphs generated for each function into a PDG. The PDG is a comprehensive graph combining control and data dependencies between statements and predicates. The PDG_e generator combines the DVG_e of each variable in a function with the single corresponding CFG_e. To do so, it uses the CFG_e as a basis. It adds the edges of each DVG_e to the CFG_e, differentiating them as either data edges or control edges and labelling the data edges with the variables they represent. As illustrated in the top-right of Figure 2, the PDG shows

only one DVG for the variable `$fee`. If more variables existed, additional DVGs would be attached to the CFG, each with its corresponding label. As both types of edges contain information about vulnerabilities, the final graph is also enriched, called *enriched-PDG* (PDG_e), in which a node corresponds to merging the nodes in the CFG_es and DVG_es so that *<(i, d, r, c, v, b, l)>= <(i, d, r, c, l)> ∪ <(i, d, 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_e and DVG_e, will be the glue points between the nodes to assemble the new one.

Differently from the DVG_e and CFG_e, the PDG_e has two distinct types of edges that are represented as tuples:

- *Control flow dependency* defined as $Ef = (N_i, N_k)$, where N_k represents a node reachable from the N_i node;
- *Data dependency* defined as $Ed = (N_i, N_k, V)$, where N_k is a node having at least one variable instantiated for the last time on N_i node, and V is the set of variables associated with this dependency.

Figure 2 shows the PDG_e of the `gymEntrance` function of Listing 1. There, one can see the edges for variable `$fee` throughout the control flow. Also, nodes 5 and 15 are tagged with security properties that represent, respectively, variable `$fee` receiving the entry point `$_GET` and reaching the sensitive sink `echo`.

C. Enriched MLKG Creation

FCG and FCG$_e$ construction. Based on the information provided by the *User Function Identification* step, namely the user functions calls, this step generates the *Enriched-FCG* (FCG_e). First, it creates the FCG of each file the web application contains, comprising thus the user functions called in each file and provided by the AST. The FCG nodes represent functions and its edges the function calls, of the form (F_i, F_k), being F_k a function called at least once inside function F_i (the callee). Next, its nodes are enriched with the security properties found in the PDG_e they represent, thus resulting in an FCG_e whose nodes are tuples *<i, f, ll >*, where *i* is the code line corresponding to that node; *f* is the function it represents; and *ll* is the set of security properties presented in its corresponding PDG_e. To obtain this set, the PDG_e is analysed to collect its security properties and aggregate them into *ll*.

Note that the *Security Properties Enricher* is used in two distinct moments: i) when the DVG_e and CGF_e are generated, their nodes are compared against an established database of security properties to know if they are associated with data about vulnerabilities; ii) to aggregate into FCG's nodes the security properties already determined in PDG_e. The upper half of Figure 2 shows the FCG_e containing ii), and the diverse PDG_e reflecting i).

MLKG$_e$ construction. While the FCG alone may not contain enough information to support any analysis, it can be combined with other data structures, such as PDGs, enabling inter-procedural analysis and, therefore, identifying potential vulnerabilities in the code. To accomplish this, our approach builds an $MLKG_e$ which integrates (1) the FCG_e as its foundational structure with (2) the previously generated PDG_es, which constitute its second layer, and (3) the security properties they contain, constituting its third layer. Therefore, after creating the FCG_es and PDG_es, the next step is to merge them to generate the $MLKG_e$. This involves extending the nodes of the FCG_e with pointers (*p*) that link to the corresponding functions' PDG_es, resulting in tuples *<i, f, p, ll >*. The resulting $MLKG_e$ is a more suitable structure for vulnerability detection, as it gives us clues as to where the vulnerabilities might or might not exist and, thus, allows for a more oriented analysis and pruning of the graph at a superficial layer (explained next).

The upper half of Figure 2 shows the $MLKG_e$ of the defined `gymEntrance` user function in Listing 1. The FCG_e begins with the `main` node that denotes the beginning of the program, and then it calls the function `gymEntrance`. In turn, that function calls other user functions that will be connected to it, meaning that they are called by it. Each function links to its PDG_e, and close to each of them, we can observe the security properties collected from PDG_es. For instance, we can see that the FCG_e's node of `gymEntrance` inherits the security property from its PDG_e. Also, assuming that the code for the `login` function is that of Listing 2, where its PDG_e's nodes corresponding to lines 2 and 3 are tagged as `entry`, and line 5 as `sink`, the `login` node in FCG_e aggregates these security properties.

```
1  function login(){
2      $u = $_POST['username'];
3      $p = $_POST['password'];
4      $q = "SELECT * FROM users WHERE user='$u' AND
           pass='$p'";
5      $r = mysqli_query($con, $q); // sink of SQLi
6  }
```

Listing 2. The login user function definition with a SQLi vulnerability.

D. Pruning

The *Pruning* step evaluates each node in the $MLKG_e$ and determines whether it can be trimmed, turned into a connector, or kept. To do this evaluation, we take into consideration several factors, such as whether the node is a `Void` function (that does not call any other functions and does not return any value) and whether the node contains any security properties or function entry parameters that would intuitively come from another region of the program. Table I resumes the combination of these factors and their resulting pruning outcomes.

