

Statically Detecting Vulnerabilities by Processing Programming Languages as Natural Languages

IBÉRIA MEDEIROS, LASIGE, Faculdade de Ciências, Universidade de Lisboa - Portugal

NUNO NEVES, LASIGE, Faculdade de Ciências, Universidade de Lisboa - Portugal

MIGUEL CORREIA, INESC-ID, Instituto Superior Técnico, Universidade de Lisboa - Portugal

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 bugs 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 hundred vulnerabilities belonging to 12 classes of input validation vulnerabilities, where 62 of them were zero-day.

CCS Concepts: •Security and privacy → Vulnerability management; Web application security; •Computing methodologies → Natural language processing; Speech recognition;

ACM Reference format:

Ibéria Medeiros, Nuno Neves, and Miguel Correia. 2016. Statically Detecting Vulnerabilities by Processing Programming Languages as Natural Languages. 1, 1, Article 1 (January 2016), 31 pages.

DOI: 10.1145/nnnnnnn.nnnnnnn

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 [39]. Millions of websites have been compromised since Oct. 2014 due to vulnerabilities in plugins of Drupal [4] and WordPress [35, 36], 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) [10, 34]. In addition, the next wave of XSS attacks has been predicted for the past two years, with an important expected growth of the problem [11, 31].

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 series of instructions) of the web application. Therefore, a good practice to enhance security is to pass inputs through *sanitization*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2016 ACM. XXXX-XXXX/2016/1-ART1 \$15.00

DOI: 10.1145/nnnnnnn.nnnnnnn

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 [6, 9, 12, 17], which is a complex task. Moreover, this knowledge might be incomplete or partially wrong, making the tools inaccurate [7]. 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*. It leverages from artificial intelligence (AI) concepts, more precisely from classification models for sequences of observations that are commonly used in the field of natural language processing (NLP). NLP is a confluence of AI and linguistics, which involves intelligent 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 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 given morpho-syntactic category) is assigned to each word in a given sentence, according to 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 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 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 [28, 29]. WAP and WAPe use a taint analyser to search for vulnerabilities and a standard classifier to confirm that the found bugs¹ can actually create security problems [17]. 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*).

Our sequence model is a *Hidden Markov Model* (HMM) [23]. 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.

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

The paper also presents the *hidDEn marKov model diAgNosing vulnerabiliTies* (DEKANT) tool that implements our approach for applications written in PHP. The tool was evaluated experimentally with a diverse set of 23 open source web applications with bugs disclosed in the past. These applications are substantial, with an aggregated size of around 8,000 files and 2.5 million lines of code (LoC). All flaws that we are aware of being previously reported were found by DEKANT. More than one thousand slices were analyzed, 714 were classified as having vulnerabilities and 305 as not. The false positives were in the order of two dozens. In addition, the tool checked 23 plugins of WordPress and found *62 zero-day vulnerabilities*. These flaws were reported to the developers, and some of them already 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.

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 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.

2 RELATED WORK

Static analysis tools search for vulnerabilities in the applications usually by processing the source code (e.g., [2, 6, 9, 12, 17, 27, 33]). 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). Pixy [12] was one of the first tools to automate this kind of analysis on PHP applications. Later on, RIPS [6] extended this technique with the ability to process more advanced constructs of PHP (e.g., objects). phpSAFE [9] is a recent solution that does taint analysis to look for flaws in CMS plugins (e.g., WordPress plugins). WAP [17, 18] 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. Other works [42, 43] detect vulnerabilities by processing source code properties represented as graphs. In this paper, we propose an novel approach which, unlike these works, does not involve programming information about bugs, but instead extracts this knowledge from annotated code samples and thus learns to find the vulnerabilities.

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 [1, 14]. Other approaches resort to machine learning to predict if there are vulnerabilities in a program [19, 22, 38], 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 [19], or a combination of code-metric analysis with metadata gathered from application repositories [22]. In particular, PhpMinerI and PhpMinerII predict the presence of vulnerabilities in PHP programs [28–30]. 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 [17, 18]. 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 [44] discovers vulnerabilities by identifying missing checks in C language software. VulDeePecker [15]

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. [25] developed a vulnerability detection tool for C and C++ based on features learning from a dataset and artificial neural network. Scandariato et al. [26] performs text mining to predict vulnerable software components in Android applications. SuSi [24] employs machine learning to classify sources and sinks in the code of Android API.

This paper extends our previous work [16]. Our approach 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, which is simplistic, DEKANT also does classification but takes into account the place in which code elements appear in the slice. Such form of classification assists on a more accurate and precise detection of bugs.

