Statically Detecting Vulnerabilities by Processing Programming Languages as Natural Languages

Ibéria Medeiros, *Member, IEEE,* Nuno Neves, *Member, IEEE,* and Miguel Correia, *Senior Member, IEEE*

Abstract—Web applications continue to be a favorite target for hackers due to a combination of wide adoption and rapid deployment cycles, which often lead to the introduction of high impact vulnerabilities. Static analysis tools are important to search for vulnerabilities automatically in the program source code, supporting developers on their removal. However, building these tools requires programming the knowledge on how to discover the vulnerabilities. This paper presents an alternative approach in which tools *learn* to detect flaws automatically by resorting to artificial intelligence concepts, more concretely to natural language processing. The approach employs a sequence model to learn to characterize vulnerabilities based on an annotated corpus. Afterwards, the model is utilized to discover and identify vulnerabilities in the source code. It was implemented in the DEKANT tool and evaluated experimentally with a large set of PHP applications and WordPress plugins. Overall, we found several thousand vulnerabilities belonging to 15 classes of input validation vulnerabilities, where 4143 of them were zero-day.

Index Terms—vulnerabilities, web application, software security, static analysis, sequence models, natural language processing.

1 Introduction

Web applications are being used to implement interfaces of a myriad of services. They are often the first target of attacks, and despite considerable efforts to improve security, there are still many examples of high impact compromises. In the 2017 OWASP Top 10 list, vulnerabilities like SQL injection (SQLi) and cross-site scripting (XSS) continue to raise significant concerns, but other classes are also listed as being commonly exploited [1], [2]. Millions of websites have been compromised since Oct. 2014 due to vulnerabilities in plugins of Drupal [3] and WordPress [4], [5], and the data of more than a billion users has been stolen using SQLI attacks against various kinds of services (governmental, financial, education, mail, etc.) [6], [7]. In addition, the next wave of XSS attacks has been predicted since the past three years and the number of vulnerabilities in third-parties has increased, with an expected growth of the problem [8], [9], [2].

Many of these vulnerabilities are related to malformed inputs that reach some relevant asset (e.g., the database or the user's browser) by traveling through a code *slice* (a sequence of instructions from an input to a security-sensitive instruction) of the web application. Therefore, a good practice to enhance security is to pass inputs through *sanitization functions* that invalidate dangerous metacharacters or/and *validation functions* that check their content. In addition, programmers commonly use *static analysis tools* to search automatically for bugs in the source code, facilitating their removal. The development of these tools, however, requires coding explicitly the knowledge on how each vulnerability can be detected [10], [11], [12], [13], which is a complex task. For instance, after representing the code in an abstract syntax tree (AST), the tools are programmed to navigate along the AST, deal with the semantic aspects of the language and determine whether the information it contains is associated with the vulnerability classes they seek. Moreover, this knowledge might be incomplete or partially wrong, making the tools inaccurate [14]. For example, if the tools do not understand that a certain function sanitizes inputs, they could raise an alert about a vulnerability that does not exist.

This paper presents a new approach for static analysis that is based on *learning to recognize vulnerabilities*. In other words, the proposed approach, unlike classical static analysis, learns to detected vulnerabilities withouthaving programmers code the knowledge required to find them. The approach leverages from artificial intelligence (AI) concepts to perform the analysis. AI-based approaches have been emerging in the field of static analysis [15]. They are able to compose machine learning (ML) models and to obtain results quickly, with the support of ML toolkits (e.g., NLTK, Keras) that offer several algorithms that can be applied in different contexts. In addition, the models are capable of learning security features from slices of code provided publicly (e.g., NIST SARD [16]) and to process new slices, predicting whether they are vulnerable [17], [18], [19], [20], [21].

The approach we propose leverages from classification models for sequences of observations that are commonly used in the field of natural language processing (NLP) for analysis of written language, i.e., the natural languages. In this sense, NLP is considered a sub-area of AI. It can be viewed as a new form of intelligence in an artificial way that can get insights on how humans understand natural languages. NLP tasks, such as parts-of-speech (PoS) tagging or named entity recognition (NER), are typically modelled as sequence classification problems, in which a class (e.g., a

- *I. Medeiros and N. Neves are with the LASIGE, Faculdade de Ciências, Universidade de Lisboa - Portugal (e-mail: imedeiros@di.fc.ul.pt and nuno@di.fc.ul.pt).*
- *M. Correia is with the INESC-ID, Instituto Superior Técnico, Universidade de Lisboa - Portugal (e-mail: miguel.p.correia@tecnico.ulisboa.pt).*

given morpho-syntactic category) is assigned to each word in a given sentence, according to an estimate given by a structured prediction model that takes word order into consideration. The model's parameters are normally inferred using supervised machine learning techniques, taking advantage of annotated corpora.

We propose applying a similar approach to web programming languages, i.e., to analyse source code in a similar manner to what is being done with natural language text. Even though these languages are artificial, they have many characteristics in common with natural languages, such as words, syntactic rules, sentences, and a grammar. NLP usually employs machine learning to extract rules (knowledge) automatically from a *corpus*. Then, with this knowledge, other sequences of observations can be processed and classified. NLP has to take into account the *order* of the observations, as the meaning of sentences depends on it. Therefore, NLP involves forms of classification more sophisticated than approaches based on *standard classifiers* (e.g., naive Bayes, decision trees, support vector machines), which simply check the presence of certain observations without considering any relation between them.

Our approach for static analysis resorts to machine language techniques that take the order of source code elements within instructions into account – *sequence models* – to allow accurate detection and identification of the vulnerabilities in the code. Previous applications of machine learning in the context of static analysis neither produced tools that learn to make detection nor employed sequence models. For example, PhpMinerII [22], [23] resorts to machine learning to train standard classifiers, which then verify if certain constructs (associated with flaws) exist in the code. However, it does not provide the exact location of the vulnerabilities [22], [23]. WAP and WAPe use a taint analyser to search for vulnerabilities and three standard classifiers to confirm that the found bugs[1] can actually create security problems [13]. None of these tools considers the order of code elements or the relation among them, leading to bugs being missed (*false negatives, FN*) and alarms being raised on correct code (*false positives, FP*). On the other hand, although deep learning approaches can consider the relation among code elements [18], [17], [19], their black-box nature prevents them from explaining such relations, which can also raise FPs.

Our sequence model is a *Hidden Markov Model* (HMM) [24]. A HMM is a Bayesian network composed of nodes corresponding to the states and edges associated to the probabilities of transitioning between states. States are hidden, i.e., are not observed. Given a sequence of observations, the hidden states (one per observation) are discovered following the model and taking into account the order of the observations. Therefore, the HMM can be used to find the series of states that *best* explains the sequence of observations.

The paper also presents the *hidDEn marKov model diAgNosing vulnerabiliTies* (DEKANT) tool that implements our approach for applications written in PHP, the most used language for developing web applications [25]. The tool was evaluated experimentally with a diverse set of 21 open source web applications, where 13 of them with bugs disclosed in the past, and found *3938 zero-day vulnerabilities* in the remaining 8 web applications. These 21 applications are substantial, with an aggregated size of around 7,000 files and 1.8 million lines of code (LoC). All flaws that we are aware of being previously reported were found by DEKANT. More than six thousand slices were analyzed, 4,646 were classified as having vulnerabilities and 1,404 as not. The false positives were in the order of one hundred. In addition, the tool checked 25 plugins of WordPress and found *205 zero-day vulnerabilities*. These flaws were reported to the developers, and some of them confirmed their existence and fixed the plugins. DEKANT was also assessed with several other vulnerability detection tools, and the results give evidence that our approach leads to better accuracy and precision.

This paper extends our previous work [26] with the following: (1) provides more details about ISL grammar tokens and the PHP functions they represent; (2) gives a detailed description of the corpus construction process, including an example of this process, and the corpus assessment, containing the three phases that allow the entire corpus to be evaluated with better completeness; (3) explains the implementation of DEKANT, including the extensions made to the Viterbi algorithm, namely the algorithms of the interactions *beforeVit* and *afterVit* that process sequences of observations for vulnerability detection, and their integration in the Viterbi algorithm; (4) a new experimental evaluation using more WordPress plugins and real web applications and other tools; (5) a detailed related work section.

The main contributions of the paper are: (1) a novel approach for improving the security of web applications by letting static analysis tools learn to discover vulnerabilities through an annotated corpus; (2) an intermediate language representation capturing the relevant features of PHP, and a sequence model that takes into consideration the place where code elements appear in the slices and how they alter the spreading of the input data; (3) a NLP-based static analysis tool that implements the approach; (4) an experimental evaluation that demonstrates the ability of this tool to find known and zero-day vulnerabilities with a residual number of mistakes.

The remaining of the paper is organized as follows: the next section presents in general the surface vulnerabilities our model addresses. Section 3 gives an overview of the proposed approach for detecting vulnerabilites. Sections 4, 5, and 6 detail the ISL language, the sequence model that processes slices in ISL, classifying them as vulnerable or not-vulnerable, and the construction and assessment of the corpus used by the model. Section 7 presents the DEKANT tool that implements the approach, and Section 8 shows the evaluation of the tool. Section 9 and Section 10 present, respectively, the threats to validity that the model can face in the experiments and the applicability of the tool in the real-world. Finally, the related work regarding vulnerability detection is explained in Section 11, and conclusions are presented in Section 12.

2 Surface Vulnerabilities

Many classes of security flaws in web applications are caused by improper handling of user inputs. Therefore,

1. In software security context, we consider a vulnerability as a being a bug or a flaw that could be exploitable.

PHP code	slice-isl	variable map	tainted list	slice-isl classification
1 $u = $_POST['username'];	input var	1 - u	TL = {u}	⟨input,Taint⟩ ⟨var_vv_u,Taint⟩
2 $q = "SELECT pass FROM users WHERE user='".$u."'";	var var	1 u q	TL = {u, q}	⟨var_vv_u,Taint⟩ ⟨var_vv_q,Taint⟩
3 $r = mysqli_query($con, $q);	ss var var	1 - q r	TL = {u, q, r}	⟨ss,N-Taint⟩ ⟨var_vv_q,Taint⟩ ⟨var_vv_r,Taint⟩

(a) code with SQLI vulnerability (b) *slice-isl and variable map* (c) *artefact list* (d) outputting the final classification

Fig. 1: Code vulnerable to SQLI, translation into ISL, and detection of the vulnerability.

PHP code	slice-isl	variable map	list
1 $u = (isset($_POST['name']) ? $u = $_POST['name'] : ''; 2 $a = $_POST['age']; 3 if (isset($a) && preg_match('/[a-zA-Z]+/', $u) && is_int($a)){	input var input var cond fillchk var contentchk var typechk var cond	1 - u 1 - a 0 - - a - u - a -	TL = {u}; CTL = {} TL = {u, a}; CTL = {} TL = {u, a}; CTL = {u, a}
4 echo "The user age is " . $a; 5 <input type="hidden" name="user" value=<?php echo $u; ?>> 6 } else 7 echo $u . "is an invalid user";	cond ss var cond ss var cond ss var	0 - - a 0 - - u 0 - 0 - u	TL = {u, a}; CTL = {u, a} TL = {u, a}; CTL = {u, a} TL = {u, a}; CTL = {} TL = {u, a}; CTL = {}

(a) code with XSS vulnerability and validation (b) *slice-isl* and variable map (c) artefacts lists

Fig. 2: Code with a slice vulnerable to XSS (lines {1, 2, 3, 6, 7}) and two slices not vulnerable (lines {1, 2, 3, 4} and {1, 2, 3, 5}), with ISL translation.

they are denominated *surface vulnerabilities* or *input validation vulnerabilities*. In PHP programs the malicious input arrives to the application (e.g, *$_POST*), then it may suffer various modifications and might be copied to variables, and eventually reaches a security-sensitive function (e.g., *mysqli_query* or *echo*) inducing an erroneous action. Below, we introduce the 15 classes of surface vulnerabilities that will be considered in rest of the paper.

(1) SQLI is the class of vulnerabilities with highest risk in the OWASP Top 10 list [1]. Normally, the malicious input is used to change the behavior of a query to a database to provoke the disclosure of private data or corrupt the tables.

Example 1. The PHP script of Fig. 1 (a) has a simple SQLI vulnerability. `$u` receives the username provided by the user (line 1), and then it is inserted in a query (lines 2-3). An attacker can inject a malicious username like `' OR 1 = 1 --` , modifying the structure of the query and getting the passwords of all users.

(2) XSS vulnerabilities allow attackers to execute scripts in the users' browsers. Below we give an example:

Example 2. The code snippet of Fig. 2 (a) has a XSS vulnerability. If the user provides a name, it gets saved in `$u` (line 1). Then, if the conditional validation is false (line 3), the value is returned to the user by `echo` (line 7). A script provided as input would be executed in the browser, possibly carrying out some malicious deed.

The other classes are presented briefly. (3), (4) Remote and local file inclusion (RFI/LFI) flaws also allow attackers to insert code in the vulnerable web application. While in RFI the code can be located in another web site, in LFI it has to be in the local file system (but there are also several strategies to put it there). (5) OS command injection (OSCI)) lets an attacker provide commands to be run in a shell of the OS of the web server. (6) Attackers can supply code that is executed by a `eval` function by exploring PHP command injection (PHPCI) bugs. (7)–(10) Like SQLI, LDAP injection (LDAPI), XPath injection (XPathI), NoSQL injection (NoSQLI), and SQLite injection (SQLiteI) are associated with the construction and execution of queries or filters in an engine, e.g., a database. (11), (12) An attacker can read files from the local file system by exploiting directory traversal / path traversal (DT/PT) and source code disclosure (SCD) vulnerabilities. (13) A comment spamming (CS) bug is related to the ranking manipulation of spammers' web sites. (14) Header injection or HTTP response splitting (HI) allows an attacker to manipulate the HTTP response. (15) An attacker can force a web client to use a session ID he defined by exploiting a session fixation (SF) flaw.

3 Overview of the Approach

Our approach for vulnerability detection examines program slices to determine if they contain a bug. The slices are collected from the source code of the target application, and then their instructions are represented in an intermediate language developed to express features that are relevant to surface vulnerabilities. Bugs are found by classifying the translated instructions with an HMM sequence model. Since the model has an understanding of how the data flows are affected by operations related to sanitization, validation and modification, it becomes feasible to make an accurate analysis. In order to setup the model, there is a *learning phase* where an annotated corpus is employed to derive the knowledge about the different classes of vulnerabilities. Afterwards, the model is used in a *detection phase* to detect vulnerabilities. Fig. 3 illustrates this procedure.

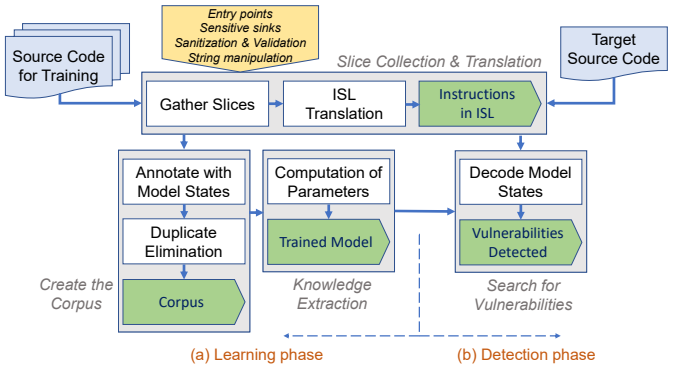


Fig. 3: Overview on the proposed approach.