If a node meets the criteria for being trimmed, it is entirely removed from the graph, i.e., its FCG_e node and PDG_e. On the other hand, if it meets the criteria for being turned into a connector, its PDG_e is trimmed; however, its FCG_e node remains in the graph but as a passage node, where only the entry and returned parameters are considered. Lastly, if a node does not meet either of these criteria, it is deemed necessary for the vulnerability analysis and is kept intact. The bottom half of Figure 2 depicts the $MLKG_e$ of the

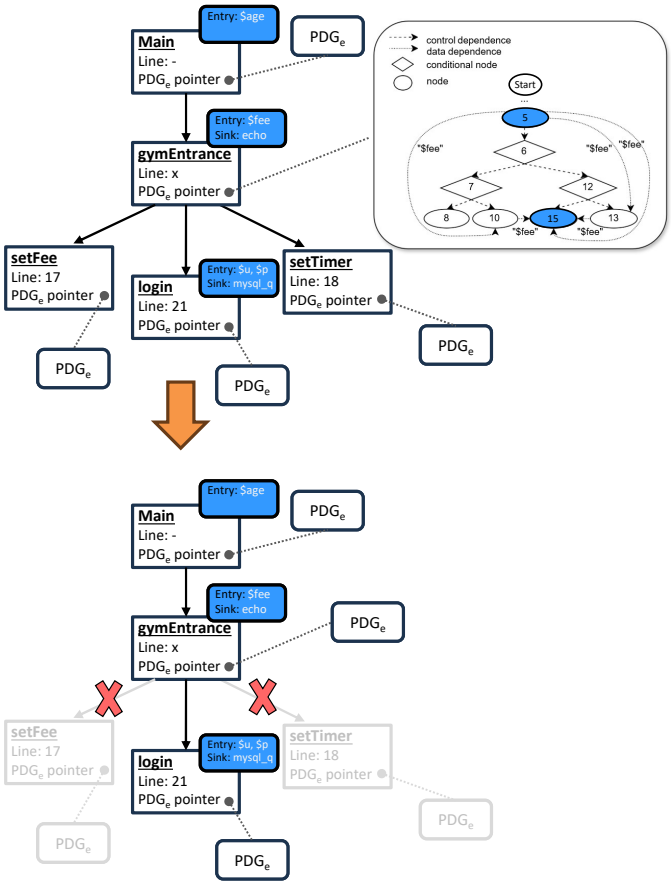

Fig. 2. Enriched Multi-Layer Knowledge Graph ($MLKG_e$) representation (top) and $MLKG_{ep}$ after pruning (bottom).

`gymEntrance` function after applying the pruning step. For example, as the `setFee` and the `setTimer` functions do not have any of those factors associated, they are entirely trimmed. In contrast, for instance, supposing the `gymEntrance` only contained an *Entry*, calls other functions, and was not the void type, it would be turned into a *Connector*. By evaluating all nodes in the $MLKG_e$ and determining which ones can be trimmed or turned into connectors, our pruning method produces a much lighter version of the graph, which will allow for saving running time and minimising false positive and negative production. This is especially important in scenarios where multiple passages through the graph are required for a complete analysis, as the efficiency gains from pruning can add up over time. As a result of the pruning method, we have $MLKG_{ep}$, an enriched and pruned graph. However, for simplicity, we call it MLKG from now on.

TABLE I
TABLE OF DECISIONS FOR PRUNING.

Type of Function	Function Parameters	Security Properties			Pruning Outcome		
		Entry	Sanitisation	Sink	Trim	Connector	Keep
Not Void	✓					✓	
		✓				✓	
	✓	✓				✓	
			✓				✓
				✓			✓
	Other Combinations						✓
Void				✓	✓		
	✓				✓		
		✓			✓		
			✓		✓		
	✓	✓			✓		
			✓		✓		
		✓	✓	✓	✓		
			✓		✓		
				✓	✓		
	Other Combinations						✓

V. VULNERABILITY DETECTOR

The vulnerability detection approach we propose is a multi-agent system (MAS) designed to enhance navigation over the MLKG, emphasising computational efficiency and vulnerability detection robustness. Our MAS introduces a novel perspective in web application security, drawing from characteristics such as partial independence, self-awareness, and autonomy, which enable the effective utilisation of the MLKG.

Each agent operates with a localised view and does not possess a global perspective of the entire system, but all together have it. Consequently, agent communication becomes vital for cooperation, ultimately contributing to the system's overall effectiveness. The MAS empowers the vulnerability assessment with a layered approach that excels at handling web application's intricate, multi-layered structures. With the capacity to allocate distinct agents to specific layers, the MAS fosters a level of granularity and context-aware analysis that is invaluable in uncovering vulnerabilities. In contrast to traditional SASTs that analyse the entire application code, which may be unnecessary and may lead to an ineffective detection, the MAS, together with the MLKG, enables the analysis in PDG_es and communication between layers when such is required. Additionally, MAS aims to reduce the incidence of false positives and negatives through cross-verification and consensus mechanisms among agents. This will bolster the reliability of vulnerability assessments, providing actionable results that security teams can confidently address. Moreover, the customisation capabilities inherent in MAS bring a tailored and precise dimension to vulnerability assessment. Each agent can be fine-tuned for specific testing scenarios or requirements, ensuring the assessment aligns closely with the application's unique architecture and potential vulnerabilities.