3 SURFACE VULNERABILITIES

Many classes of security flaws in web applications are caused by improper handling of user inputs. Therefore, 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., `mysql_query` or `echo`) inducing an erroneous action. Below, we introduce the 12 classes of surface vulnerabilities that will be considered in rest of the paper.

SQLI is the class of vulnerabilities with highest risk in the OWASP Top 10 list [39]. 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 3.1. 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.

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

Example 3.2. 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 conditional validation is false (line 3), the value is returned to the user by `echo` (line 6). A script provided as input would be executed in the browser, possibly carrying out some malicious deed.

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 = mysql_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			(c) 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'] : '';	input var	1 - u	TL = {u}; CTL = {}
2 \$a = \$_POST['age'];	input var	1 - a	TL = {u, a}; CTL = {}
3 if (isset(\$a) && preg_match('/[a-zA-Z]+/', \$u) && is_int(\$a))	cond fillchk var contentchk var	0 - - a - u - a	TL = {u, a}; CTL = {u, a}
4 echo '<input type="hidden" name="user" value="'.\$u.'">';	cond ss var	0 - - u	TL = {u, a}; CTL = {u, a}
5 else	cond	0 -	TL = {u, a}; CTL = {}
6 echo \$u . "is an invalid user";	ss var	0 - u	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, 3, 5, 6}) and a slice not vulnerable (lines {1, 2, 3, 4}), with ISL translation.

The other classes are presented briefly. 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). OS command injection (OSCI) lets an attacker to provide commands to be run in a shell of the OS of the web server. Attackers can supply code that is executed by a eval function by exploring PHP command injection (PHPCI) bugs. LDAP injection (LDAPI), like SQLI, is associated to the construction and execution of queries, in this case for the LDAP service. An attacker can read files from the local file system by exploiting directory traversal / path traversal (DT/PT) and source code disclosure (SCD) vulnerabilities. A comment spamming (CS) bug is related to the ranking manipulation of spammers' web sites. Header injection or HTTP response splitting (HI) allows an attacker to manipulate the HTTP response. An attacker can force a web client to use a session ID he defined by exploiting a session fixation (SF) flaw.

4 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 to detect vulnerabilities. Fig. 3 illustrates this procedure.

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. Then, each instruction of a slice is represented into the *Intermediate Slice Language* (ISL) (Section 5). ISL is a categorized

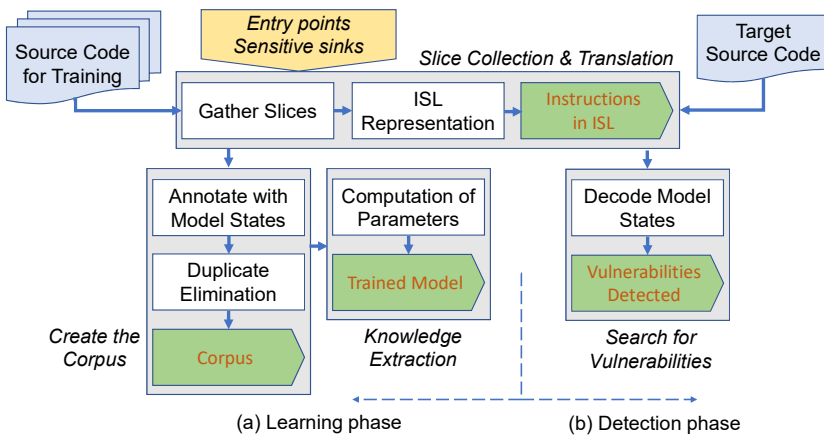


Fig. 3. Overview on the proposed approach.

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 corresponds to a sequence of observations. These observations are classified by the model, tracking the variables from the previous instructions to find out which emission probabilities are selected. 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 (Section 5 and 6). Then, the four above steps are elaborated in Section 7. An overview of the tool that implements our approach is given in Section 8.

5 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.

5.1 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 instructions that could manipulate entry points and be associated to bugs or prevent them (e.g., functions that replace characters in strings). In addition, for PHP functions, we studied cautiously their parameters to determine which of them are crucial for our analysis. In the end, we defined around twenty tokens that are sufficient to describe the instructions of a PHP program.

Example 5.1. 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.

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

Table 1. Intermediate Slice Language tokens.