In more detail, the following steps are carried out. The *learning phase* is composed mainly of steps (1)-(3) while the *detection phase* encompasses (1) and (4):

(1) *Slice collection and translation:* get the slices from the application source code (either for learning or detection). Since we are focusing on surface vulnerabilities, the only slices that have to be considered need to start at some point in the program where an user input is received (i.e., at an *entry point*) and then they have to end at a security-sensitive instruction (i.e., a *sensitive sink*). The resulting slice is a series of tracked instructions between the two points,

which can include, for instance, if-statements, sanitization and validations functions. Then, each instruction of a slice is represented into the *Intermediate Slice Language* (ISL). ISL is a categorized language with grammar rules that aggregate in classes the code elements by functionality. A slice in the ISL format is going to be named as *slice-isl*;

(2) *Create the corpus:* build a corpus with a group of instructions represented in the intermediate language, which are labeled either as vulnerable or non-vulnerable. The instructions are provided individually or gathered from slices of training programs. Overall, the corpus includes both representative pieces of programs that have various kinds of flaws and that handle inputs adequately;

(3) *Knowledge extraction:* acquire knowledge from the corpus to configure the HMM sequence model, namely compute the probability matrices;

(4) *Search for Vulnerabilities:* use the model to find the best sequence of states that explains a slice in the intermediate language. Each instruction in the slice-isl corresponds to a sequence of observations. These observations are classified by the model, tracking the variables from the previous instructions to find out which states are outputted. The state computed for the last observation of the last instruction determines the overall classification, either as vulnerable or not. If a flaw is found, an alert is reported including the location in the source code.

The next two sections explain the ISL language and the sequence model (Sections 4 and 5). The four above steps are elaborated in Sections 4, 5, and 6, more specifically step (1) in Section 4, step (2) in Section 6, and steps (3) and (4) in Section 5. An overview of the tool that implements our approach is given in Section 7.

4 Intermediate Slice Language

All slices commence with an entry point and finish with a sensitive sink; between them there can be an arbitrary number of statements, such as assignments that transmit data to intermediate variables and various kinds of expressions that validate or modify the data. In other words, a slice contains all instructions (lines of code) that manipulate and propagate an input arriving at an entry point and until a sensitive sink is reached, but no other statements.

ISL expresses an instruction into a few tokens. The instructions are composed of *code elements* that are categorized in *classes* of related items (e.g., class `input` takes PHP entry points like `$_GET` and `$_POST`). Therefore, classes are the *tokens* of the ISL language and these are organized together accordingly to a grammar. Next we give a more careful explanation of ISL assuming that the source code is programmed in the PHP language. However, the approach is generic and other languages could be considered.

4.1 Tokens

ISL abstracts away aspects of the PHP language that are irrelevant to the discovery of surface vulnerabilities. Therefore, as a starting point to specify ISL, it was necessary to identify the essential tokens. To achieve this, we followed an iterative approach where we began with an initial group of tokens which were gradually refined. In every iteration, we examined various slices (vulnerable and not) to recognize the important code elements. We also looked at the PHP functions that could manipulate entry points and be associated with bugs or prevent them (e.g., functions that replace characters in strings), since they can change the maliciousness of inputs [13], [27]. In addition, for the PHP functions that were picked, we studied cautiously their parameters to determine which of them are crucial for our analysis.

Example 3. Function *mysqli_query* and its parameters correspond to two tokens: `ss` for sensitive sink; and `var` for variable or `input` if the parameter receives data originating from an entry point. Although this function has three parameters (the last of them optional), notice that just one of them (the second) is essential to represent.

Overall, we studied all PHP functions of different categories, namely the sensitive sinks and sanitization functions regarding to the surface vulnerabilities we consider, and those that modify strings and verify the content of strings and numbers. Afterwards, we selected the ones relevant for bug detection. In the end, we defined around twenty tokens that are sufficient to describe the instructions of a PHP program and are capable of representing 168 PHP code elements, 160 of which are PHP functions and the remaining 8 are operations, function parameters, and the `if` statement instruction.

Table 1 summarizes the currently defined ISL tokens. The first column shows above the twenty tokens that stand for PHP code elements, whereas the last two tokens are necessary only for the description of the corpus and the implementation of the model (see Sections 5 and 6). The next three columns explain succinctly the purpose of the token, and show the code elements it represents, as well as the number of them it comprises. Column fifth defines the taintedness status of each token which is used when building the corpus or performing the analysis of slice-isl. Some tokens have their status well defined due to their nature, i.e., they are always tainted (e.g., `input`) or untainted (e.g., `sanit_f`), whereas others have not, wherein it depends on the state of the analysis in a given instant, i.e., the status of code elements already processed.

A more cautious inspection of the tokens shows that they enable many relevant behaviors to be expressed. For example, since the manipulation of strings plays a fundamental role in the exploitation of surface vulnerabilities, there are various tokens that enable a precise modeling of these operations (e.g., `erase_str` or `sub_str`). Tokens `char5` and `char6` represent parameters of such functions and they act as the amount of characters that are manipulated by functions that extract or replace the contents from a user input, respectively up to 5 and 6 or more characters. Moreover, `char6` represent possible malicious code, since the length of malicious code, generally, is greater than 5 (we observed this after inspecting several injected codes of different vulnerability classes). The place in a string where modifications are applied (begin, middle or end) is described by `start_where`; Token `cond` can correspond to an `if` statement that might have validation functions over variables (e.g., user inputs) as part of its conditional expression. This token allows the correlation among the validated variables and the variables that appear inside

TABLE 1: Intermediate Slice Language tokens.

Token	Description	PHP code elements	#Code elements	Taintedness
input	entry point	$_GET, $_POST, $_COOKIE, $_REQUEST $_HTTP_GET_VARS, $_HTTP_POST_VARS $_HTTP_COOKIE_VARS, $_HTTP_REQUEST_VARS $_FILES, $_SERVERS	10	Yes
sanit_f	sanitization function	mysql_escape_string, mysql_real_escape_string mysqli_escape_string, mysqli_real_escape_string mysqli_stmt_bind_param, mysqli::escape_string mysqli::real_escape_string, mysqli_stmt::bind_param db2_escape_string, pg_escape_string, pg_escape_bytec sqlite_escape_string htmlentities, htmlspecialchars, strip_tags, urlencode escapeshellcmd, escapeshellarg	18	No
ss	sensitive sink	mysql_query, mysql_unbuffered_query, mysql_db_query mysqli_query, mysqli_real_query, mysqli_master_query mysqli_multi_query, mysqli_stmt_execute, mysqli_execute mysqli::query, mysqli::multi_query, mysqli::real_query mysqli_stmt::execute db2_exec, pg_query, pg_send_query sqlite_query, sqlite_exec, sqlite_array_query sqlite_single_query, sqlite_unbuffered_query ldap_add, ldap_delete, ldap_list, ldap_read, ldap_search xpath_eval, xptr_eval, xpath_eval_expression fopen, file_get_contents, file, copy, unlink, move_uploaded_file imagecreatefromgd2, imagecreatefromgd2part, imagecreatefromgd imagecreatefromgif, imagecreatefromjpeg, imagecreatefrompng imagecreatefromstring, imagecreatefromwbmp imagecreatefromxbm, imagecreatefromxpm require, require_once, include, include_once readfile passthru, system, shell_exec, exec, pcntl_exec, popen echo, print, printf, sprintf, vprintf, die, error, exit file_put_contents, file_get_contents, vfprintf, fprintf, fscanf eval setcookie, setdrawcookie, session_id find, findOne, findAndModify, insert, remove, save, execute header, mail	82	Yes
typechk_str	type checking string function	is_string, ctype_alpha, ctype_alnum, str_val	4	Yes
typechk_num	type checking numeric function	is_int, is_double, is_float, is_integer, intval, boolval is_long, is_numeric, is_real, is_scalar, ctype_digit, floatval	12	No
contentchk	content checking function	preg_match, preg_match_all, ereg, eregi strnatcmp, strcmp, strncmp, strncasecmp, strcasecmp	9	No
fillchk	fill checking function	isset, empty, is_null	3	Yes
join_str	join string function	implode, join	2	No
erase_str	erase string function	trim, ltrim, rtrim	3	Yes
replace_str	replace string function	preg_replace, preg_filter, str_ireplace, str_replace ereg_replace, eregi_replace, str_shuffle, chunk_split	8	No
split_str	split string function	str_split, preg_split, explode, split, spliti	5	Yes
add_str	add string function	str_pad	2	Yes/No
sub_str	substring function	substr	1	Yes/No
sub_str_replace	replace substring function	substr_replace	1	Yes/No
char5	substring with less than 6 chars	–	1	No
char6	substring with more than 5 chars	–	1	Yes
start_where	where the substring starts	–	1	Yes/No
cond	*if* instruction	if, else	2	No
conc	concatenation operator	–	2	Yes/No
var	variable	–	1	No
var_vv	variable tainted	–		Yes
miss	miss value	–		Yes/No

the `if` branches. Token `conc` expresses the concatenation operation, which also plays an important role in string manipulation.

There are a few tokens that are *context-sensitive*, i.e., whose selection depends not only on the code elements being translated but also on how they are utilized in the program. Tokens `char5` and `char6` are two examples as they depend on the substring length. If this length is only defined at runtime, it is impossible to know precisely which token should be assigned. This ambiguity may originate errors in the analysis, either leading to false positives or false negatives. However, since we prefer to be conservative (i.e., report false positives instead of missing vulnerabilities), in the situation where the length is undefined, ISL uses the `char6` token because it allows larger payloads to be manipulated. Something similar occurs with the `contentchk` token that depends on the verification pattern.

ISL must be able to represent PHP instructions in all steps of the two phases of the approach. When slices are extracted for analysis, all variables in ISL are set to the default token value `var`, as at that time there is still no information about the taintedness status of the variables. However, when instructions are placed in the corpus or are processed by the detection procedure, it is necessary to keep information about taintedness. In this case, tainted and untainted variables are depicted respectively by the tokens `var_vv` and `var`. The `miss` token is also used with the corpus and it serves to normalize the length of sequences (Section 7).

As a final note, ISL has no token to represent user func-

tion calls because the instructions that represent them are not relevant to the analysis, but the instructions they contain are. Hence, the latter are translated to ISL. For instance, a PHP slice, when extracted, can contain user function calls and instructions from them that handle inputs and input-dependent variables. When the slice is translated to ISL, only these instructions will be translated, discarding thus the instruction referring to the user function call. Similarly for loops and switch headers, ISL also has no tokens.

4.2 Grammar

The ISL grammar is specified by the rules in Listing 1. It allows the representation of the code elements included in the instructions into the tokens (Table 1, column 3 entries are transformed into the column 1 tokens). A slice translated into ISL consists of a set of *statement+* (line 2), each one defined by either: a rule that covers various operations like string concatenation (lines 4-8); or an conditional (line 9); or an assignment (line 19). The rules take into consideration the syntax of the functions (in column 3 of the table) in order to convey: a sensitive sink (line 11), the sanitization (line 12), the validation (line 13), the extraction and modification (lines 14-17), and the concatenation (line 18).

As we will see in Section 5, tokens will correspond to the observations of the HMM. However, while a PHP assignment sets the value of the right-hand-side expression to the left-hand side, the tokens will be processed from left to right by the model; therefore, the assignment rule in ISL follows the HMM scheme.

```
1 grammar isl {
2 slice-isl : statement+
3 statement :
4     sensitive_sink
5   | sanitization
6   | validation
7   | mod_all | mod_add | mod_sub | mod_rep
8   | concat
9   | cond statement+ cond?
10  | assignment
11 sensitive_sink : ss (param | concat)
12 sanitization : sanit_f param
13 validation : (typechk_str | typechk_num | fillchk |
       contentchk) param
14 mod_all : (join_str | erase_str | replace_str |
       split_str) param
15 mod_add : add_str param num_chars param
16 mod_sub : sub_str param num_chars start_where?
17 mod_rep : sub_str_replace param num_chars param
       start_where?
18 concat : (statement | param) (conc concat)?
19 assignment : (statement | param) attrib_var
20 param : input | var
21 attrib_var : var
22 num_chars : char5 | char6
23 start_where : begin | middle | end
24 }
```

Listing 1: Grammar rules of ISL.

Example 4. PHP instruction `$u = $_GET['user'];` is translated to *input var*. First, the `statement` rule (line 3) determines the type of the PHP instruction. As it is an assigment, the `assignment` rules are chosen (lines 10 and 19). Applying the rules to the instruction, the right-hand-side expression (`$_GET['user']`) is interpreted as being a parameter, hence the `param` rule (line 20) is used, producing the *input* token. For the left-hand-side expression (`$u`), it is used the `attrib_var` (attribution) rule that produces the *var* token (line 21). Fig. 4 depicts the parse tree resulting of the grammar application.

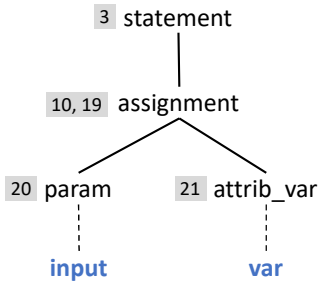


Fig. 4: Parse tree of the *input var* sequence (the numbers correspond to the applied grammar rules).

4.3 Translation to ISL

As we stated previously, a PHP slice must be translated to ISL into a slice-isl to be processed by the model. However, ISL does not maintain much information about the variables portrayed by the *var* token. This knowledge is nevertheless crucial for a more accurate vulnerability detection as variables are related to the inputs in distinct manners and their contents can suffer all sorts of modifications. Therefore, to address this issue, we resort to a data structure called *variable map* while the PHP slice is translated to ISL through the grammar rules. The map associates each occurrence of *var* in the *slice-isl* with the name of the variable that appears in the source code. This lets us track how input data propagates to different variables when the slice code elements are processed.

There is an entry in the variable map per instruction. Each entry starts with a flag, 1 or 0, indicating if the statement is an assignment or not. The rest of the entry includes one value per token of the instruction, which is either the name of the variable (without the `$`) or the – character (stands for a token that is not occupied by a variable).

Example 5. Fig. 1(a) displays a PHP code snippet that is vulnerable to SQLI and Fig. 1(b) shows the translation into ISL and the variable map (ignore the right-hand side for now). The first line is the assignment of an input to a variable, *`$u = $_POST['username'];`*. As explained above, it becomes *input var* in ISL. The variable map entry `1 – u` is initialized to `1` to denote that the instruction is an assignment to the `var` in the second position which portrays the variable `u`. The next line is an assignment of a SQL query composed by concatenating constant substrings with a variable. It is represented in ISL by *var var* and in the variable map by *1 u q*. The last line corresponds to a sensitive sink (*ss*) and two variables.

Example 6. Fig. 2 has a slightly more complex code snippet. The code contains three slices: lines {1, 2, 3, 4}, {1, 2, 3, 5} and {1, 2, 3, 6, 7}. Note that, although the code snippet has two execution paths (lines {1, 2, 3, 4, 5} and {1, 2, 3, 6, 7}), as the first one contains two sensitive sinks (`echo` function in lines 4 and 5), it in fact corresponds to two slices ({1, 2, 3, 4}, {1, 2, 3, 5}). The first two slices prevent an attack with a form of input validation, but the third is vulnerable to XSS. The corresponding ISL and variable map are shown in the

middle columns. The interesting cases are in lines 3–5, which are the `if` statement and its true branch. They are prefixed with the `cond` token and the former also ends with the same token. This `cond` termination makes a distinction between the two types of instructions (`if` statement and its true branch instructions). In addition, the sequence model will understand that variables from the former may influence those that appear in latter instructions.

5 The Sequence Model to Detect Vulnerabilities

This section presents the *sequence model* that supports our approach for vulnerability detection.

5.1 Hidden Markov Model

A Hidden Markov Model (HMM) is a statistical generative model that represents a process as a Markov chain with unobserved (hidden) states. It is a dynamic Bayesian network with nodes that stand for random variables and edges that denote probabilistic dependencies between these variables [28], [29], [30]. The variables are divided into two groups: observed variables – *observations* – and hidden variables – *states*. A state transitions to other states with some probability and emits observations (see example in Fig. 7).