Figure 3 overviews our MAS for vulnerability detection. In MAS, different types of agents are carefully designed, each assigned a distinct task and specialised focus. These categories include *Travel Agents* (TA) responsible for discovering paths comprising security properties throughout the FCG_e nodes, *Verification Agents* (VA) focused on scrutinising code for vulnerabilities from the paths provided by the former agents, single *Translation Agents* (TrA) adept at translating data and information across the MLKG's layers, *Control Flow Agents* (FA) and *Data Agents* (DA) responsible for into PDG_e, respectively, handle and inspect control and data flows. Next, we detail each agent.

Travel Agent. The *Travel Agent* (TA) is responsible for discovering possible vulnerability paths over the first layer of MLKG. Each FCG_e's node has its TA, and it operates by traversing the FCG_e starting from entry points existing in its node and looking for paths that end in a sensitive sink, either

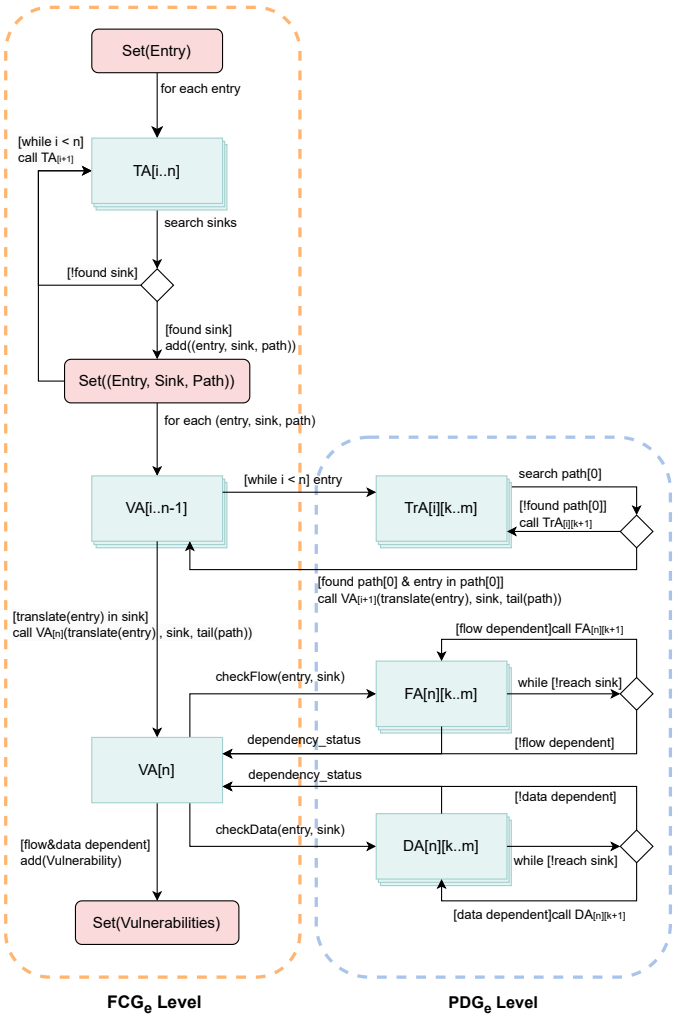

Fig. 3. Multi-Agent System (MAS) for vulnerability detection.

existing in its node or another. During its search, an agent can communicate with other TAs below it to check if any FCG_e's nodes have sensitive sinks in their labels. If a sensitive sink is found, the agent saves the path it took to get there, signalising that a possible vulnerability may be in that path. For instance, considering the MLKG of Figure 2 and the TA of `Main` node, it will find two paths `P1:[<$age, echo>, <Main, gymEntrance>]`, `P2:[<$age, mysql_query>, <Main, gymEntrance, login>]`. In contrast, the TA of `gymEntrance` will also find two paths `P3:[<$fee, echo>, <gymEntrance>]`, `P4:[<$fee, mysql_query>, <gymEntrance, login>]`. Later, all other types of agents use this information to conduct a directed search, avoiding unnecessary paths and making the analysis more efficient, since it identifies all the possible vulnerabilities and the function paths we should check to verify whether it represents a real vulnerability or not.

Verification Agent. The *Verification Agent* (VA) is responsible for inferring whether a data and control flow pathway exists between identified pairs of entry points and sinks, which the TA has flagged as potentially vulnerable. Analogous to the functionality of TAs, the VA operates exclusively at the FCG_e level, with each node possessing its own dedicated VA.

This agent traverses the MLKG, deploying *Translation Agents* for each identified potential vulnerability. This process examines how variables linked to entry points are referenced in subsequent functions along the path leading to a sensitive sink. The VA collaborates with other VAs within the path, sharing insights gleaned from the analysis. Upon reaching the path's terminal function node, it initiates the *Control Flow & Data Agents*, which are elaborated upon later — to assess the existence of an actual vulnerability.

Translation Agent. The *Translation Agent*'s (TrA) job is to figure out how a variable that comes from a specific entry point is translated in the subsequent functions along the designed path. This is done for each FCG_e node in the path, which is fed to the TrA from the VA.