Token	Description	PHP Function	Taint
input	entry point	\$_GET, \$_POST, \$_COOKIE, \$_REQUEST \$_HTTP_GET_VARS, \$_HTTP_POST_VARS \$_HTTP_COOKIE_VARS, \$_HTTP_REQUEST_VARS \$_FILES, \$_SERVERS	Yes
var	variable	-	No
sanit.f	sanitization function	mysql_escape_string, mysql_real_escape_string mysql_escape_string, mysql_real_escape_string mysql_stmt_bind_param, mysql::escape_string mysql::real_escape_string, mysql_stmt::bind_param	No
ss	sensitive sink	htmlentities, htmlspecialchars, strip_tags, urlencode mysql_query, mysql_unbuffered_query, mysql_db_query mysql_query, mysql_real_query, mysql_master_query mysql_multi_query, mysql_stmt_execute, mysql_execute mysql::query, mysql::multi_query, mysql::real_query mysql_stmt::execute 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, pentl_exec, popen echo, print, printf, die, error, exit file_put_contents, file_get_contents eval is_string, ctype_alpha, ctype_alnum is_int, is_double, is_float, is_integer is_long, is_numeric, is_real, is_scalar, ctype_digit preg_match, preg_match_all, ereg, eregi strnatcmp, strcmp, strncmp, strncasecmp, strcasecmp isset, empty, is_null if implode, join trim, ltrim, rtrim preg_replace, preg_filter, str_ireplace, str_replace ereg_replace, eregi_replace, str_shuffle, chunk_split str_split, preg_split, explode, split, spliti str_pad substr substr_replace	Yes
typechk_str	type checking string function	eval	Yes
typechk_num	type checking numeric function	is_string, ctype_alpha, ctype_alnum is_int, is_double, is_float, is_integer is_long, is_numeric, is_real, is_scalar, ctype_digit	No
contentchk	content checking function	preg_match, preg_match_all, ereg, eregi strnatcmp, strcmp, strncmp, strncasecmp, strcasecmp isset, empty, is_null	No
fillchk	fill checking function	isset, empty, is_null	Yes
cond	if instruction presence	if	No
join_str	join string function	implode, join	No
erase_str	erase string function	trim, ltrim, rtrim	Yes
replace_str	replace string function	preg_replace, preg_filter, str_ireplace, str_replace ereg_replace, eregi_replace, str_shuffle, chunk_split	No
split_str	split string function	str_split, preg_split, explode, split, spliti	Yes
add_str	add string function	str_pad	Yes/No
sub_str	substring function	substr	Yes/No
sub_str_replace	replace substring function	substr_replace	Yes/No
char5	substring with less than 6 chars	-	No
char6	substring with more than 5 chars	-	Yes
start_where	where the substring starts	-	Yes/No
conc	concatenation operator	-	Yes/No
var_vv	variable tainted	-	Yes
miss	miss value	-	Yes/No

description of the corpus and the implementation of the model. The next two columns explain succinctly the purpose of the token and give a few examples. Column four defines the taintedness status of each token which is used when building the corpus or performing the analysis.

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` act as the amount of characters that are manipulated by functions that extract or replace the contents from a user input; 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 the `if` branches.

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, ISL sets all variables to the default token value `var`. 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 8).

5.2 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 statements (line 2), each one defined by either: a rule that covers various operations like string concatenation (lines 4-11); or an conditional (line 12); or an assignment (line 13). The rules take into consideration the syntax of the functions (in column 3 of the table) in order to convey: a sensitive sink (line 4), the sanitization (line 5), the validation (line 6), the extraction and modification (lines 7-10), and the concatenation (line 11).

```

1 grammar isl {
2 slice-isl : statement+
3 statement :
4 sensitive_sink
5 | sanitization
6 | validation
7 | mod_all
8 | mod_add
9 | mod_sub
10 | mod_rep
11 | concat
12 | cond statement+ cond?
13 | assignment
14 sensitive_sink : ss (param | concat)
15 sanitization : sanit_f param
16 validation : (typechk_str | typechk_num | fillchk | contentchk) param
17 mod_all : (join_str | erase_str | replace_str | split_str) param
18 mod_add : add_str param num_chars param
19 mod_sub : sub_str param num_chars start_where?
20 mod_rep : sub_str_replace param num_chars param start_where?
21 concat : (statement | param) (conc concat)?
22 assignment : (statement | param) attrib_var
23 param : input | var
24 attrib_var : var
25 num_chars : char5 | char6
26 }
```

Listing 1. Grammar rules of ISL.

As we will see in Section 6, 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.

Example 5.2. PHP instruction `$u = $_GET['user'];` is translated to input `var`. The assignment and parameter rules (lines 13, 22 and 23) derive the input token, while the attribution rule produces the `var` token (line 24).