A HMM is specified by the following: (1) a *vocabulary*, a set of words, symbols or tokens that make up the sequence of observations; (2) the *states*, a group of states that classify the observations of a sequence; (3) *parameters*, a set of probabilities where (i) the initial probabilities indicate the probability that a sequence of observations begins at each start-state; (ii) the transition probabilities between states; and (iii) the emission probabilities, which specify the probability of a state emitting a given observation.

In the context of NLP, sequence models are used to classify a series of observations, which correspond to the succession of words observed in a sentence. In particular, a HMM is used in PoS tagging tasks, allowing the discovery of a series of states that best explains a new sequence of observations. This is known as the *decoding problem*, which can be solved by the Viterbi algorithm [31]. This algorithm resorts to dynamic programming to pick the best hidden state sequence. Although the Viterbi algorithm employs *bigrams* to generate the *i-th* state, it takes into account all previously generated states, but this is not directly visible. In a nutshell, the algorithm iteratively obtains the probability distribution for the *i-th* state based on the probabilities computed for the *(i-1)-th* state, taking into consideration the parameters of the model.

The parameters of the HMM are *learned* by processing a corpus that is created for training. Observations and state transitions are counted, and afterwards the counts are normalized in order to obtain probability distributions; a smoothing procedure may also be applied to deal with rare events in the training data (e.g., add-one smoothing).

5.2 Vocabulary and States

As our HMM operates over the program instructions translated into ISL, the vocabulary is composed of the previously described ISL tokens. The states are selected to represent the fundamental operations that can be performed on the input data as it flows through a slice and express the type of operation a token produces. Five states were defined as displayed Table 2 (columns 1 and 2). The final state of an instruction in ISL is either vulnerable (`Taint`) or not-vulnerable (`N-Taint`). However, in order to attain an accurate detection, it is necessary to take into account the sanitization (`San`), validation (`Val`) and modification (`Chg_str`) of the user inputs and the variables that may depend on them. Therefore, these three factors are represented as intermediate states in the model. As strings are on the base of web surface vulnerabilities, these three states allow the model to determine the intermediate state when an application manipulates them.

TABLE 2: HMM states and the observations they emit.

State	Description	Emitted observations
Taint	Tainted	conc, input, var, var_vv
N-Taint	Not tainted	conc, cond, input, var, var_vv, ss
San	Sanitization	input, sanit_f, var, var_vv
Val	Validation	contentchk, fillchk, input, typechk_num, typechk_str, var, var_vv
Chg_str	Change string	add_str, char5, char6, erase_str, input, join_str, replace_str, split_str, start_where, sub_str, sub_str_replace, var, var_vv

5.3 Parameters

The parameters of the model, as we stated previously, are a set of probabilities for the initial states, the state transitions, and the observation emissions. They correspond to the *knowledge extraction* step of Fig. 3. The probabilities are computed from the corpus by counting the number of occurrences of observations and/or states. The result is 3 matrices of probabilities with dimensions of $(1 \times s)$, $(s \times s)$ and $(t \times s)$, where s and t are the number of states and tokens of the model. The matrices are calculated as follows: *Initial-state probabilities:* count how many sequences begin in each state. Then, get the probability for each state by dividing these counts by the number of sequences of the corpus. This produces a matrix with the dimension (1×5).

Example 7. To obtain the initial-state probability of the `San` state, we count how many sequences begin with the `San` state and divide by the size of the *corpus*.

Transition probabilities: count how many times in the corpus a certain state i transits to a state k (including itself). The transition probability is obtained by dividing this count by the number of pairs of states that appear in the corpus that begin with the i state. The resulting matrix has a dimension of (5×5), keeping the various probabilities for all possible transitions between the five states.

Example 8. The transition probability for the `N-Taint` state to the `Taint` state is the number of occurrences of this transition in the corpus divided by the number of pairs of states that begin in the `N-Taint` state.

Emission probabilities: count how many times a certain observation is emitted by a particular state, i.e., count how many times a certain pair $\langle$`token,state`$\rangle$ appears in the corpus. Then, calculate the emission probability by dividing this count by the total number of pairs $\langle$`token,state`$\rangle$ that occur for that specific state. The resulting matrix – called *global emission probabilities matrix* – has a dimension

of (22×5) in order to have a probability for the 22 tokens that could be emitted by each of the 5 states.

Example 9. To obtain the probability that the `Taint` state emits the `input` token ($\langle$`input,Taint`$\rangle$), first get the number of occurrences of this pair in the corpus, and next divides it by the total number of pairs of the `Taint` state.

Zero-probabilities should be avoided because the Viterbi algorithm uses multiplication to calculate the probability of moving to the next state, and therefore one needs to ensure that this multiplication is never zero. The *add-one smoothing* technique [29] can address this issue and help to compute the values of the parameters. This technique simply adds a unit to all counts, making zero-counts equal to one and the associated probability different from zero.

5.4 Detecting Vulnerabilities

The detection of vulnerabilities corresponds to the *detection phase* of our approach, as depicted in Fig. 3, that comprises steps *slice collection and translation* (see Section 4) and *search for vulnerabilities*. This section explains in detail the second step.

Given the source code of an application, the collector gathers the slices that should be examined, and then every slice is inspected separately. To commence, the instructions of the slice are translated to ISL. This means that the slice becomes a list of sequences of observations, each one corresponding to a PHP instruction. The discovery of flaws is accomplished by processing the sequences in the order of appearance, starting with the first and concluding with the last.

The HMM model is applied to each sequence of observations to find out the associated states. We resort to an extension of the Viterbi algorithm to perform this task. The algorithm employs dynamic programming to compute the most likely succession of states that explain a sequence of observations. As the algorithm finishes with a sequence, a final state comes out, either as `Taint` or `N-Taint`. This information is then propagated to the next sequence. The process is repeated for all sequences, and the final state of the last sequence defines the outcome for the slice — either as vulnerable (if it is tainted) or non-vulnerable (if it is untainted).

For the classification to be carried out effectively, it is necessary to spread faithfully the taintedness among the sequences under analysis, which means keeping information about the variables that are tainted. For this purpose, we use three artifacts that are updated as the execution evolves:

- *Tainted List* (TL): as sequences are processed, it keeps the identifiers of the variables that are perceived as tainted;
- *Conditional Tainted List* (CTL): contains the inputs (token `input`) and tainted variables (belong to TL) that have been validated (e.g., by tokens `typechk_num` and `contentchk`);
- *Sanitized List* (SL): has essentially a similar aim as CTL, except that it maintains the variables that are sanitized or modified (e.g., with functions that manipulate strings).

In our approach, the Viterbi algorithm was extended to explore the information kept in the variable map (see Section 4.3) and in these artifacts (further details in Section 7.3). Handling a sequence of observations becomes a three step procedure: (1) a preprocessing step is carried out – `beforeVit`; (2) then, the decoding step of the Viterbi algorithm is applied – `decodeVit`; (3) and lastly, a post-processing step is executed – `afterVit`. They work as follows:

`beforeVit`: the variable map is visited to get the name of the variable associated to each `var` observation. The TL and SL are checked to determine if they hold that name. In case the sequence starts with the token `cond`, the list CTL is also accessed. If a variable only belongs to TL, then the `var` observation is modified to `var_vv`, thus capturing the effect of the variable being tainted. Finally, an emission probability sub-matrix for the observations of the sequence is also retrieved from the global emission probabilities matrix;

`decodeVit`: for each observation, the Viterbi algorithm calculates the probability of each state to emit it, considering the probabilities of emission, of transition, and of the states already discovered. The multiplication of these three probabilities results in a probability called *score of state*. The state that is assigned to an observation is the one that has the highest score. The process is repeated for all observations and the state of the last observation is the one that classifies the sequence as `Taint` or `N-Taint`.
In more detail, the three probabilities are obtained as follows: emission come from the sub-matrix of emission probabilities, regarding the observations that will be processed; transition are from the matrix of transition probabilities; previous state is determined by picking up the highest score computed for the previous observation. This last probability brings to the calculation the order in which the observations appear in the sequence and the knowledge already discovered about the previous observations. However, since this knowledge does not exist for the first observation of the sequence, in this case the initial-state probabilities are used;

`afterVit`: if the sequence *is an assignment* (i.e., the last observation of the sequence is a `var` token and the entry in the variable map starts with 1), then the corresponding variable name is obtained from variable map. Next, the TL is updated: (i) inserting the variable name if the final state is `Taint` (and `var` is updated to `var_vv`); or (ii) removing it if the state is `N-Taint` and the variable is in TL; in the presence of a sanitization sequence, the variable name is also added to SL. In case the sequence *is an `if` condition* (i.e., the first and last observations are a `cond` token), then the variable map is searched for each `var` and `var_vv` observation. Next, the TL is searched to discover if it includes the name, and in that situation if validation tokens are involved with that variable (e.g., `typechk_num` token), the CTL is updated by inserting that name.

The end result of these actions is that one gets the ability to keep the relevant knowledge about the propagation of inputs through the slice, and thus determine how they can influence the sensitive sinks.

Example 10. Fig. 1(a), (b) and (c) shows an example of the

detection of a bug. It comprises from left to right: the PHP code, the representation in ISL and the variable map, and the TL after observations are classified. In line 1, the Viterbi algorithm is applied and as a result the *var* observation is tainted because by default an *input* observation is so; the model classifies it correctly and variable *u* is inserted in TL and *var* updated to *var_vv*. In line 2, the first *var* observation is updated to *var_vv* because it corresponds to variable *u* that belongs to TL, and then the Viterbi algorithm is applied; the *var_vv var* sequence is classified by the model and the final state is *Taint*; therefore, variable *q* is inserted in TL. The process is repeated for the next line, allowing the discovery of the flaw. Fig. 1(d) presents the decoding of the slice while the processing progresses. Here, it is possible to see the places where *var* is replaced by *var_vv*, with the relevant variable name as suffix. In addition, the states of each observation are also added. By following the generated states, one can understand the effects of the code execution (without actually running it), which variables are tainted, and why the code is vulnerable. The state of the last observation indicates the final classification — a vulnerability.

Example 11. Fig. 2 has the PHP code for the three slices composed of lines {1, 2, 3, 4}, {1, 2, 3, 5} and {1, 2, 3, 6, 7} respectively. After processing the first two slices, TL = {u, a} and CTL = {u, a} as variable *u* is the parameter of the *contentchk* token and variable *a* is the parameter of the *typechk_int* token. The final state for both slices is *N-Taint* because variables *u* and *a* are included in CTL and lines 4 and 5 belong to the true branch of the *if* statement; so affected by CTL. In the third slice, TL = {u, a} and CTL = { } since there is no validation and the final state is *Taint*.

6 Learning and Corpus Assessment

This section explains the main activities related to the corpus, more specifically, the learning phase and the validation of the model. Learning encompasses a number of activities (see Figure 3) that culminate with the computation of the parameters of the HMM model performed by the process explained in Section 5.3. A corpus was also created to train the model and an assessment was made to verify its consistency. This corpus can be extended in the future with additional annotated sequences, allowing the model to evolve its knowledge and detection capabilities. In addition, the section presents the knowledge graph of the model, which is derived from the corpus and depicts the knowledge it comprises, and the validation of the model based on the current corpus.

6.1 Methodology to Build the Corpus

The corpus plays an important role as it incorporates the knowledge that will be learned by the model, namely which instructions may lead to a flaw. In our case, the *corpus* is a group of instructions (belonging to slices) converted to ISL, where tokens are tagged with information related to taint propagation. The model sees the tokens of an instruction in ISL as a sequence of observations. The tags correspond to the states of the model. Therefore, an alternative way to look at the corpus is as a group of sequences of observations annotated with states.

The process of creating the corpus involves four steps: (1) *collection* of a group of instructions that are vulnerable and not-vulnerable, which are placed in a bag; (2) *representation* of each instruction in the bag in ISL; (3) *annotation* of the tokens of every instruction (e.g., as tainted or sanitized), i.e., associate a state to each observation of the sequence; and (4) *removal* of duplicated entries in the bag. In the end, an instruction becomes a list of pairs of ⟨token, state⟩.

In the first step, it is necessary to get representative instructions of all classes of bugs that one wants to catch, various forms of validations (for numbers and strings), diverse forms of manipulating (changing) strings, and different combinations of code elements. To achieve this in practice, we can gather individual instructions or/and we can select a large number of slices captured from open source training applications. Therefore, both the collection and representation can be performed in an automatic manner (with the slice collector and ISL translator modules), but the annotation of the tokens is done manually (as in all supervised machine learning approaches).

As mentioned in Section 4, the token var_vv is not produced when slices are translated into ISL, but used in the corpus to represent variables with state *Taint* (tainted variables). In fact, during translation into ISL, variables are not known to be tainted or not, so they are represented by the var token. In the corpus, if the state of the variable is annotated as *Taint*, the variable is portrayed by var_vv, forming the pair ⟨var_vv, Taint⟩.

The state of the last observation of a sequence corresponds to a final state, and therefore it can only be *Taint* (vulnerable) or *N-Taint* (not-vulnerable). If this state is tainted then it means that a malicious input is able to propagate and potentially compromise the execution. Therefore, in this case, the instruction is perceived as vulnerable. Otherwise, the instruction is deemed correct (non-vulnerable).

Example 12. Instruction `$v = $_POST['paramater']` becomes *input var* in ISL, and is annotated as ⟨input, Taint⟩ ⟨var_vv, Taint⟩. Both states are *Taint* (compromised) because the *input* can be the source of malicious data, and therefore is always *Taint*, and then the taint propagates to the variable which will be portrayed by *var_vv*.

Example 13. The `$v = htlmentities($_GET['user'])` instruction is translated to *sanit_f input var* and is placed in the corpus as the succession of pairs ⟨sanit_f, San⟩ ⟨input, San⟩ ⟨var, N-Taint⟩. The first two tokens are annotated with the *San* state because function htlmentities sanitizes its parameter; the last token is labeled with the *N-Taint* state, meaning that the ultimate state of the sequence is not tainted.

6.2 Corpus Construction and Assessment

The model needs to classify correctly the sequences of observations or, in our case, needs to detect vulnerabilities without mistakes. Since the model is configured with the corpus, its quality depends strongly on incorporating valid and enough information in the corpus. Therefore, to build the corpus, we resorted to a method inspired in Jurafsky and Martin [29]. The method operates iteratively in three phases to gradually assess and improve the resulting model.

The *evaluation phase* verifies if the model outputs correctly a sequence of observations O for a given sequence of states S. The *decoding phase* determines if the model outputs a S that explains correctly a given O. This phase corresponds to the objective of our approach. The last phase, *re-learning*, verifies if the model needs adjustments to its *parameters* in order to maximize the results of the previous phases. It consists of enhancing the model by adding more sequences to the corpus and running another cycle of the method.

After applying our methodology (described in the previous section) and the method, the resulting corpus had 510 slices, where 414 are vulnerable and 96 are non-vulnerable. These slices were extracted from various open source PHP applications[2] and had flaws from the fiftheen classes presented in Section 2.

```
1  $var = $_POST['parameter']
2  $var = $_GET['parameter']
3  $var = htmlentities($_POST['parameter'])
4  $var = mysqli_real_escape_string($con, $_GET['parameter'])
5  $var = htmlentities($var)
6  $var = "SELECT field FROM table WHERE field = $var"
7  $var = mysqli_query($con, $var)
8  $var = mysql_query($var)
9  echo $var
10 include($var)
11 $var = (isset($var)) ? $var : ''
12 if (isset($var) && $var > number)
13 if (is_string($var) && preg_match('pattern', $var))
14 if (isset($var) && preg_match('pattern', $var) &&
       is_int($var))
```

Listing 2: Creating the corpus: collection step.