The TrA takes into account the entry point passed from the VA$[i]$ and navigates through the PGD_e of the corresponding i function starting at TrA$[i][k]$ where k corresponds to the node's index of the entry point location. It then communicates with each successor TrA$[i][k..m]$ that is both data and flow depending on the entry point, ensuring there is a flow connection and the variable still exists in that path. This happens until it reaches the TrA$[i][m]$ where m corresponds to the index of the node where the desired function is called (which also corresponds to the function in path$[0]$ in Figure 3, as being the following function on the path). At this point, translation is performed, and the translated entry is passed along to VA$[i+1]$. However, if the agent reaches a sanitisation function instead, it checks if the entry corresponds to the sanitised variable. If it does, the agent stops calling and immediately informs the verification agent that the variable is untainted (i.e., not compromised). Taking the $P1$ path as an example, the VA of `Main` node triggers the TrA to verify in the PDG_e of the function `Main` if the `$age` entry point has any interactions and changes. The result is back to the VA, which forwards it to the VA of the `gymEntrance` node (according to the following function in $P1$). The process repeats unless the final element in the path is a sink. The VA triggers the *Control Flow & Data Agents* in such a case.

Control Flow & Data Agents. The *Control Flow & Data Agents* (FA & DA) are very similar in function but differ in their objective. They are responsible for evaluating flow and data dependencies accordingly in the last step of the evaluation process. They operate on the last PDG_e in the path identified previously by TA and are triggered by VA$[n]$, corresponding to function n where the sensitive sink is located. They communicate with flow-dependent and data-dependent node successors from FA$[n][k]$/DA$[n][k]$ (k being the index where the entry point is located), until they reach FA$[n][m]$/DA$[n][m]$ where the sensitive sink is located. Suppose the entry point matches the one in the sink. In that case, they give feedback to the VA$[n]$ that the pair is flow/data-dependent, so it can conclude that a vulnerability exists. For instance, $P3$ fits in this case. In contrast, if a sanitisation function is reached, similarly to the TrA, they give feedback that the variable is not compromised.

VI. Implementation

We implemented the approach in the KAVE tool [29], using Python. The tool comprises six modules (all built from scratch), each responsible for generating specific types of code representation graphs: *FCG & FCG_e Creator*, *CFG Creator*, *DVG Creator*, and *PDG_e Generator*. Additionally, there is a module dedicated to assembling the MLKG (*MLKG assembler*), which also encompasses the pruning logic, and one designated for MAS, the *MAS Vulnerability Detector*.

We used the NetworkX package [42] as the backbone for our graph structure implementation due to its versatility in managing complex networks, offering capabilities beyond basic graphs. Its flexibility of node representation allows seamless association of diverse data formats, while its edge attributes enable precise differentiation between flow and data dependencies, which is pivotal for our *PDG Generator*.

VII. Experimental Evaluation

To conduct an evaluation of our approach and KAVE, we composed a ground truth of 3,205 PHP code snippets from SARD [43], already labelled as vulnerable and not-vulnerable, and selected 12 open-source PHP web applications that are known as being vulnerable, assessed them with the KAVE and other tools, and wanted to answer the following questions: (1) Is KAVE capable of generating code representations and MLKGs correctly? (2) Can KAVE detect potential vulnerabilities? (3) Is KAVE effective in the vulnerabilities it reports, not generating false positives? (4) Is KAVE more precise than standard SASTs? (5) Is KAVE more effective than tools that resort to standard code property graphs?

A. Assessment with Real Web Applications

1) Web Applications Characterisation:
Our goal drove the selection of applications to compare our model's results against other SASTs. To ensure an equitable comparison, we worked with applications already tested by other vulnerability detection tools. The final selection of applications and their characterisation is detailed in Table II.

KAVE analysed a total of 12 applications with more than 26,000 lines of code (LoC) distributed in 160 files. This procedure resulted in 160 MLKGs (one for each file) representing 257 functions with 1,834 distinct variables and assembling 2,497 graphs (DVGs, CFGs, and FCGs) with 9,518 nodes and 80,487 edges. The tool took 51 seconds to process and analyse the software packages, which comprised the time spent in the generation of the code representation graphs, the MLKG, and the detection of vulnerabilities. We can notice that the number of LoC 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 number of agents needed for the system to operate correlates with the size of the MLKG in terms of nodes. In addition, the number of paths those agents are required to traverse also grows accordingly. It is also noticeable that a higher number of variables relates to a larger MLKG (in terms of nodes and edges), but it does not correlate with the number of files, functions, or graphs.

TABLE II
MLKG Web applications characterisation.

Web Application	Files	LoC	Time(s)	#Functions	#Variables	#Graphs	#Nodes	#Edges
Butterfly insecure	16	2,364	6	25	214	279	1,264	2,905
Butterfly secure	15	2,678	7	25	226	291	1,437	3,113
currentcost	3	270	1	5	67	80	163	319
Ghost	14	398	1	14	34	76	339	422
gilbitron-PIP	14	328	1	14	28	70	171	168
Measureit 1.14	2	967	5	51	266	370	736	2,249
Mfm 0.13	7	5,859	11	20	387	434	2,020	61,585
OWASP Vicnum	22	814	2	22	109	175	310	579
Peruggia	10	988	2	10	87	117	673	967
PHP X Template 0.4	10	3,009	8	10	124	154	663	5,811
Webchess 1.0	37	7,704	5	49	216	343	1351	1,501
ZiPEC 0.32	10	765	2	12	76	108	391	868
Total	**160**	**26,144**	**51**	**257**	**1,834**	**2,497**	**9,518**	**80,487**

TABLE III
Real Application analysis results comparison between KAVE, WAP, Pixy and PHPCorrector.