Listing 2 displays an excerpt of the result of the first step. It contains fourteen PHP instructions collected from vulnerable and non-vulnerable slices. The representation of the instructions into ISL is illustrated in Listing 3 (second step). It is possible to observe that some instructions may have more than one representation, depending on the extracted slice being vulnerable or not. For example, the instruction at the fifth position in Listing 3 appears as two series (the two lines immediately below of it) corresponding to the sanitization of an untainted and a tainted variable, respectively. In the listing, the difference between the `var` and `var_vv` tokens is visible. Listing 4 has the final corpus that is produced after applying the last two steps. Each sequence of observations is annotated with the state as explained in the previous section. The duplicated sequences are eliminated as several PHP instructions can result in the same sequence. For example, PHP instructions in lines 1 and 2 (Listing 2) become the same sequence (line 1 of Listing 4).

The probability matrices (model's parameters) that were computed based on this corpus are shown in Fig. 5.

To perform a preliminary assessment the resulting model, we applied a *10-fold cross validation* [32]. This form of validation involves dividing the training data (the corpus of 510 slices) in ten folds. Then, the model is trained with a sub-corpus of nine of the folds and tested with the tenth fold. This process is repeated ten times to evaluate every fold with a model trained with the rest. The metrics that

2. bayar, bayaran, ButterFly, CurrentCost, DVWA 1.0.7, emoncms, glfusion-1.3.0, hotelmis, Measureit 1.14, Mfm-0.13, mongodb-master, Multilidae 2.3.5, openkb.0.0.2, Participants-database-1.5.4.8, phpbttrkplus-2.2, SAMATE, superlinks, vicnum15, ZiPEC 0.32, Wordpress 3.9.1.

```
1  $var = $_POST['parameter']
   input var_vv
2  $var = $_GET['parameter']
   input var_vv
3  $var = htmlentities($_POST['parameter'])
   sanit_f input var
4  $var = mysqli_real_escape_string($con, $_GET['parameter'])
   sanit_f input var
5  $var = htmlentities($var)
   sanit_f var var
   sanit_f var_vv var
6  $var = "SELECT field FROM table WHERE field = $var"
   var var
   var_vv var_vv
7  $var = mysqli_query($con, $var)
   ss var var
   ss var_vv var_vv
8  $var = mysql_query($var)
   ss var var
   ss var_vv var_vv
9  echo $var
   ss var_vv
   ss var
10 include($var)
   ss var_vv
   ss var
11 $var = (isset($var)) ? $var : ''
   var var
   var_vv var_vv
12 if (isset($var) && $var > number)
   cond fillchk var_vv cond
   cond fillchk var cond
13 if (is_string($var) && preg_match('pattern', $var))
   cond typechk_str var_vv contentchk var_vv cond
   cond typechk_str var_vv contentchk var cond
   cond typechk_str var contentchk var_vv cond
   cond typechk_str var contentchk var cond
14 if (isset($var) && preg_match('pattern', $var) &&
       is_int($var))
   cond typechk_str var_vv contentchk var_vv typechk_int
       var_vv cond
   cond typechk_str var_vv contentchk var_vv typechk_int var
       cond
   cond typechk_str var_vv contentchk var typechk_int var_vv
       cond
   cond typechk_str var_vv contentchk var typechk_int var
       cond
   cond typechk_str var contentchk var_vv typechk_int var_vv
       cond
   cond typechk_str var contentchk var_vv typechk_int var
       cond
   cond typechk_str var contentchk var typechk_int var_vv
       cond
   cond typechk_str var contentchk var typechk_int var cond
```

Listing 3: Creating the corpus: representation step.

are used in the evaluation are: *Accuracy (acc)* measures the ratio of well-classified slices (as vulnerable and non-vulnerable) over the total number of slices (*N*), whereas *precision (pr)* assesses the fraction of classified bugs that are really vulnerabilities. The objective is high accuracy and precision or, similarly, to minimize the *false positive rate (fpr)* which is the rate of generating false alarms for slices that are correct, and to minimize the *false negative rate (fnr)* which is the rate of missing certain vulnerable slices. Given that *TP* and *TN* are the well-classified instances as vulnerable and non-vulnerable, while *FP* is the false alarms and *FN* is the missing flaws, the metrics are computed with: $acc = (TP + TN)/N$; $pr = TP/(TP + FP)$; $fpr = FP/(FP + TN)$; and $fnr = FN/(FN + TP)$.

Table 3 presents a confusion matrix for the alerts produced in the first two phases of the method. For example, the first row says that it issued 419 alerts in the evaluation

```
1  <input,Taint> <var_vv,Taint>
2  <sanit_f,San> <input,San> <var,N-Taint>
3  <sanit_f,San> <var,San> <var,N-Taint>
4  <sanit_f,San> <var_vv,San> <var,N-Taint>
5  <var,N-Taint> <var,N-Taint>
6  <var_vv,Taint> <var_vv,Taint>
7  <ss,N-Taint> <var,N-Taint> <var,N-Taint>
8  <ss,N-Taint> <var_vv,Taint> <var_vv,Taint>
9  <ss,N-Taint> <var_vv,Taint>
10 <ss,N-Taint> <var,N-Taint>
11 <cond,N-Taint> <fillchk,Val> <var_vv,Val> <cond,N-Taint>
12 <cond,N-Taint> <fillchk,Val> <var,Val> <cond,N-Taint>
13 <cond,N-Taint> <typechk_str,Val> <var_vv,Val>
       <contentchk,Val> <var_vv,Val> <cond,N-Taint>
14 <cond,N-Taint> <typechk_str,Val> <var_vv,Val>
       <contentchk,Val> <var,Val> <cond,N-Taint>
15 <cond,N-Taint> <typechk_str,Val> <var,Val>
       <contentchk,Val> <var_vv,Val> <cond,N-Taint>
16 <cond,N-Taint> <typechk_str,Val> <var,Val>
       <contentchk,Val> <var,Val> <cond,N-Taint>
17 <cond,N-Taint> <typechk_str,Val> <var_vv,Val>
       <contentchk,Val> <var_vv,Val> <typechk_int,Val>
       <var_vv,Val> <cond,N-Taint>
18 <cond,N-Taint> <typechk_str,Val> <var_vv,Val>
       <contentchk,Val> <var_vv,Val> <typechk_int,Val>
       <var,Val> <cond,N-Taint>
19 <cond,N-Taint> <typechk_str,Val> <var_vv,Val>
       <contentchk,Val> <var,Val> <typechk_int,Val>
       <var_vv,Val> <cond,N-Taint>
20 <cond,N-Taint> <typechk_str,Val> <var_vv,Val>
       <contentchk,Val> <var,Val> <typechk_int,Val>
       <var,Val> <cond,N-Taint>
21 <cond,N-Taint> <typechk_str,Val> <var,Val>
       <contentchk,Val> <var_vv,Val> <typechk_int,Val>
       <var_vv,Val> <cond,N-Taint>
22 <cond,N-Taint> <typechk_str,Val> <var,Val>
       <contentchk,Val> <var_vv,Val> <typechk_int,Val>
       <var,Val> <cond,N-Taint>
23 <cond,N-Taint> <typechk_str,Val> <var,Val>
       <contentchk,Val> <var,Val> <typechk_int,Val>
       <var_vv,Val> <cond,N-Taint>
24 <cond,N-Taint> <typechk_str,Val> <var,Val>
       <contentchk,Val> <var,Val> <typechk_int,Val>
       <var,Val> <cond,N-Taint>
```

Listing 4: Creating the corpus: annotation and removal steps.

phase but that 14 of them were mistakes (columns 2 and 3). In the evaluation phase, the precision and accuracy were very high, around 0.97 and 0.95, and the rates were small (*fpr* is 0.15 and *fnr* is 0.02). In the decoding phase, the results were even more positive, with a precision and accuracy approximately of 0.96 and rates of 0.17 and 0.005 (almost null *fnr* rate). Since there is a trade-off between the two

$$\begin{bmatrix} 0.062 & 0.323 & 0.062 & 0.015 & 0.538 \end{bmatrix}$$

(a) initial-state probabilities.

$$\begin{bmatrix} 0.619 & 0.099 & 0.174 & 0.059 & 0.333 \\ 0.115 & 0.641 & 0.304 & 0.353 & 0.373 \\ 0.027 & 0.028 & 0.435 & 0.059 & 0.020 \\ 0.009 & 0.033 & 0.043 & 0.471 & 0.020 \\ 0.009 & 0.006 & 0.043 & 0.059 & 0.255 \end{bmatrix}$$

(b) transition probabilities.

$$\begin{bmatrix} 0.085 & 0.015 & 0.103 & 0.030 & 0.075 \\ 0.326 & 0.010 & 0.154 & 0.030 & 0.075 \\ 0.008 & 0.005 & 0.256 & 0.030 & 0.015 \\ 0.008 & 0.051 & 0.026 & 0.030 & 0.015 \\ 0.380 & 0.406 & 0.026 & 0.030 & 0.015 \\ 0.008 & 0.005 & 0.026 & 0.091 & 0.015 \\ 0.008 & 0.005 & 0.026 & 0.091 & 0.015 \\ 0.008 & 0.005 & 0.026 & 0.061 & 0.015 \\ 0.008 & 0.076 & 0.026 & 0.030 & 0.015 \\ 0.008 & 0.005 & 0.026 & 0.030 & 0.060 \\ 0.008 & 0.005 & 0.026 & 0.030 & 0.060 \\ 0.008 & 0.005 & 0.026 & 0.030 & 0.060 \\ 0.008 & 0.005 & 0.026 & 0.030 & 0.060 \\ 0.008 & 0.005 & 0.026 & 0.030 & 0.134 \\ 0.008 & 0.005 & 0.026 & 0.030 & 0.104 \\ 0.008 & 0.005 & 0.026 & 0.030 & 0.134 \\ 0.008 & 0.061 & 0.026 & 0.030 & 0.015 \\ 0.008 & 0.005 & 0.026 & 0.061 & 0.015 \\ 0.070 & 0.020 & 0.026 & 0.030 & 0.015 \\ 0.016 & 0.294 & 0.051 & 0.212 & 0.075 \\ 0.008 & 0.005 & 0.026 & 0.030 & 0.015 \\ 0.270 & 0.208 & 0.056 & 0.061 & 0.015 \end{bmatrix}$$

(c) global emission probabilities.

Fig. 5: Parameters of our HMM model extracted from the corpus. Columns correspond to the 5 states (in the same order of column 1 of Table 2). The lines of matrix (c) are the tokens (in the same order of column 1 of Table 1).

TABLE 3: Confusion matrix. *Observed* is the reality, where there are 414 slices with flaws and 96 correct. *Predicted* is the output of DEKANT with our corpus (419 vuln., 91 not vuln. in the evaluation phase; 428 vuln., 82 not vuln. in the decoding phase).

		Observed			
		Evaluation phase		Decoding phase	
		Vul	N-Vul	Vul	N-Vul
Predicted	Vul	405	14	412	16
	N-Vul	9	82	2	80

rates, it is interesting to notice that there is a very low *fnr* that leads to a few FPs (wrong alerts). This is advantageous because the alternative would mean missing vulnerabilities. So, these results provide promising evidence of the excellent performance of the resulting model, something that we will be check more thoroughly in the experimental evaluation section (see Section 8).

6.3 Knowledge Graph of the Model

The knowledge graph can be derived from the knowledge that the corpus contains, thus illustrating a graphical representation of the knowledge it contains. In other words, the graph expresses what the model learns and, therefore, how it can detect vulnerabilities. Fig. 6 displays the graph, where the nodes constitute the states and the edges the transitions between them. The dashed squares next to the nodes hold the observations that can be emitted in each state (as the second column of Table 2 shows).

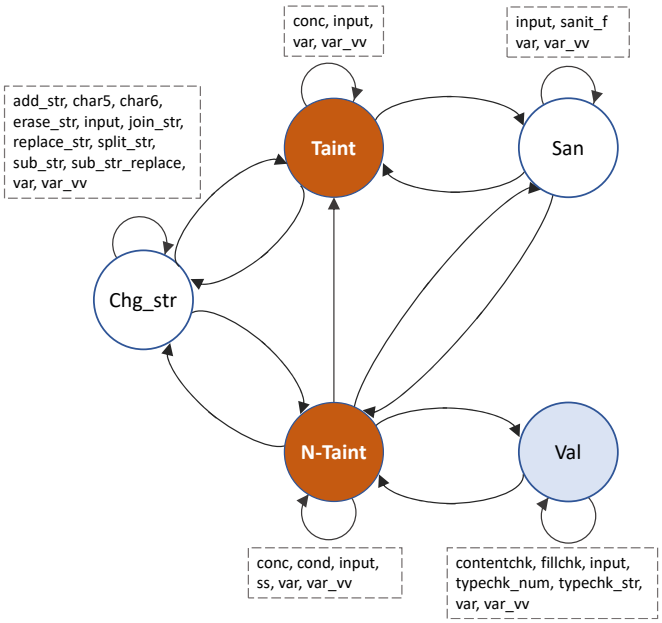


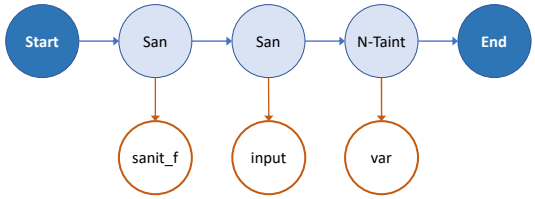
Fig. 6: Knowledge graph for the proposed HMM.

In the corpus, an ISL instruction corresponds to a sequence of observations with its tokens tagged with states. The sequence can start in any state except `Val`, as there is no sequence that begins with a validation function (e.g., there is no PHP instruction that initiates with the `is_string` function – the `typechk_str` token). However, it can reach the `Val` state for example due to conditionals that check the input data. When the processing of the sequence completes,

the model is always either in the `Taint` or `N-Taint` states. Therefore, the final state determines the overall classification of the statement, i.e., if the instruction is vulnerable or not. We recall that every sequence in the corpus ends in one of these two states (see Listing 4). Moreover, the model uses this information to propagate these states between sequential sequences in order to improve the precision in the detection of vulnerabilities (see Section 5.4).

The knowledge graph of the model can be seen as a deterministic finite automata (DFA), also known as a deterministic finite-state machine (DFSM) [33]. A DFA consists of a five-tuple $\langle S, V, t, F, s_0 \rangle$ like our HMM. S and V are the states and vocabulary already presented in Section 5.2. t is the transition function (computed by our modified Viterbi algorithm) that determines which is the next state that the DFA will jump to by taking as input an observation and the current state. F is the set of accepting states, either `Taint` or `N-Taint`, forcing every acceptable sequence to end in one of these states. s_0 is the initial (start) state that each sequence must begin.

Our HMM has similar behavior to a DFA, i.e, it receives observations as input and processes them by using a transition function to determine the next state. For each observation, the DFA jumps deterministically from one state to another state by following the transition edge. This means that in the corpus each sequence of observations and its sequence of states is unique (the *removal* step of the corpus creation ensures the elimination of duplicates (see Section 6.1)). In addition, as our model learns the knowledge contained in the graph, it will output deterministically the single sequence of states for a sequence of observations.



PHP instruction: `$p = mysqli_real_escape_string($con, $_GET['user'])`
ISL instruction: `sanit_f input var`
Sequence: ⟨`sanit_f,San`⟩ ⟨`input,San`⟩ ⟨`var,N-Taint`⟩

Fig. 7: Graph instantiation for an example sequence of the corpus.

Example 14. Fig. 7 shows an instantiation of a sequence of the corpus where the knowledge graph has been applied. The sequence starts in the `San` state and emits the `sanit_f` observation; next it remains in the same state and emits the `input` observation; finally, it transits to `N-Taint` state, emitting the `var` observation (untainted variable).