Web application	KAVE				WAP				Pixy				PHPCorrector			
	SQLi	XSS	FP	FN	SQLi	XSS	FP	FN	SQLi	XSS	FP	FN	SQLi	XSS	FP	FN
Butterfly insecure	0	8	0	2	0	10	0	0	0	6	3	4	0	6	0	4
Butterfly secure	0	5	0	0	0	4	0	1	0	5	11	0	-	-	-	-
currentcost	3	6	0	0	3	4	2	2	3	5	3	1	0	0	0	9
Ghost	1	17	0	1	0	3	0	16	2	15	3	2	1	4	0	19
gilbitron-PIP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Measureit 1.14	0	37	2	1	1	7	7	30	1	16	16	21	-	-	-	-
Mfm 0.13	0	6	0	4	0	8	3	2	0	10	8	0	0	4	0	6
OWASP Vicnum	1	42	3	2	3	1	3	41	3	1	3	41	3	27	3	15
Peruggia	18	8	0	14	0	22	0	18	12	21	10	7	5	10	2	25
PHP X Template 0.4	0	0	0	0	0	0	0	0	-	-	-	-	0	0	0	0
Webchess 1.0	20	59	5	27	0	13	0	93	28	78	403	0	1	42	8	63
ZiPEC 0.32	4	0	0	7	3	0	1	8	2	7	8	2	0	0	0	13
Total	**47**	**188**	**10**	**58**	**10**	**72**	**16**	**211**	**51**	**164**	**468**	**78**	**10**	**93**	**13**	**154**

2) Vulnerability Detection with KAVE*:*
We conducted a manual review of KAVE outcomes to categorise the findings as either XSS or SQLi vulnerabilities in case the predictions were correct or as false positives (FP) if the predictions were incorrect. False negatives (FN) were determined by comparing the vulnerabilities identified by KAVE with those detected by other tools that KAVE failed to find, and likewise for the discrepancies in the findings of these tools compared to each other and KAVE. Columns 2 to 5 of Table III show the evaluation results. KAVE detected a total of 235 vulnerabilities. Among these, 47 were classified as SQLi, while 188 were identified as XSS. Remarkably, a mere 10 FPs were generated in our analysis, a statistically insignificant figure, allowing us to effectively address the question (3).

Demonstrating the tool's ability to identify application vulnerabilities constituted a pivotal milestone for our MAS. This takes us to affirm the tool indeed exhibits proficiency in vulnerability detection (question (2)). Additionally, the MAS's ability to uncover vulnerabilities relies on the MLKG comprising multiple code graphs. Thus, we ascertain the tool's competence in generating and amalgamating these graphs into the MLKG, satisfactorily addressing the question (1).

Derived from our tool's execution, we inferred a precision rate of 95.9%. These statistics illustrate well the potential of our solution. Coupled with the tool's commendable performance in execution time, quantifiable in seconds, we confidently assert that our multi-agent system adeptly traverses the MLKG. This substantiates the tool's capacity to swiftly and precisely identify vulnerabilities within the applications.

B. Effectiveness Comparison

Concerning effectiveness, our initial comparison involves evaluating KAVE against WAP [6], Pixy [12], and PHPCor-

rector [15] standard SAST tools. This assessment hinges on outcomes derived from both real application analyses and testing results conducted across the SARD database [43]. Subsequently, we juxtapose KAVE with NAVEX-f [23], [44], an analogous framework leveraging code property graph analysis, to further reinforce our methodology.

1) Comparison with Standard Static Analysis Tools: We pursued two distinct directions in our comparative analysis of KAVE against other SASTs. Initially, we tested these tools using the ground truth we composed from SARD. This approach established a benchmark for evaluation since SARD labels the examples as Safe or Unsafe. Subsequently, we evaluated and compared the results obtained by running these tools over real applications, recognising that this practical application remains the primary focus of such tools.

Synthetic Ground-Truth dataset. The synthetic ground-truth dataset comprises a total of 3,205 code snippets, with 1,569 labelled as Unsafe and 1,636 as Safe. Among these, only KAVE and WAP successfully tested all the examples, while Pixy and PHPCorrector generated 477 and 148 errors, respectively.

Comprehensive results are collated in Table IV along with the corresponding metrics in Table V. These results show that PHPCorrector exhibited subpar performance, labelling only 28 cases as Unsafe. Despite its reported accuracy of 0.53 (0.5 with error cases) and precision of 0.86, these values lack significance due to the unbalanced predictions, resulting in a meagre 0.03 F1-score. Pixy, on the other hand, seemingly performed well by correctly identifying approximately twice as many examples as KAVE and WAP as Unsafe. However, it tends to have a higher FP rate, which could be attributed to its higher inclination toward labelling code as Unsafe. Yet, when considering true negatives, KAVE demonstrated a capability of correctly labelling around 500 more cases. Moreover, if considered, the substantial number of error cases implies that Pixy's accuracy would diminish from 0.66 to 0.56, aligning more closely with KAVE's performance.

WAP and KAVE, while predicting a similar number of Unsafe cases accurately, showcased disparities. KAVE produced less than half the number of FP, evident from its FP rate of 0.17 compared to WAP's 0.44 (also applicable to Pixy). Additionally, KAVE outperformed WAP in terms of predicting true negatives, having an accuracy of 0.56 against WAP's 0.43, along with a precision of 0.6 compared to WAP's 0.37.

TABLE IV
CONFUSION MATRIX FROM SARD SAMPLES ANALYSIS BY KAVE, WAP, PIXY, AND PHPCORRECTOR.

SARD Label	KAVE		WAP		Pixy		PHPCorrector	
	Unsafe	Safe	Unsafe	Safe	Unsafe	Safe	Unsafe	Safe
Unsafe	429	1140	452	1117	984	357	24	1437
Safe	286	1350	719	917	566	821	4	1592

TABLE V
METRICS FROM SARD SAMPLES ANALYSIS BY KAVE, WAP, PIXY, AND PHPCORRECTOR.

Metric	KAVE	WAP	Pixy	PHPCorrector
Accuracy (acc)	0.56	0.43	0.66	0.53
Recall (rec)	0.27	0.29	0.73	0.02
Precision (pr)	0.60	0.37	0.64	0.86
F1-score	0.38	0.33	0.68	0.03
FP rate (fpr)	0.17	0.44	0.41	0.00
FN rate (fnr)	0.73	0.71	0.27	0.98

While the benchmark methodology offers a valuable indication of the tools' validity, it is crucial to acknowledge that most of this benchmark comprises synthetic examples that frequently do not align with real-world vulnerabilities. Additionally, numerous examples exhibit subtle nuances, implying that a tool inclined to mislabel one example as Safe will likely repeat this error across similar instances, creating bias. Conversely, if it mislabels an example as Unsafe, it may propagate this misclassification, potentially amplifying bias in the opposite direction.

Real Applications. With the real application, only KAVE and WAP were executed without encountering errors. Pixy failed to process one application entirely and reported errors across 15 files within other applications. Similarly, PHPCorrector had issues processing two applications and reported 7 additional file errors. Indeed, in contrast to the synthetic dataset analysis, where the focus was on categorising code as Safe or Unsafe, this evaluation goes beyond mere labelling. Here, the emphasis lies in accurately quantifying the actual number of vulnerabilities within the applications and precisely identifying their specific locations. This approach enables a more granular understanding of the vulnerabilities present, facilitating their targeted resolution and enhancing the overall effectiveness of the assessment process. The summarised outcomes are presented in Table III and corresponding metrics in Table VI.

TABLE VI
METRICS FROM REAL APPLICATION ANALYSIS BY KAVE, WAP, PIXY, AND PHPCORRECTOR.

Metric	KAVE	WAP	Pixy	PHPCorrector
Precision (pr)	0.96	0.84	0.32	0.89
Recall (rec)	0.80	0.28	0.73	0.40
F1-score	0.87	0.42	0.44	0.55

This analysis reveals that KAVE identified the highest number of vulnerabilities, totalling 235 compared to WAP's 82, Pixy's 215, and PHPCorrector's 103. Despite Pixy's seemingly substantial vulnerability identification, it is linked to a notably higher number of FP, underscoring its tendency to label code as Unsafe, as observed in the Synthetic Ground-Truth. This characteristic renders Pixy unsuitable for real-world scenarios, notably being the only tool falling below the 0.8 precision threshold. KAVE, WAP, and PHPCorrector exhibited a similar number of FP, thereby reflecting comparable precision levels. However, KAVE was still superior and surpassed the others by not only identifying more than double the number of vulnerabilities but also exhibiting notably fewer FN. Consequently, KAVE achieved better recall, resulting in a better f1-score when compared to the others. This allows us to answer question (4) affirmatively.

2) Comparison with Code Property Graphs: In our final comparison, we sought to validate the effectiveness of KAVE by contrasting it NAVEX-f, a tool that uses code property graphs for static analysis. To facilitate this comparison, we selected 6 real applications previously scrutinised by NAVEX-f. The analysis results, combined with findings from

TABLE VII
REAL APPLICATION ANALYSIS RESULTS IN COMPARISON BETWEEN KAVE AND NAVEX-F.

Web application	KAVE				NAVEX-f			
	SQLI	XSS	FP	FN	SQLI	XSS	FP	FN
60CycleCMS	5	18	2	0	1	5	3	17
AMSS++	1323	2476	62	0	-	-	-	3799
CandidatATS	0	140	5	15	15	35	10	105
eLection	12	26	10	0	0	0	20	38
GUnet OpenEclass E-learning	1	716	14	0	0	0	1	717
Persian VIP Download Script	21	7	0	1	15	8	3	6
rConfig	2	32	3	0	0	11	7	23
Total	**1364**	**3415**	**96**	**16**	**31**	**59**	**44**	**4705**

[45], are consolidated and presented in Table VII.