7 Implementation

Our approach is implemented in the DEKANT tool. This section explains the modules that compose the tool and the algorithms we consider more important, in particular the *slice translator* module and the extensions we made to the Viterbi algorithm.

7.1 Modules of DEKANT

DEKANT is programmed in Java and its architecture is divided in four major modules, which are explained below in more detail:

(1) *Knowledge extractor:* operates separately from the other modules and is executed when the corpus is built or later modified. It runs in three steps: (i) the sequences composed of series of annotated tokens are loaded from a plain text file. Each sequence is separated in pairs ⟨`token,state`⟩ and the elements of each pair are inserted in the matrices called *observations* and *states*. Since sequences normally have different numbers of pairs, it becomes necessary to *normalize the length of all sequences* in the corpus. This is accomplished by first determining the length of the largest sequence, and then by padding shorter sequences with the `miss` token together with the state of the last observation (i.e., with pairs ⟨`miss,Taint`⟩ or ⟨`miss,N-Taint`⟩) to ensure that all sequences have the same length; (ii) then, the various probabilities of the model are computed as explained in Section 5.3; (iii) lastly, all relevant information about the model is saved in a plain text file to be loaded by the vulnerability detector module.

(2) *Slice collector:* uses a lexer and a parser to process PHP code (based on ANTLR[3]). It searches the application files for places where inputs arrive from the user and then tracks the data flows until either a security-critical instruction is reached or the program exits. Slices that have both an entry point and a sensitive sink are passed to the translator (and the others are discarded). The information about which entry points and sensitive sinks should be considered is provided in a configuration file.

(3) *Slice translator:* The module reads configuration files describing the classes of tokens, e.g., containing the PHP functions that are represented by tokens. Some of them are transversal to any class of vulnerability, whereas others are specific to a particular bug. For example, the `input` file contains `$_GET` and `$_POST` global arrays and the `ss_xss` file has the security-sensitive functions associated with XSS (e.g., `echo`). The module first parses the slice and next verifies which tokens should be assigned to each PHP instruction, following the ISL grammar rules. Simultaneously, it also generates the variable map.

(4) *Vulnerability detector:* works in three steps to find the bugs. (i) the probabilities are loaded from a file and the model is setup internally; (ii) the slice translated into the intermediate language is processed using the modified Viterbi algorithm. Sometimes, it occurs that a sequence has more observations than the largest sequence that was seen in the corpus. When this happens, it is necessary to divide the sequence in sub-sequences with at most the maximum corpus sequence length. Then, each one is classified separately, but the algorithm is careful to ensure that the initial probability of the following sub-sequence is equal to the probability resulting from the previous sub-sequence; (iii) lastly, the various probabilities are estimated for a sequence of observations to be explained by particular sequences of states, and the most probable is chosen. An alert message is issued if a vulnerability is found.

3. https://www.antlr.org/

7.2 Slice translator

Listing 5 shows the algorithm of the slice translator module.

```
1  /* >>> Data structures and variables <<<
2  ** ISL - ISL tokens list
3  ** CF - configuration files with PHP code elements
4  ** VM - variable map
5  ** AST - abstract syntax tree of the PHP slice
6  ** code_elem - a PHP code element
7  ** first - variable to indicate the 1st code element of
       an instruction
8  ** VM_inst - variable to represent the VM of an inst.
9  ** isl_inst - variable that represents the isl instruction
10 ** assign - variable for controlling assignment instructs.
11 ** if_st - variable for controlling ifs instructions
12 */
13
14 load configuration files to CF
15 AST = get ast from PHP-slice
16
17 for each branch in AST do
18    first = 1
19    assign = ""
20    isl_inst = ""
21    VM_inst = ""
22    if_st = 0
23    for each code_elem in branch do
24       if first = 1 then
25          first = 0
26          if code_elem starts with $ then
27             get var_name of code_elem
28             assign = var_name
29             VM_inst = 1
30          else
31             if code_elem starts with if then
32                if_st = 1
33                isl_inst = cond
34                VM_inst .= 0 -
35             else
36                VM_inst = 0
37             end_if
38          end_if
39       else
40          if code_elem starts with $ then
41             get var_name of code_elem
42             isl_inst .= var
43             VM_inst .= var_name
44          else
45             if code_elem starts with $_ then
46                isl_inst .= input
47                VM_inst .= -
48             else
49                isl_token = ""
50                get isl_token using (code_elem, ISL, CF)
51                if isl_token != "" then
52                   VM_inst .= -
53                   isl_inst .= isl_token
54                end_if
55             end_if
56          end_if
57       end_if
58    for_end
59
60    if assign != "" then
61       VM_inst .= assign
62       isl_inst .= var
63    end_if
64    if if_st = 1 then
65       VM_inst .= -
66       isl_inst .= cond
67    end_if
68
69    insert VM_inst in VM
70    apply grammar to isl_inst
71 for_end
```

Listing 5: Slice translator algorithm.

After the configuration files (`CF`) are loaded and the PHP slice is parsed, the the abstract syntax tree (`AST`) is generated (lines 14 and 15). Next, each `AST` branch that represents a slice instruction is analyzed. For the first code element of the branch, it is checked if it is a variable, meaning that we are in the presence of an assignment instruction, or a `if` statement (lines 26 to 39). Two other checks are performed for every PHP code element to determine if they are a variable or an input (lines 40 to 48). Besides these checks, the code elements are analyzed against the `CF` in order to obtain the ISL tokens (lines 49 to 53). In all cases, the variable map (`VM_inst`) is filled and the `isl-inst` composed accordingly. After the `AST` branch is processed, the `VM_int` and the `isl-inst` keep the assignment and if statement cases information and the `VM` is updated (lines 60 to 69). Finally, the `isl-inst` is analyzed to find out that it follows the grammar rules and is also translated (line 70).

7.3 Extensions to the Viterbi algorithm

We extended the Viterbi algorithm with the two procedures of Section 5.4 (*`beforeVit`* and *`afterVit`*) to track the propagation of inputs while processing a slice and to explore the data structures that keep relevant knowledge about variables (e.g., the three artifacts TL, CTL, SL and the variable map).

Listing 6 presents the *`beforeVit`* preprocessing procedure that is run before the Viterbi algorithm. `beforeVit` does a few tests to manipulate some flags and change the data structures. For each observation (`obs`) in the sequence (`inst_slice_isl`) there are checks to find out: (i) the presence of sanitization (`sanit_f`) or a `cond` tokens. For the latter case, the `obs` position is verified in the sequence to discover if the instruction is an `if` statement, an instruction inside of a conditional statement, or an `else` statement (lines 19 to 28); (ii) an `if` statement is searched for validation functions and if their parameters are a variable or an input (i.e., `var` or `input`). In such case, `var` or `input` are inserted in CTL (lines 30 to 46); (iii) an instruction inside an `if` statement is checked if the `var` and `input` tokens belong to CTL and/or SL. VM (variable map) is accessed to get the name of variable associated to `var` token. If the token `input` belongs to the SL or CTL lists, it is replaced by the `var` token because it has to loose its taintedness (we recall that by default this token is tainted and the `var` token is untainted, so this replacement is required) (lines 48 to 59); (iv) in presence of another instruction and if the observation is a `var` token, i.e., the `inst_slice_isl` is out of the validation scope, the name of variable is taken from VM and checked if it belongs to TL but not in SL. In such a case, the variable is tainted, and the observation is replaced by the `var_vv` token (lines 61 to 69). For all four verifications, the emission probability of the observation in analysis is retrieved from the global emission probabilities matrix (`GEP`), then it is inserted in the emission probabilities matrix (`EP`) of the `inst_slice_isl` (line 70).

Afterwards, the traditional Viterbi algorithm is executed (*`decodeVit`* step as explained in Section 5.4) and then the post-processing `afterVit` procedure runs.

Listing 7 shows `afterVit`. It takes as inputs the final state of the `inst_slice_isl` (`state`) and the assignment value (`value`) of the instruction stored in VM (lines 11-12), and then makes the following checks: (i) if the instruction is an assignment, then the last observation of the sequence is a variable (`var` or `var_vv` token), so the name of the variable (`var_name`) is taken from VM (lines 14-15); (ii) if the instruction is classified as *`Taint`*, then the assignment

variable is tainted, so the var_name is put in TL. If this var_name already belongs to SL, it is removed from this list (lines 16-20); (iii) if the instruction is classified as *N-Taint*, then the assignment variable is untainted, and therefore it can be removed from TL. Additionally, it is verified if the instruction is a result of a sanitization operation, and in such case the name is inserted in SL (lines 22-26).

```
1  /* >>> Data structures and variables <<<
2  ** VM - variable map
3  ** TL - tainted list
4  ** CTL - conditional tainted list
5  ** SL - sanitized list
6  ** obs_index - index of obs in the instruction_slice_isl
7  ** var_name - variable name of the obs from inst_slice_isl
8  ** condition - variable for controlling if stattements
9  ** val - variable for controlling validation functions
10 ** san - variable for controlling sanitization functions
11 ** EP - emission probability matrix of
        instruction_slice_isl
12 ** GEP - global emission probabilities matrix
13 ** obs_ep - emission probability of the obs in analysis
14 */
15
16 val = 0
17 san = 0
18 for each obs in inst_slice_isl do
19    if obs = sanit_f then san = 1 end_if
20
21    if obs = cond then
22       if obs_index = 1 then
23          if size(inst_slice_isl) = 1 then condition = 0
                else condition = 1 end_if
24       else
25          condition = 2
26       end_if
27       get obs_ep from GEP
28    end_if
29
30    if condition = 1 and obs_index <> 1 then
31       if obs in [typechk_num, contentchk] then
32          val = 1
33       end_if
34
35       if obs = var and val = 1 then
36          get var_name of obs from VM
37          insert var_name in CTL
38          val = 0
39       end_if
40
41       if obs = input and val = 1 then
42          insert input in CTL
43          val = 0
44       end_if
45       get obs_ep from GEP
46    end_if
47
48    if condition = 2 then
49       if obs = var then
50          get var_name of obs from VM
51          if var_name in [CTL, SL] then
52             get obs_ep from GEP
53          end_if
54       end_if
55       if obs = input and input in [CTL, SL] then
56          obs = var
57          get obs_ep from GEP
58       end_if
59    end_if
60
61    if condition = 0 then
62       if obs = var then
63          get var_name of obs from VM
64          if var_name in TL and not in SL then
65             obs = var_vv
66          end_if
67       end_if
68       get obs_ep from GEP
69    end_if
70    insert obs_ep in EP
71 end_do
```

Listing 6: *beforeVit* extension to the Viterbi algorithm.

```
1  /* >>> Data structures and variables <<<
2  ** VM - variable map
3  ** TL - tainted list
4  ** SL - sanitized list
5  ** state - state of the last obs from inst_slice_isl
6  ** value - assignament value of inst_slice_isl on VM
7  ** var_name - variable name of the obs from inst_slice_isl
8  ** san - variable for controlling sanitization functions
9  */
10
11 get state of inst_slice_isl
12 get value from VM
13
14 if value = 1 then
15    get var_name of the last_obs from VM
16    if state = taint then
17       insert var_name in TL
18       if var_name in SL then
19          remove var_name from SL
20       end_if
21    else
22       if san = 1 then
23          insert var_name in SL
24          san = 0
25          if var_name in TL then
26             remove var_name from TL
27          end_if
28       end_if
29    end_if
30 end_if
```

Listing 7: *afterVit* extension to the Viterbi algorithm.

8 Experimental Evaluation

Our experimental evaluation addresses the following questions about DEKANT: (1) Is the tool able to discover novel vulnerabilities? (Section 8.1); (2) Can it classify correctly various classes of vulnerabilities? (Section 8.1); (3) Is DEKANT more accurate and precise than tools that search for vulnerabilities in plugins (Section 8.2); (4) Is DEKANT more accurate and precise than tools that do data mining using standard classifiers (Section 8.3); (5) Is it more effective than tools that resort to code property graphs (Section 8.4)?

8.1 Open Source Software Evaluation

This section assesses the ability of DEKANT to classify different vulnerabilities by analyzing 25 WordPress plugins [34] and 21 packages of real web applications. All of these are written in the PHP language. The plugins and 8 web applications are used to determine if the tool is useful for the discovery of new (zero-day) vulnerabilities. The remaining 13 web applications serve as a ground truth for the evaluation, since they have known vulnerabilities found by [27]. In every test, DEKANT resorted to the corpus explained in the previous section (however, *none* of the programs utilized in the evaluation was employed to build the corpus). All outputs of the tool were confirmed by us manually to pinpoint valid detections and mistakes.

8.1.1 Zero-day Vulnerabilities in Plugins

WordPress is the most adopted Content Management System (CMS) worldwide, and therefore its plugins are interesting targets for our study. WordPress offers a set of functions to sanitize and validate the data types (e.g., *sanitize_text_field*, *esc_html*), to read entry points (e.g., *get_post*), and to handle SQL commands (*$wpdb* class), which are invoked by some of the plugins. Therefore, we configured DEKANT with information about these functions, mapping

them to the ISL tokens. Recall that ISL abstracts the PHP instructions, enabling certain behaviors to be captured like sanitization.

We selected a diverse set of plugins based on two criteria, the development team and the number of downloads. For the former, we chose 15 plugins built by companies and the other 10 by individual developers. For the second, we picked 9 with less than 20,000 downloads, 5 with less 100,000 downloads, and the other 11 with more than 100,000 downloads (391,500 in average), some of which having more than 1M of downloads. Note that plugins with less downloads were not always those created by individual developers. The plugins were chosen to have also diverse characteristics with regard to the number of files and lines of code (LoC). Although plugins are often believed to be small, in a few cases they had more than 200 files and 100,000 LoC (see Table 7). We chose two versions for 6 of them, one with vulnerabilities discovered and registered in the past (marked with X in the third column of Table 4), for which a new and fixed version was generated, and the latest one (which is not the fixed one). In addition, 2 plugins with 2 versions (one old and the latest one) also were selected (*Gantry Framework* and *WP Shop*), but without any information about vulnerabilities already discovered. For these two, the third column of Table 4 is empty.

DEKANT extracted 455 slices from the plugins that begin at an entry point and end at a sensitive sink. Next, it translated them into ISL and executed the detection procedure. In total 295 slices are reported as potentially being vulnerable, but 5 of them are actually invalid alarms (i.e., *false positives (FP)*). There are 205 new vulnerabilities that no one had previously found, and 85 bugs that had already been published by us [26] and other researchers (marked with X in Table 4, third column). The remaining slices, a group of 160, are correctly perceived as not vulnerable. The flaws belong to six classes of vulnerability, ranging from SQLI to CS (columns 5-8).

TABLE 4: Vulnerable slices in plugins found by DEKANT.