As anticipated, by analysing prior NAVEX-f results comparisons [45], KAVE showcased superior performance over NAVEX-f in identifying vulnerabilities, which allows us to answer affirmatively to question (5). Despite having twice as many FP, this was largely influenced by NAVEX-f's inability to process AMSS+, the application housing the most vulnerabilities, thereby increasing the likelihood of generating FP. NAVEX-f exhibited superiority solely in detecting SQL injection in CandidatATS and discovering one additional XSS in Persian. However, in the broader context of results, these instances appear relatively inconsequential.

VIII. RELATED WORK

This section contains some related works, and although there are several research works on detecting web vulnerabilities, we only present the ones that are more related to ours.

SASTs are critical in identifying vulnerabilities in source code that can make it susceptible to attacks. These tools have been developed over the years to help programmers identify potential security risks in their code. One such tool (the pioneer) is the bounded model checking method developed by Huang et al. [4], [5], which uses a lattice-based algorithm approach that taints code through type systems and typestate. The approach is sound and provides counter-examples, making it useful in identifying potential vulnerabilities. Another work in the field of SASTs is the development of the dependence analysis method proposed by [18]. The method evaluates program operations to produce execution order constraints that control whether an operation data depends on another operation and should be executed only after the previous one. The author takes advantage of merging PDGs from single application procedures within a call graph, generating a System Dependence Graph (SDG) [46] representing the entire program structure. This method is useful in identifying potential vulnerabilities in dynamic languages such as PHP. Pixy [12], PHPCorrector [15], RIPS [14], and WAP [6] are SASTs for web application vulnerability discovery based on AST and taint analysis. PHPCorrector and WAP also remove the vulnerabilities by fixing the application code. SKYPORT [47] also patches web applications that contain injection vulnerabilities, but first, searches the code looking for them.

Other tools resort to different techniques for web vulnerability detection. FuzzOrigin [48] is a black-box to detect universal XSS (UXSS) in web browsers. PIDGIN [20] aims to find security guarantees in legacy programs to create new or adjust existing policies with application development and develop policies based on known vulnerabilities. Leopard [24] also identifies vulnerabilities in program code but through metrics that allow measuring program code elements. Kassar et al. [49] and Medeiros et al. [50], [51] study the impact of coding style on SASTs ability to discover vulnerabilities in PHP.

Backes et al. [21] used inter-procedural analysis techniques on code property graphs (introduced by [7], [22]) to efficiently analyse large amounts of code, storing them in highly efficient graph databases that allow users to query for vulnerabilities. On the other hand, MERLIN [16] is a tool that detects vulnerabilities in web applications by combining data flow analysis over intermediate code parsed from Java Bytecode generated from multiple high-level languages. It uses taint analysis and machine learning to automatically discover vulnerabilities by classifying code as vulnerable or not. BEACON [52] is a new method for grey-box fuzzing that aims to improve the effectiveness of fuzz testing in web applications by combining symbolic execution, machine learning, and provable path pruning. FUGIO [53] is an automatic exploit generation tool designed to exploit PHP Object Injection vulnerabilities. After finding the vulnerabilities, the tool generates exploits and executes them to confirm its findings. VulEye [54] 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 PHP code, slices the PDG with sensitive functions contained in the source code into sub-graphs called Sub-Dependence Graphs, and uses these as input for a GNN model.

IX. CONCLUSIONS

This paper presented the KAVE system approach to detect web vulnerabilities based on multi-layer knowledge graphs (MLKGs) constructed from diverse code representation graphs and on a multi-agent system (MAS) to discover vulnerabilities over these MLKGs. This research has added to the existing knowledge in the field by providing KAVE that implements the presented approach and has demonstrated the capability of effectively finding vulnerabilities in PHP coded applications. The findings from KAVE provide valuable insights that can inform programmers and entities if their web applications are vulnerable and where in their code, which can contribute to improving code quality and software security. Moreover, the results reveal that, in many situations, KAVE obtained results better or at least comparable to the ones of other existing tools for vulnerability detection.

ACKNOWLEDGMENTS

This work was supported by FCT through the LASIGE Research Unit, ref. UIDB/00408/2020 (https://doi.org/10.54499/UIDB/00408/2020) and ref. UIDP/00408/2020 (https://doi.org/10.54499/UIDP/00408/2020). It is based upon work from COST Action CA22104 – Behavioral Next Generation in Wireless Networks for Cyber Security (BEiNG-WISE), supported by COST (European Cooperation in Science and Technology) www.cost.eu.

REFERENCES

[1] J. Daley, "Insecure software is eating the world: Promoting cybersecurity in an age of ubiquitous software embedded systems.," *Stanford Technology Law Review*, vol. 19, no. 3, 2017.

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

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

[4] 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*, WWW '04, (New York, NY, USA), p. 40–52, Association for Computing Machinery, 2004.

[5] 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*, DSN '04, (USA), p. 199, IEEE Computer Society, 2004.

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

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

[8] O. Foundation, "Open web application security project." https://www.owasp.org/, 2023.

[9] T. M. Corporation, "Common weakness enumeration." https://cwe.mitre.org/, 2023.

[10] W3Techs - Web Technology Surveys, "Usage statistics of server-side programming languages for websites." https://w3techs.com/technologies/overview/programming_language, 2023.