WordPress Plugin	Ver	R	Slices	Real Vulnerabilities				FP
				SQLI	XSS	Files	oth	
Appointment Booking Calendar	1.1.7	x	15	3	4			
Appointment Booking Calendar	1.3.39		9	3				
Authorizer	2.3.6		2		2			
Contact formgenerator	2.0.1	x	14	11				
Contact formgenerator	2.1.82		18	10				
Easy2map	1.2.9	x	13		1	2		
Ecwid Shopping Cart	3.4.6	x	1		1			
Ecwid Shopping Cart	6.9.6		1		1			
Gantry Framework	4.1.6		3		3			
Gantry Framework	4.1.21		3		3			
Google Maps Travel Route	1.3.1		10	1	2			1
Payment Form for Paypal pro	1.0.1	x	19		2			
Payment Form for PayPal Pro	1.1.64		11		2			
ResAds	1.0.1	x	17		17			
ResAds	2.0.3		13		13			
Simple support ticket system	1.2	x	37	18				
Simple support ticket system	1.3.6		42	13	4			
The Cart Press eCommerce Shopp.	1.4.7		25	8	17			
WP Easy Cart - eCommerce Shopp.	3.2.3		78	13	6	29	12	
WP Marketplace	2.4.1	x	45	2	24			3
WC Marketplace	3.4.11		29		19			1
WP Shop	3.5.3		22	7	10			
WP Shop	3.9.6		22	4	17			
WP Simple e-Commerce Shopp. Cart	2.2.5		3	2	1			
WP Web Scraper	3.5		3		3			
Total		8	455	95	152	31	12	5

Files: DT & RFI, LFI vulnerabilities others: SCD, HI, CS vulnerabilities
R: registered vulnerable plugin

The zero-day vulnerabilities appear in 17 plugins: 10 developed by companies and 7 by individual programmers; and 9 having more than 100,000 downloads. The most vulnerable plugin is the one that has more files, while the plugins appearing in the next places are smaller, and the largest plugin in terms of LoC has less than 4 identified bugs. These results reveal that, independently of the development teams, number of downloads, files, and LoC, several of the WordPress plugins used in the wild are insecure. In the group of 8 plugins with two versions, all 6 plugins with vulnerabilities already registered, still have flaws in their new versions, sometimes keeping the same bugs identified in the past. For the other 2 plugins, both versions contain flaws, occasionally they have remained between versions (e.g., *Gantry Framework*) or have increased (e.g., *WP Shop* for XSS).

The new flaws were reported to the developers, and in some cases they have already been acknowledged and fixed, resulting in the release of updated versions of the plugins[4]. Overall, these experiments are encouraging because the approach demonstrated the potential for the discovery of many classes of vulnerabilities in several open-source plugins, some of them with considerable user bases.

8.1.2 Real Web Applications

To determine if DEKANT is effective at classifying the vulnerabilities belonging to the fifteen classes under study and discovering zero-day vulnerabilities, we run the tool with two sets of open source software packages, respectively, one containing well known vulnerable packages to validate the capabilities of the tool and another to find zero-day vulnerabilities regarding SQLi and XSS (the two most exploited classes of vulnerabilities).

The first set is composed of 13 applications with more than 4,000 files and almost 1 million LoC (Table 5). A few of the packages are large, such as *Play sms* and *Clip Bucket*, with approximately 250 and 150 thousand LoC. There are 727 slices evaluated in this experiment, which were classified manually to enable the validation of the outcomes of DEKANT. Table 5, in columns 6-9, displays the results of this effort, where *Vul* stands for vulnerable slices, *San* for sanitized, and *VC* for validated and/or changed.

DEKANT takes a short time to perform the analysis, in the order of tens of seconds (column 5). Columns 10-13 show that the tool correctly classifies 503 slices as being vulnerable (*Vul*), 14 slices are wrongly labeled as having bugs (FPs) and 4 have errors that remain undetected (i.e., *false negatives (FN)*). Columns 14-21 present how the 503 slices are sorted out into the fifteen classes of vulnerabilities (column Files aggregates three classes). Misclassification (FPs and FNs) is mainly explained by the presence of validation and string modification functions with context-sensitive states. In particular, most FPs belong to the class PHPCI, a type of vulnerability related to the execution of *preg_match* and *preg_replace* functions (the remaining were in classes HI and XSS). The FNs are also associated with PHPCI bugs.

4. For example, plugins *appointment-booking-calendar 1.1.7*, *easy2map 1.2.9*, *payment-form-for-paypal-pro 1.0.1*, *resads 1.0.1* and *simple-support-ticket-system 1.2* were fixed thanks to this work.

TABLE 5: Slices in open source applications processed by DEKANT.

Web application	Version	Files	LoC	Analysis time (s)	Slices				Classification				Vulnerability class							
					Vul	San	VC	Total	Vul	N-Vul	FP	FN	SQLI	XSS	Files*	SCD	HI	CS	LDAP	SF
Admin Control Panel Lite 2	0.10.2	14	1984	1	81		1	82	81		1		9	72						
Clip Bucket	2.7.0.4	597	148129	11	22	4	5	31	22	6	3			10	11		1			
Clip Bucket	2.8	606	149830	12	26	4	5	35	26	6	3		4	10	11		1			
Ldap address book	0.22	18	4615	2	40	50		90	40	50				39					1	
Minutes	0.42	19	2670	1	10			10	10				9				1			
Mle Moodle	0.8.8.5	235	59723	18	7		3	10	6	3		1		5	1					
Php Open Chat	3.0.2	249	83899	7	11			11	11					10					1	
Pivotx	2.3.10	254	108893	10	4	3	6	13	4	9				1	2					1
Play sms	1.3.1	1420	248875	19	6		2	8	5	2		1		5						
RCR AEsir	0.11a	8	396	1	13		1	14	13	1				9	3			1		
SAE	1.1	150	47207	7	148	38	15	201	148	48	5		61	65	20	1	1			
Tomahawk Mail	2.0	155	16742	3	3		3	6	3	3			2	1						
vfront	0.99.3	438	93042	15	136	50	30	216	134	78	2	2	32	68	24		10			
Total		4163	966005	107	507	149	71	727	503	206	14	4	117	295	72	1	14	1	2	1

*DT & RFI, LFI vulnerabilities

Summing-up, the results are reassuring as DEKANT correctly classifies every vulnerability that was described in [27], but actually with less FP. These results are very similar to the ones of the plugins (see Section 8.1.1), demonstrating that the tool is capable of detecting vulnerabilities and of classifying them correctly independently of their classes.

For the second set, we run DEKANT with 8 applications in order to discover zero-day vulnerabilities (Table 6). In total more than 3,000 files and almost 900,000 LoC were analyzed. The largest packages are *CandidATS* and *AMSS++*, with approximately 303 and 233 thousand LoC.

DEKANT classifies 4,129 slices as having bugs but, after we checked them manually, 92 alarms are invalid (columns 5-6). The vulnerabilities pertain to SQLI and XSS (columns 7-8) and the FPs occur mostly in the XSS class (85 out of 92) due essentially to the use of user functions to replace metacharacters in entry points and codification of entry points in base64. 3811 out of 4037 flaws correctly classified are in *AMSS++*, a support system to area management, and 127 out of the remaining 226 are in *GUnet OpenEclass E-learning*, a platform for e-learning education. Such results denote the lack of code security existent in systems and platforms highly used for management of online services. Curiously, these two applications are those we found zero-day vulnerabilities, meaning that we discovered 3938 new vulnerabilities, while the other 99 flaws are already registered in CVE [35]. We have already started the process of vulnerability disclosure and contacting the developers.

Overall, DEKANT had both high accuracy and precision. These results are very similar to the ones of the first set, demonstrating that the tool is capable of detecting vulnerabilities and of classifying them correctly independently of their classes. Given the results we obtained with plugins and web applications, questions 1 and 2 have an affirmative response.

TABLE 6: Slices in open source software with vulnerabilities discovered by DEKANT.

Web application	Version	Files	LoC	Classif.		Vul. classes	
				Vul	FP	SQLI	XSS
60CycleCMS	2.5.2	62	40,398	11		2	9
AMSS++	4.31	688	232,869	3811	58	1480	2331
CandidATS	2.1.0	1,293	303,400	50	10	15	35
eLection	2.0	4	1,070	0	0		
GUnet OpenEclass E-learning	1.7.3	592	164,025	127	15	5	122
Persian VIP	1.0	16	2,223	17		13	4
rConfig	3.9	235	91,167	13	8	2	11
YzmCMS	5.5	161	26,199	8	1	8	
Total		3,051	861,351	4,037	92	1,525	2,512

8.2 Comparison with Plugin Analysis Tools

The section tests plugin analysis tools, namely WAPe [27] and phpSAFE [36], and compares them to DEKANT. The two tools implement taint analysis in a diverse manner, but still with the aim of tracking data that flows from the entry points to the sensitive sinks. WAPe is an extension of WAP, and since it is highly configurable, we could set it up with the same knowledge about WordPress functions as DEKANT. phpSAFE only looks for SQLI and XSS vulnerabilities in WordPress plugins. Therefore, to make the comparison among tools fair, we decided to consider only these two classes in the evaluation, and accounted the slices with other bugs as not vulnerable. The experiments are based on the 25 plugins previously presented, which have a total of 471 slices (the 455 slices of Section 8.1.1 plus 16 extra slices that were extracted by the other two tools). The results are summarized in Table 7.

DEKANT evaluates 455 slices (columns 5-8) and outputs 252 of them as potentially vulnerable to SQLi and XSS. Out of this group, 247 of them have real bugs and 5 are FPs. The remaining 203 slices are correctly classified as not vulnerable. While processing the results, we observed that: (i) there are five vulnerabilities that only DEKANT is able to find; (ii) a few slices with bugs are not collected by DEKANT, which inevitably leads to FNs. This last observation confirms the fundamental role of the slice extractor in these tools, as it gets the paths in the code that end up being inspected.

WAPe discovers 164 bugs but misses 99 (columns 9 to 13). The tool includes a false positive predictor, whose aim is to look at the results of taint analysis and exclude bug reports that are potentially invalid — these are called *false positives predicted (FPP)*. After analysis, six cases are deemed FPP, leaving only one FPs. In the case of DEKANT, five of these seven slices are placed in the non-vulnerable set. WAPe and DEKANT extract 171 slices in common, but there is one slice that is only obtained by the former tool. This slice is correctly classified as vulnerable by WAPe (and causes a FN in the other tools).

phpSAFE could only process 20 plugins (out of 25) and four of them partially (columns 14 to 18). For this reason, only 361 slices out of 471 are examined. Within the group

TABLE 7: Vulnerability discovery results with WordPress plugins for DEKANT, WAPe, and phpSAFE.

Plugin	Version	Files	LoC	DEKANT				WAPe					phpSAFE				
				SQLI	XSS	FP	FN	SQLI	XSS	FPP	FP	FN	SQLI	XSS	FP	PFP	FN
Appointment Booking Calendar	1.1.7	6	2,955	3	4			1	3	1		3	3	4	2	14	
Appointment Booking Calendar	1.3.39	16	4,735	3			4	1		1		6	3	4		6	
Authorizer	2.3.6	164	159,023		2				2					1			1
Contact formgenerator	2.0.1	42	9,187	11				11							3		11
Contact formgenerator	2.1.82	47	9,753	10			3	10				3		3	25	79	10
Easy2map	1.2.9	16	3,193		1				1					1	8	10	
Ecwid Shopping Cart	3.4.6	61	16,807		1				1				–	–	–	–	1
Ecwid Shopping Cart	6.9.6	196	32,876		1		1		1			1		2		1	
Gantry Framework	4.1.6	274	50,717		3				1			2		1			2
Gantry Framework	4.1.21	287	55,266		3				1			2			3	3	3
Google Maps Travel Route	1.3.1	10	1,692	1	2	1		1	2					1	7	10	2
Payment form for Paypal pro	1.0.1	10	3,920		2				2					2	19	2	
Payment Form for PayPal Pro	1.1.64	13	4,379		2		1		2			1	1	2	19	2	
ResAds	1.0.1	30	3,168		17				2			15		17			
ResAds	2.0.3	31	3,496		13				3			10		13		6	
Simple support ticket system	1.2	20	1,533	18				18					3		2	7	15
Simple support ticket system	1.3.6	24	2,040	13	4			13	3			1				11	17
The Cart Press eCommerce Shopping	1.4.7	220	47,114	8	17			8	17				–	–	–	–	25
WP Easy Cart - eCommerce Shopping	3.2.3	623	126,448	13	6			13	6				–	–	–	–	19
WP Marketplace	2.4.1	88	15,485	2	24	3	3		9		1	20	2	27	18	30	
WC Marketplace	3.4.11	500	78,472		19	1	4		14	1		9		23	6	25	
WP Shop	3.5.3	49	9,171	7	10				5	1		12	7	10	5	29	
WP Shop	3.9.6	102	22,480	4	17			2	5	2		14	4	17	8	18	
WP Simple e-Commerce Shopping Cart	2.2.5	92	21,003	2	1			2	1				–	–	–	–	3
WP Web Scraper	3.5	89	8,116		3				3				–	–	–	–	3
Total		3010	693029	95	152	5	16	80	84	6	1	99	23	128	125	253	112

of analyzed slices, there are 151 vulnerabilities that are found and 112 that are missed. However, phpSAFE finds three errors that no other tool is able to discover. The 125 FPs are caused by the inclusion of sanitization and input change functions in the slices, such as *substr* and *preg_replace* from PHP and *esc_attr*, *esc_html* and *prepare* from WordPress (the last one protects a SQL statement from SQLI attacks, providing similar functionality as prepared statements).

phpSAFE scans 253 extra slices (aside from the 471 group), which are labeled as *possible false positives (PFP)* in our evaluation. These cases are associated with parts of the code where the results of SQL queries are used in some sink (e.g., to embed database content in a web page returned to a browser). The tool considers any of these results as malicious input, independently of the type of query (e.g., an INSERT or UPDATE SQL command) and the sanitization of query' parameters. In addition, the tool does not seem to correlate these queries with the ones that insert data in the database, and therefore it is difficult to conclude that these slices have any real problem. Therefore, due to this ambiguity, we keep these slices separate from the rest.

TABLE 8: Evaluation metrics of DEKANT, WAPe, phpSAFE, PhpMinerII, and NAVEX-f for the detection of SQLI and XSS.

Metric	Plugins			WebApps – Data mining			WebApps – CPG	
	DEKANT	WAPe	phpSAFE	DEKANT	WAPe	PhpMiner II	DEKANT	NAVEX-f
acc	0.96	0.79	0.50	0.98	0.97	0.78	0.97	0.04
pr	0.98	0.99	0.58	0.98	0.96	0.57	0.98	0.65
fpr	0.03	0.01	0.58	0.006	0.01	0.07	0.65	0.35
fnr	0.06	0.38	0.45	0.08	0.10	0.72	0.005	0.98

acc: accuracy; pr: precision; fpr: false positive rate; fnr: false negative rate
CPG: Code Property Graphs

Table 8 has the metrics results for the three tools (columns 2-4). DEKANT is superior with the highest combined accuracy and precision and low FP and FN rates. WAPe is second, being the tool with the lowest FP rate and the second highest FN rate. phpSAFE has the worst performance, with significantly lower accuracy and precision. Notice that the 253 PFPs of phpSAFE are disregarded from the calculations. Based on results we got, we can answer question 3 positively.

8.3 Comparison with Standard Classifier Tools

There are no recent tools in the literature that apply machine learning, such as deep learning or NLP techniques, to discover vulnerabilities in PHP code. The tools mentioned in Section 11 resorting deep learning are dedicated to C programs (e.g., VulDeePecker [20], Devign [37], and Chucky [38]), in which their bugs are different than those in web applications. A few other tools have implemented data mining mechanisms for tasks related with bug discovery in PHP, namely WAPe and PhpMinerII [22], [23]. WAPe and PhpMinerII classify slices by resorting to data mining with standard classifiers, which do not consider order. WAPe obtains the slices with taint analysis and then predicts if they are FPs or TPs with the classifiers, with the aim of reducing the alerts that are generated by mistake. PhpMinerII uses data mining to find out if slices hold attributes that make them look vulnerable, without specific concerns about false positives. This tool handles only SQLI and reflected XSS vulnerabilities.

Since PhpMinerII is not configurable with information about WordPress, and consequently it would perform much worse with plugins, we opted to experiment with the first set of 13 application packages.

We observed that the various tools (from this section) survey different groups of slices because of their specific implementation of the slice extractor. Therefore, we decided to create a superset with all slices that could be captured based on the outputs of the tools, which contains 1852 slices. This set was then manually examined to determine which slices are vulnerable, and it serves as a ground truth. Overall there are 541 slices with vulnerabilities (117 SQLI, 333 XSS, and 91 others) and 1311 slices without problems. This second group was divided in a few subsets, namely, slices with sanitized input, slices with validated or modified input, and

TABLE 9: Comparison of results between DEKANT, WAPe, and PhpMinerII with open source projects.

Web application	DEKANT					WAPe						PHPMiner II			
	SQLI	XSS	others	FP	FN	SQLI	XSS	others	FPP	FP	FN	SQLI	XSS	FP	FN
Admin Control Panel Lite 2	9	72		1		9	72		8	1		9	23	1	49
Clip Bucket		10	12	3	9		10	12	2	4	9		9	20	
Clip Bucket	4	10	12	3	9	4	10	12	2	4	9	3	9	17	1
Ldap address book		39	1				36	1		2	3				39
Minutes	9		1		5	6		1			8		5	7	9
Mle Moodle		5	1		5		5	1	3		5		5	27	
Php Open Chat		10	1				9	1			1		9	7	
Pivotx		1	3		3		1	3	9		3		3	1	
Play sms		5			7		5		2		7		7	12	
RCR AEsir		9	4				9	4	1				3		6
SAE	61	65	22	5		61	65	20		10	2		8	2	118
Tomahawk Mail	2	1				2	1		3				1	1	2
vfront	32	68	34	2		32	68	34	24	2			1		96
Total	117	295	91	14	38	114	291	89	54	23	47	12	83	95	320

slices without external sources (i.e., without entry points) but with a sensitive sink. This last group was provided by PhpMinerII and we designate it as the *no-source* subset.

8.3.1 All Vulnerability Classes

A summary of the experimental results is included in Table 9. The vulnerabilities are distributed by classes SQLI, XSS and others, to facilitate the assessment of alternative tools that only address specific bugs (like PhpMinerII). Columns 2 to 6 are about DEKANT, displaying a total of 503 identified bugs. Notice that there are 34 more FNs than in Table 5 because now we are covering a larger number of slices, some of which are not extracted by DEKANT. The next six columns display WAPe's results. WAPe reports less vulnerabilities and a few more FPs and FNs.

With regard to false positives, DEKANT judges correctly as not vulnerable the 71 validated and/or changed slices (i.e., column *VC* in Table 5) but WAPe just predicts 48 of them as FPP. Even though WAPe handles a considerable number of symptoms to reduce mistakes, there is a lack of attribute relation verification that induces erroneous decisions — the tool only checks if attributes exist in a slice but does not have a way to relate them.

The difference in false negatives between the tools is also explained by the same reason, plus the importance of considering the order of the code elements in the slice. In particular, a misclassification can occur when there is a concatenation of tainted with untainted variables (i.e., which were validated or modified); this causes the data mining classifier to find symptoms related with validation and outputs the slices as FPs. DEKANT implements a sequence model that takes into account how the code elements appear in the slice, prevailing in these situations.

Table 10 sums up de evaluation, combining the confusion matrix and metrics. The results are encouraging with DEKANT performing a bit better than WAPe, namely because it shows superior FP and FN rates.

TABLE 10: Confusion matrix of DEKANT and WAPe for the detection of all vulnerability classes.

	Observed				Metric	DEKANT	WAPe
	DEKANT		WAPe		acc	0.97	0.96
Predicted	Vul	N-Vul	Vul	N-Vul	pr	0.97	0.96
Vul	503	14	494	23	fpr	0.010	0.017
N-Vul	38	1297	47	1288	fnr	0.07	0.09

acc: accuracy; pr: precision; fpr: false positive rate; fnr: false negative rate

8.3.2 Just SQLI and XSS

This subsection only considers SQLI and reflected XSS for a fair comparison with PhpMinerII. PhpMinerII does not come trained when downloaded, and so we had to build a dataset for that purpose. The training dataset was constructed by recreating the procedure explained in [22], [23], where the WEKA package implemented the data mining tasks [39]. The same classifiers were evaluated to select the best. Overall, the C4.5/J48 classifier was chosen, with an accuracy and precision close to 0.92.

Table 9 has the results for PhpMinerII. The tool obtains 1052 slices, where 219 are reported as vulnerable and 833 as not-vulnerable. Manually, we inspected these slices and found out that only 604 were correctly labeled, 124 as vulnerable and 480 as not-vulnerable. Consequently, the tool generates 95 FP and 320 FN. This notable misclassification is explained by various factors, such as missing validations and string modifications of inputs, and not taking into account the order of code elements. In addition, some of the slides belong to the no-source subset and they lead necessarily to invalid alarms (as there is no entry point to be maliciously exploited).

DEKANT outputs 412 vulnerabilities and 8 incorrect reports (out of the 14 shown in table). It also misses 38 slices with bugs (out of the 38 shown in table). WAPe classifies 405 vulnerabilities, but with 16 FPs (of the 23 presented in table) and 47 FNs (out of the 47). Only 82 of the 124 identified bugs by PhpMinerII are also flagged as being vulnerable by DEKANT and WAPe. This means that the 42 remaining vulnerable slices justify the increase of FN in the two tools.

Table 8 displays the calculated metrics when only SQLI and XSS are contemplated. DEKANT and WAPe surpass PhpMinerII, exhibiting higher quality values for all metrics. Both DEKANT and WAPe have an excellent accuracy and precision, but the former is superior with 0.98 on both metrics. In addition, DEKANT has better rates for false positives and false negatives. Therefore, we can answer question 4 positively, based on the results we obtained from both experiments above.

8.4 Comparison with Code Property Graph Tools

There are two recent tools that explore code property graphs and taint analysis to locate vulnerabilities in PHP, namely Joern-php[5] and NAVEX[6]. They perform traversal graphs

5. https://github.com/octopus-platform/joern/
6. https://github.com/aalhuz/navex

to extract path-findings (i.e., data flows). Traversal graphs apply a taint analysis bottom-up approach, i.e., they identify sensitive sinks and then perform a backtrack taint analysis to extract data flows and determine if they start with an entry point. Such traversals are written in Gremlin [40], a graph traversal language able to perform queries over a graph database, such as Neo4j [41]. Both tools contain the *joernsteps* API to create the queries more easily, being built for Joern-php and later enhanced in NAVEX.

For Joern-php, we tried to contact the authors in order to reproduce the tool and use the queries employed in [42]. However, we did not get any answer. On the other hand, NAVEX is reproducible and contains the queries [43], but the mentioned API is not totally available, missing the two fundamental and critical functions that do the taint analysis and execute the queries. However, a fixed version of the tool can be found at [44], thanks to an anonymous author, which we will call NAVEX-f.

Therefore, our evaluation compares DEKANT with NAVEX-f for the detection of SQLi and XSS vulnerabilities while processing the second set of 8 application packages of Section 8.1.2. To the effect, we built the traversal graphs for both classes of flaws, which were configured with the sensitive sinks, sanitization functions and entry points presented in [43].

Similarly to what happened in Section 8.3, we observed that NAVEX-f extracted different slices than those extracted by DEKANT due the approach and code structure it uses. Therefore, as we did in that section, we created a dataset with all slices that were gathered by both tools, which contains 4,198 slices, with 20 of them being common to both tools. We analyzed manually these slices to find out which are vulnerable, and the results serve as a ground truth. Overall there are 4,057 vulnerable slices (1,527 SQLI, 2,530 XSS) and 142 that are correct.

TABLE 11: Slices in open source software with zero-day vulnerabilities and bugs disclosed in the past, analyzed by DEKANT and NAVEX-f.

Web application	Ver	DEKANT				NAVEX-f			
		SQLI	XSS	FPl	FN	SQLI	XSS	FPl	FN
60CycleCMS	2.5.2	2	9		3	1	5	3	8
AMSS++	4.31	1480	2331	58		–	–	–	3811
CandidATS	2.1.0	15	35	10		15	35	10	
eLection	2							20	
GUnet OpenEclass E-learning	1.7.3	5	122	15				1	127
Persian VIP Download Script	1	13	4		8	15	8	3	2
rConfig	3.9	2	11	8	9		11	7	11
YzmCMS	5.5	8		1				5	8
Total		1,525	2,512	92	20	31	59	49	3,967

The results of the evaluation are displayed in Table 11. Columns 3 to 6 are dedicated to DEKANT, where the first three of them are a copy of Table 6. Therefore, DEKANT classified as vulnerable 4,037 slices correctly and 92 slices incorrectly (FP), and missed 20 slices (FN), which these were only extracted by NAVEX-f. NAVEX-f results are displayed in the last four columns of the table. The tool was unable to process *AMSS++*, the application that had more slices and was the most vulnerable. It gathered 139 slices, meaning that it generated this amount of alerts. Out of this group, 90 correspond to slices with real bugs and the remaining 49 to false alerts. These FPs occur essentially in slices with functions that encode data (e.g., *json_encode*) and with connections to the database where the name of the database is an entry point, even when the queries are static (queries without variables). This demonstrates the importance of the identification of false positive symptoms and of evaluating the slices taking into consideration the order of code elements (like DEKANT does). NAVEX-f does not catch 3,967 vulnerabilities (FN) from the ground truth, but 3,811 are associated with the package that the tool was not able to process.

Table 8 presents the metrics for these tools (last two columns). The results corroborate the promising detection capabilities of DEKANT, as the tool that has the best accuracy and precision and the lowest FN rate. Regarding FP, the tool presents the highest rate due the 58 FPs found in that most vulnerable application. NAVEX-f had the worst rates due to missing that application. However, if we omit such package from the comparison, DEKANT still has better results than NAVEX-f, and its FP rate decreases significantly. Given these observations, question 5 has a positive answer.

9 Threats to Validity

This section presents the threats to validity following the four main categories presented by Cook and Campbell [45].

- *Internal Validity.* The process design conducted in our experiments was aimed at validating and evaluating our model. For this and to reach the highest level of confidence of the experimental results, and thus to minimize the primary threat to internal validity – application selection –, we chose two types of software (plugins and web applications), purposefully vulnerable (i.e., with publicly registered vulnerabilities) and not-vulnerable (i.e., with no vulnerability repositories records). These applications were also carefully selected from various contexts (e.g., contacts, market), and with different code sizes and complexity (e.g., ranges from less than 1000 LoC to well over 100,000 LoC). Validation was done on the vulnerable software to determine that the model was able to detect the known vulnerabilities, and the assessment was made with the not-vulnerable software to check the model's ability to discover previously unknown (zero-day) vulnerabilities. With the latter software, the model was also compared with other tools, from different categories, to check if it outperformed them. Furthermore, for the construction of the ground-truth datasets, throughout the experiments, we did not consider only the slices extracted by DEKANT, but all the slices returned by the different tools used in the experiments. With these choices, we attempted to enhance completeness of data and relevant factors, considering the confidence in the outcomes obtained in the experiments with the proposed model.
- *Construction Validity.* To minimize these types of threats, namely the appropriateness of data, experiments bias and measurement method, we manually constructed the corpus, which the model learns and uses to detect vulnerabilities. In addition, the ground-truth datasets were manually checked and labeled as vulnerable or not-vulnerable. Also, we assessed the corpus in Section 6 before the experimental evaluation of Section 8, as the reliability of the latter depends on the completeness of

the former. The summary of this assessment, shown in Table 3, indicated that the corpus contains appropriate and enough data to avoid experiment bias of the model, which in fact was shown in Section 8 and hence all research questions had a positive response. However, there is always the possibility that the performance of the model may decrease if the PHP slices to be classified contain functions that are not recognized by any token. We envisage two threat cases: (1) slices with classes of vulnerabilities not addressed by the model (and corpus) which may contain sensitive sinks that are not recognized by the token `ss`; (2) any other PHP function not associated with a token, which is not a sensitive sink or a sanitization/validation, but that may be relevant for the correct vulnerability discovery. In both situations, the ISL translator will ignore such functions since it has no knowledge about them, and, consequently, the model will likely behave inappropriately and classify incorrectly. These cases can be resolved by updating the tokens with the new functions (sinks and other functions) and by extending the corpus with slices from such classes of vulnerabilities. However, until discovering the symptoms of the threats that affect the model and making the necessary updates to the corpus and tokens, the results of the model will remain inaccurate.

- *Conclusion Validity.* Our dataset of 6,521 slices was obtained from the 46 plugins and web applications. More specifically, 471 slices were extracted from plugins, 1,852 slices were provided from 13 known vulnerable web applications, and 4,198 slices were collected from 8 web applications that we did not know their security state. This number of slices is quite high, minimizing the threats to the reliability of the derived conclusions.
- *External Validity.* The results of the experiments conducted in Section 8 showed that DEKANT was able to process 46 different types of applications (plugins and web applications) with different characteristics. The metrics of precision and accuracy, obtained from the results, presented values close to 1. Hence, based on the characteristics and metrics, the risk of low generalization is reduced in terms of software that the DEKANT can externally process. Another aspect that our evaluation allowed was the possibility of DEKANT processing slices extracted by other tools. Again, this reduces the risk of low generalization as traditional static analysis tools can act as slice collectors of DEKANT.

10 Real-World Applicability of DEKANT

DEKANT is a static analysis tool as it looks for vulnerabilities in the source code without executing it. The tool has two main parts: one programmed and one learned. The first corresponds to the slice collector that does what other static analysis tools do, i.e., it parses code and extracts slices that start at entry points and end at a sensitive sink. The second uses an HMM, the sequence model we propose, configured with the knowledge of vulnerabilities contained in the corpus, to determine whether the slices are vulnerable or not. The model considers not only the presence of the code elements belonging to the slices but also the order in them and the relations between them, minimizing hence the FP and FN rates.

DEKANT can be extended for other languages and some of its modules can be exchanged and/or extended. To show that this is the case, we discuss how it can be adapted and applicable to different real-world scenarios, for PHP and other server-side languages.

10.1 PHP Language

The current implementation of DEKANT is for PHP and it can be used to inspect real web applications and plugins. There are three areas where improvements can occur:

- *Extend the PHP code elements.* Add extra code elements to the third column of Table 1 that are associated with the tokens. To do so, the user needs to insert the elements into the correct configuration files. For example, suppose a new entry point is defined for PHP, the user can add it to the configuration file of the `input` token.
- *Extend the corpus.* Add more annotated sequences of observations to the corpus. In this case, the task may be more time-consuming, as the user needs to analyze which sequences are currently included in the corpus and then attach the new ones. But a good way to accomplish this task is to incrementally append the slices that the tool will misclassify. This involves the effort of manually checking whether the tool is wrong, adding the slices to the corpus, after translating them to ISL and annotating them, and then removing the duplicated sequences (see Section 6.1).
- *Extend with new tokens.* Define new tokens that are not present in the first column of Table 1. This task, depending on the type of token the user wants to create, can be harder to implement, as it may require internal tool modifications, namely, the grammar, the slice translator and the vulnerability detector (i.e., to the extensions we made to the Viterbi algorithm), and requires the inclusion in the corpus of annotated sequences of observations related to the token.

Despite these types of improvements that can be made, we believe that the third is the most improbable to be necessary, as the tool showed high precision and accuracy when assessed with different types of applications. In contrast, the other two improvements are more likely to be done, although the current corpus is quite reliable based on the results we obtained in Section 6.2 and the analysis we made to PHP functions to identify the most relevant code elements. However, as PHP continues to evolve, new releases may contain new functions that may be related to vulnerabilities, requiring improvements at these two levels.

10.2 Other Server-Side Languages

DEKANT can be adapted to process other server-side languages, such as Java, Python and ASP. The easiest approach to do this is by keeping the current tokens, changing the code elements presented in the third column of Table 1, and making the necessary adjustments in the slice translator module to handle the new language. For example, if we intend to adapt the tool to Java, the following steps would have to be performed: (1) analyze the Java code elements to

map them to the existing tokens; (2) create the corresponding configuration files for Java; (3) make the adjustments in the slice translator module. For example, for the slice translator to verify whether a Java instruction is an assignment, instead of checking if it starts with the $ character (as in the case of PHP), it will check if the = operator succeeds the first code element of the instruction. Note that with this process, no new corpus will be needed to support the new programming language, since the sequences of the corpus are made up of tokens (not code elements).