[11] T. D. Oyetoyan, B. Milosheska, M. Grini, and D. S. Cruzes, "Myths and facts about static application security testing tools: An action research at telenor digital," in *Agile Processes in Software Engineering and Extreme Programming - 19th International Conference, XP 2018, Porto, Portugal, May, Proceedings* (J. Garbajosa, X. Wang, and A. Aguiar, eds.), vol. 314 of *Lecture Notes in Business Information Processing*, pp. 86–103, Springer, 2018.

[12] 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)*, pp. 6 pp.–263, 2006.

[13] P. Nunes, J. Fonseca, and M. Vieira, "phpSAFE: A security analysis tool for OOP web application plugins," in *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2015.

[14] J. Dahse and T. Holz, "Simulation of built-in PHP features for precise static code analysis," in *Proceedings of the Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February*, The Internet Society, 2014.

[15] R. Morgado, I. Medeiros, and N. Neves, "Towards web application security by automated code correction," in *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering*, pp. 86–96, Apr. 2020.

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

[17] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proceedings of the 33rd Conference on Advances in Neural Information Processing Systems*, pp. 10197–10207, Dec. 2019.

[18] M. Wijngaard, "Dependence analysis in php," Aug. 2016.

[19] 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, p. 319–349, jul 1987.

[20] A. Johnson, L. Waye, S. Moore, and S. Chong, "Exploring and enforcing security guarantees via program dependence graphs," *SIGPLAN Not.*, vol. 50, p. 291–302, jun 2015.

[21] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and flexible discovery of PHP application vulnerabilities," in *Proceedings of the 2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 334–349, Apr. 2017.

[22] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, pp. 797–812, May 2015.

[23] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan, "NAVEX: Precise and scalable exploit generation for dynamic web applications," in *Proceedings of the 27th USENIX Security Symposium*, pp. 377–392, Aug. 2018.

[24] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang, "Leopard: identifying vulnerable code for vulnerability assessment through program metrics," in *In Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May* (J. M. Atlee, T. Bultan, and J. Whittle, eds.), pp. 60–71, IEEE / ACM, 2019.

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

[26] L. Panait and S. Luke, "Cooperative multi-agent learning: The state of the art," *Springer*, 2005.

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

[28] 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.

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

[30] A. C. for Schools, "Western association of schools and colleges." https://www.acswasc.org/, 2023.

[31] Kingthorin, "Sql injection." https://owasp.org/www-community/attacks/SQL_Injection, 2023.

[32] S. Kost, "An introduction to sql injection attacks for oracle developers," 2007.

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

[34] D. Wichers, A. Dabirsiaghi, S. D. Paolo, M. Heiderich, E. A. V. Nava, and J. Williams, "Types of xss." https://owasp.org/www-community/Types_of_Cross-Site_Scripting, 2023.

[35] R. Johnson, D. Pearson, and K. Pingali, "The program structure tree: Computing control regions in linear time," PLDI '94, (New York, NY, USA), p. 171–185, Association for Computing Machinery, 1994.

[36] F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, p. 1–19, jul 1970.

[37] M. De Domenico, A. Solé-Ribalta, E. Cozzo, M. Kivelä, Y. Moreno, M. A. Porter, S. Gómez, and A. Arenas, "Mathematical formulation of multilayer networks," *Physical Review X*, vol. 3, no. 4, p. 041022, 2013.

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

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

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

[41] D. C. Smith, A. Cypher, and J. Spohrer, "Kidsim: Programming agents without a programming language," *Commun. ACM*, vol. 37, p. 54–67, jul 1994.

[42] P. J. S. Aric A. Hagberg, Daniel A. Schult, "Exploring network structure, dynamics, and function using networkx," 2008.

[43] National Institute of Standards and Technology (NIST), "NIST Software Assurance Reference Dataset (SARD)," 2023. Accessed December 15, 2023.

[44] NAVEX-fixed., 2019. https://github.com/UUUUnotfound/Navex_fixed.

[45] I. Medeiros, N. Neves, and M. Correia, "Statically detecting vulnerabilities by processing programming languages as natural languages," *IEEE Transactions on Reliability*, vol. 71, no. 2, pp. 1033–1056, 2022.

[46] J. Graf, "Speeding up context-, object- and field-sensitive sdg generation," in *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, pp. 105–114, 2010.

[47] Y. Shi, Y. Zhang, T. Luo, X. Mao, Y. Cao, Z. Wang, Y. Zhao, Z. Huang, and M. Yang, "Backporting security patches of web applications: A prototype design and implementation on injection vulnerability patches," in *31st USENIX Security Symposium*, pp. 1993–2010, Aug. 2022.

[48] 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), pp. 1008–1023, USENIX Association, Aug. 2022.

[49] F. A. Kassar, G. Clerici, L. Compagna, D. Balzarotti, and F. Yamaguchi, "Testability tarpits: the impact of code patterns on the security testing of web applications," in *29th Annual Network and Distributed System Security Symposium*, Apr. 2022.

[50] I. Medeiros and N. Neves, "Effect of coding styles in detection of web application vulnerabilities," in *16th European Dependable Computing Conference*, pp. 111–118, 2020.

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

[52] 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)*, pp. 36–50, 2022.

[53] 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), pp. 197–214, USENIX Association, Aug. 2022.

[54] 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.