11 Related Work

This section summarizes the main related work in the areas of static analysis, code property graphs and machine learning for the detection of vulnerabilities.

11.1 Detecting vulnerabilities with static analysis

Static analysis tools search for vulnerabilities in the applications usually by processing the source code (e.g., [11], [12], [46], [47], [10], [13], [42]). Many of these tools perform taint analysis, tracking user inputs to determine if they reach a sensitive sink (i.e., a function that could be exploited). CQUAL [46] and Splint [48] were the first to implement this technique (both for the C language), using two qualifiers – *tainted* and *untainted* – to manually annotate certain parts of the program (e.g., function parameters or return values) where untrusted / trusted data may flow. User inputs were followed through the code to find out if tainted data would arrive to a parameter labeled as untainted. If this happened, an alarm would be raised.

Pixy [12] was one of the first tools to automate this kind of analysis on PHP applications. Later on, RIPS [10] extended this technique with the ability to process more advanced PHP constructs (e.g., objects). phpSAFE [11], [36] is a recent solution that does taint analysis to look for flaws in CMS plugins (e.g., WordPress plugins). Besides taking into account the sanitization functions from PHP, it is configured to recognize CMS functions handling entry points, sanitization/validation and sensitive sinks.

Static analysis tools tend to generate many false positives and false negatives due to the complexity of coding knowledge about vulnerabilities. WAP [13], [27] also does taint analysis, but aims at reducing the number of false positives by resorting to data mining, besides also correcting automatically the located bugs. WAPe [27] is an extension of WAP, which allows the user to configure the detection of new vulnerability classes.

11.2 Detecting vulnerabilities with code property graphs

Yamaguchi et al. [49] presented a method for a more precise static analysis that explores a data structure called code property graph. They combine different source code representation graphs, such as abstract syntax trees (AST), control flow graphs (CFG) and program dependence graphs (PDG), in a single graph, and then query the graph to extract data flows and analyze them in order to discover vulnerabilities.

Joern [49], [50] was the first tool that implemented this approach for C programs. Later, it was extended for PHP programs, becoming the Joern-php tool [42]. NAVEX [43] is a tool that improved the latter and added other features, but not related to code property graphs. However, the vulnerability detection for these three tools is made manually over the outputted data flows.

In this paper, we propose a novel approach which, unlike these works of static analysis and code property graphs, does not involve programming information about bugs, but instead extracts this knowledge from annotated code samples and thus learns to find the vulnerabilities automatically. The slices extracted from the source code, i.e., excerpts of code that begin in entry points and end in a sensitive sink, are processed by the DEKANT tool to discover if they are vulnerable or not, i.e., contain or not a vulnerability.

11.3 Vulnerabilities and machine learning

Machine learning has been used in a few works to measure the quality of software by collecting a series of attributes that reveal the presence of software defects [51], [52]. Other approaches resort to machine learning to predict if there are vulnerabilities in a program [53], [54], [55], which is different from identifying precisely the bugs, something that we do in this paper. To support the predictions they employ various features, such as past vulnerabilities and function calls [53], or a combination of code-metric analysis with metadata gathered from application repositories [55].

In particular, PhpMinerI and PhpMinerII predict the presence of vulnerabilities in PHP programs [22], [23], [56]. The tools are first trained with a set of annotated slices that end at a sensitive sink (but do not necessarily start at an entry point), and then they are ready to identify slices with errors. WAP and WAPe are different because they use machine learning and data mining to predict if a vulnerability detected by taint analysis is actually a real bug or a false alarm [13], [27]. In any case, PhpMiner and WAP tools employ standard classifiers (e.g., Logistic Regression or a Multi-Layer Perceptron) instead of structured prediction models (i.e., a sequence classifier) as we propose here.

There are a few static analysis tools that implement machine learning techniques. Chucky [38] discovers vulnerabilities by identifying missing checks in C language software. The tool does taint analysis to locate the checks between entry points and sensitive sinks, applies text mining to discover the neighbors of these checks, and then builds a model to see if there are checks that might be absent. Soska et al. aim to predict whether a website will become malicious in the future, before it is actually compromised [57]. Scandariato et al. [58] performs text mining to predict vulnerable software components in Android applications. SuSi [59] employs machine learning to classify sources and sinks in the code of Android API.

Recently, deep learning has started to be applied in the vulnerability detection field [60], [61], [20], [37], [17], [18], essentially in finding C/C++ bugs [20], [37]. VulDeePecker [20] resorts to code gadgets to represent parts of C programs and then transforms them into vectors. A neural network system then determines if the target program is vulnerable due to buffer or resource management errors. Russell et al. [61] developed a vulnerability detection tool for C/C++ based on features learning from a dataset and

artificial neural network. There are very few models for finding faults in web applications [17], [18], [19], which follow a something similar approach for finding C/C++ bugs.

Deep learning requires big datasets for training the models and is not able to identify and explain the presence of vulnerabilities it classifies as such, due to its black-box nature which hides its internal logic and makes it difficult to understand the classification operation [62]. To attain that, Devign [37] combines traditional graph code representations (e.g., AST and CFG) and Natural Code Sequence in a same graph. However, the approach creates an overly complicated representation of the code though. Moreover, its code and dataset are unavailable to the public and the results are questionable, considering the outdated tools they compare the model with.

Our approach works differently from all these approaches. It extracts PHP slices, but contrary to the others it translates them into a tokenized language to be processed by a HMM. While tools in the literature collect attributes from a slice and classify them without considering ordering relations among statements, DEKANT also does classification but takes into account the place in which code elements appear in the slice and shows the classification it makes for every code element it processes. Such form of classification assists on a more accurate and precise detection of bugs.

12 Conclusion

The paper explores a new approach to detect web application vulnerabilities inspired by NLP in which static analysis tools *learn* to detect vulnerabilities automatically using machine learning. Whereas in classical static analysis tools it is necessary to code knowledge about how each vulnerability is detected, our approach obtains knowledge about vulnerabilities automatically.

The approach uses a sequence model (HMM) that, first, learns to characterize vulnerabilities from a corpus composed of sequences of observations annotated as vulnerable or not, then processes new sequences of observations based on this knowledge, taking into consideration the order in which the observations appear.

The model was implemented in the DEKANT tool. The tool was evaluated with WordPress plugins and real software packages, and compared with five tools from diverse vulnerability detection techniques. Overall, the results showed that the tool presents promising capabilities of detection and identification of vulnerabilities.

Acknowledgments

This work was partially supported by the national funds through FCT with reference to SEAL project (PTDC/CCI-INF/29058/2017, LISBOA-01-0145-FEDER-029058, POCI-01-0145-FEDER-029058), LASIGE Research Unit (UIDB/00408/2020 and UIDP/00408/2020), and INESC-ID Research Unit (UIDB/50021/2020).

References

[1] J. Williams and D. Wichers, "OWASP Top 10 2017 – The Ten Most Critical Web Application Security Risks," 2017.

[2] Imperva, "The state of web application vulnerabilities in 2019," Jan. 2020.

[3] BBC Technology, "Millions of websites hit by Drupal hack attack," Oct. 2014, http://www.bbc.com/news/technology-29846539.

[4] The Hacker News, "Wordpress plugin used by 300,000+ sites found vulnerable to sql injection attack," Jun. 2017, https://thehackernews.com/2017/06/wordpress-hacking-sql-injection.html.

[5] threatpost, "Million-plus wordpress sites exposed by vulnerable plugin," 2017, https://threatpost.com/million-plus-wordpress-sites-exposed-by-vulnerable-plugin/123983/.

[6] The Hacker News, "It's 3 billion! yes, every single yahoo account was hacked in 2013 data breach," Oct. 2017, https://thehackernews.com/2017/10/yahoo-email-hacked.html.

[7] Help Net Security, "Hacker breached 60+ unis, govt agencies via SQL injection," Feb. 2017, https://www.helpnetsecurity.com/2017/02/16/hacker-govt-agencies-via-sql-injection/.

[8] Sink, "XSS attacks: The next wave," Jun. 2017, https://snyk.io/blog/xss-attacks-the-next-wave/.

[9] Imperva, "The state of web application vulnerabilities in 2017," Dec. 2017.

[10] J. Dahse and T. Holz, "Simulation of built-in PHP features for precise static code analysis," in *Proceedings of the 21st Network and Distributed System Security Symposium*, Feb 2014.

[11] J. Fonseca and M. Vieira, "A practical experience on the impact of plugins in web security," in *Proceedings of the 33rd IEEE Symposium on Reliable Distributed Systems*, Oct. 2014, pp. 21–30.

[12] N. Jovanovic, C. Kruegel, and E. Kirda, "Precise alias analysis for static detection of web application vulnerabilities," in *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, Jun. 2006, pp. 27–36.

[13] I. Medeiros, N. F. 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, March 2016.

[14] J. Dahse and T. Holz, "Experience report: An empirical study of PHP security mechanism usage," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, Jul 2015, pp. 60–70.

[15] H. Hanif, M. H. N. Md Nasir, M. F. Ab Razak, A. Firdaus, and N. B. Anuar, "The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches," *Journal of Network and Computer Applications*, vol. 179, p. 103009, 2021.

[16] NIST, "SAMATE - Software Assurance Metrics and Tool Evaluation," https://samate.nist.gov/.

[17] Y. Fang, S. Han, C. Huang, and R. Wu, "TAP: A static analysis model for php vulnerabilities based on token and deep learning technology," *PLoS One*, vol. 14, no. 11, Nov 2019.

[18] A. Fidalgo, I. Medeiros, P. Antunes, and N. Neves, "Towards a deep learning model for vulnerability detection on web application variants," in *13th IEEE International Conference on Software Testing, Verification and Validation Workshops*, Oct. 2020, pp. 465–476.

[19] R. Rabheru, H. Hanif, and S. Maffeis, "A hybrid graph neural network approach for detecting PHP vulnerabilities," *CoRR*, vol. abs/2012.08835, Dec. 2020.

[20] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Annual Network and Distributed System Security Symposium*, Feb. 2018.

[21] C. Li, Y. Wang, C. Miao, and C. Huang, "Cross-site scripting guardian: A static xss detector based on data stream input-output association mining," *Applied Sciences*, vol. 10, no. 14, Jul. 2020.

[22] L. K. Shar and H. B. K. Tan, "Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 1293–1296.

[23] ——, "Predicting common web application vulnerabilities from input validation and sanitization code patterns," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 310–313.

[24] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[25] W. W. T. Surveys, https://w3techs.com/technologies/overview/programming_language/.

[26] I. Medeiros, N. F. Neves, and M. Correia, "DEKANT: a static analysis tool that learns to detect web application vulnerabilities," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, Jul. 2016.
[27] ——, "Equipping WAP with weapons to detect vulnerabilities," in *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2016.
[28] L. E. Baum and T. Petrie, "Statistical inference for probabilistic functions of finite state markov chains," *The Annals of Mathematical Statistics*, vol. 37, no. 6, pp. 1554–1563, 1966.
[29] D. Jurafsky and J. H. Martin, *Speech and Language Processing*. Prentice Hall, 2008.
[30] N. A. Smith, *Linguistic Structure Prediction*. Graeme Hirst, 2011.
[31] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 260–269, Apr. 1967.
[32] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *The Journal of Machine Learning Research*, vol. 7, pp. 1–30, Dec 2006.
[33] M. O. Rabin and D. Scott, "Finite automata and their decision problems," *IBM Journal of Research and Development*, vol. 3, no. 2, pp. 114–125, 1959.
[34] WordPress, https://wordpress.org/.
[35] CVE, http://cve.mitre.org.
[36] 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*, Jun. 2015.
[37] 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*, Dec. 2019, pp. 10 197–10 207.
[38] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *Proceedings of the 20th ACM SIGSAC Conference on Computer Communications Security*, Nov. 2013, pp. 499–510.
[39] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, 3rd ed. Morgan Kaufmann, 2011.
[40] Apache, "Apache tinkerpop. The gremlin graph traversal machine and language." https://tinkerpop.apache.org/gremlin.html.
[41] Neo4j., https://neo4j.com.
[42] 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)*, Apr. 2017, pp. 334–349.
[43] 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*, Aug. 2018, pp. 377–392.
[44] NAVEX-fixed., https://github.com/UUUUnotfound/Navex_fixed.
[45] T. D. Cook and D. T. Campbell, *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin, 1979.
[46] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, "Detecting format-string vulnerabilities with type qualifiers," in *Proceedings of the 10th USENIX Security Symposium*, Aug. 2001.
[47] S. Son and V. Shmatikov, "SAFERPHP: Finding semantic vulnerabilities in PHP applications," in *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, 2011.
[48] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Software*, pp. 42–51, Jan/Feb 2002.
[49] 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*, May 2014, pp. 590–604.
[50] 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*, May 2015, pp. 797–812.
[51] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *Journal of Systems and Software*, vol. 83, no. 1, pp. 2–17, 2010.
[52] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, 2008.
[53] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007, pp. 529–540.
[54] J. Walden, M. Doyle, G. A. Welch, and M. Whelan, "Security of open source web applications," in *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 545–553.
[55] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, Oct 2015, pp. 426–437.
[56] L. K. Shar, H. B. K. Tan, and L. C. Briand, "Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis," in *Proceedings of the 35th International Conference on Software Engineering*, 2013, pp. 642–651.
[57] K. Soska and N. Christin, "Automatically detecting vulnerable websites before they turn malicious," in *Proceedings of the 23rd USENIX Security Symposium*, Aug. 2014, pp. 625–640.
[58] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2014.
[59] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," in *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS)*, Feb. 2014.
[60] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, Mar. 2016, p. 85?96.
[61] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. A. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley, "Automated vulnerability detection in source code using deep representation learning," in *Proceedings of the International Conference on Machine Learning and Application (ICMLA)*, Dec. 2018.
[62] R. Shwartz-Ziv and N. Tishby, "Opening the black box of deep neural networks via information," 2017, arXiv preprint arXiv:1703.00810.

Ibéria Medeiros is an Assistant Professor in the Department of Informatics, at the Faculty of Sciences of University of Lisbon (FCUL). She is a member of the LASIGE research unit, and the Navigators research group. Her research interests are in software security, cybersecurity, vulnerability and attack detection, and machine learning. She is author of tools for software security and cybersecurity, which WAP (Web Application Protection) is the most known and an OWASP project. Currently, she is the principal investigator of the SEAL national project and the XIVT European project, and has been involved in international and national projects, including the ADMORPH, DiSIEM, SEGRID, and MASSIF European projects, and the REDBOOK national project. More information about her at http://www.di.fc.ul.pt/∼imedeiros/.

Nuno Neves is Professor at the Department of Computer Science, Faculty of Sciences of the University of Lisboa. He is Head of the Department, leads the Navigators research group and he is on the scientific board of the LASIGE research unit. His main research interests are in security and dependability aspects of distributed systems. Currently, he is investigator in several projects, such as SEAL and uPVN. His work has been recognized in several occasions, for example with the IBM Scientific Prize and the William C. Carter award. He is vice-chair of the IEEE Computer Society TC on Dependable Computing and Fault Tolerance. More information about him can be found at http://www.di.fc.ul.pt/∼nuno/.

Miguel Correia is Professor at Instituto Superior Técnico (IST) of Universidade de Lisboa (ULisboa), and a Senior Researcher at INESC-ID in the Distributed Systems Group (GSD). He has been involved in several international and national research projects related to cybersecurity, including the DE4A, BIG, SPARTA, QualiChain, SafeCloud, PCAS, TCLOUDS, ReSIST, CRUTIAL, and MAFTIA European projects. He has more than 200 publications and is Senior Member of the IEEE. More information about him at http://www.gsd.inesc-id.pt/∼mpc/.