6 THE SEQUENCE MODEL

This section presents the *sequence model* that supports vulnerability detection. It explains the graph that represents the model, identifying the states and the observations that can be emitted.

6.1 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 [3, 13, 32]. The variables are divided in 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. 5).

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 of 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 [37]. 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).

6.2 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. Five states were defined as displayed Table 2. The final state of an instruction in ISL is either vulnerable (Taint)

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

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.

6.3 Graph of the Model

Our HMM consists of the graph in Fig. 4, 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.

An ISL instruction corresponds to a sequence of observations. The sequence can start in any state except Val. However, it can reach the Val state for example due to conditionals that check the input data. In the example of Fig. 2 (b), in line 3, one notices a sequence that initiates with a cond observation that could be emitted by the N-Taint initial state. Then, it would transit to the Val state due to the check that is carried out in the if conditional. When the processing of the sequence completes, the model is always either in the Taint or N-Taint states. Therefore, the final

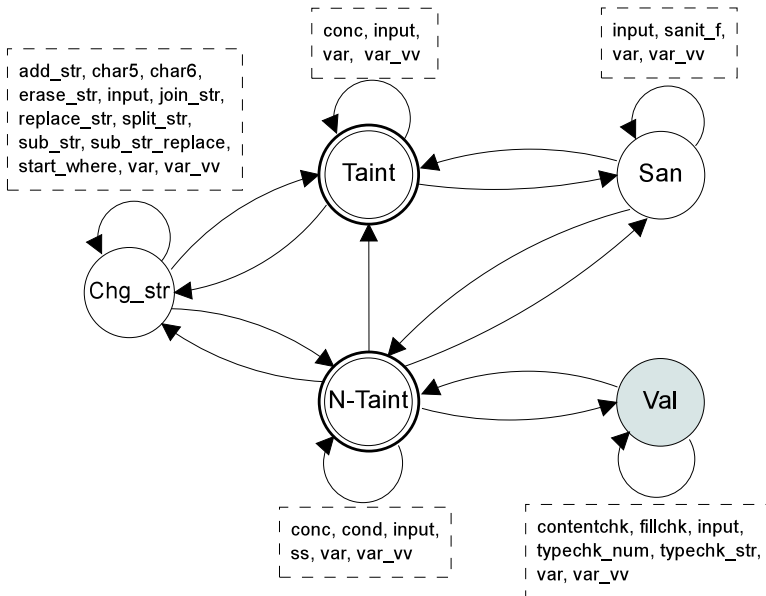
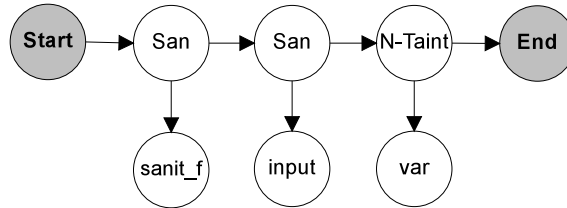


Fig. 4. Model graph of the proposed HMM.

state determines the overall classification of the statement, i.e., if the instruction is vulnerable or not.

Example 6.1. Fig. 5 shows an instantiation of the model for one sequence. The sanitization instruction is translated to the ISL sequence `sanit.f input var`. 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; then, it transits to N-Taint state, emitting the `var` observation (untainted variable).



(a) 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. 5. Graph instantiation for an example sequence.

7 LEARNING AND VULNERABILITY DETECTION

This section explains the main activities related with our approach. The learning phase encompasses a number of activities that culminate with the computation of the parameters of the HMM model. Following that, the vulnerabilities are found by processing the slices of the target application through model in the detection phase. Fig. 3 illustrates the fundamental steps.

7.1 Slice Extraction and Translation Process

The slice extractor analyses files with the source code, gathering the slices that start with an entry point and eventually reach some security-sensitive sink. The instructions between these points are those that implement the application logic based on the user input data. The slice extractor performs intra- and inter-procedural analysis, as it tracks the inputs and its dependencies along the program, walking through the invoked functions. The analysis is context-sensitive as it takes into account the results of function calls.

A translation process occurs when the instructions are collected and consists in representing them as ISL tokens. 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 update a data structure called *variable map* while the slice is translated. 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 7.1. 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. 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 7.2. Fig. 2 has a slightly more complex code snippet. The slice extractor takes from the code two slices: lines `{1, 2, 3, 4}` and `{1, 3, 5, 6}`. The first prevents an attack with a form of input validation, but the second is vulnerable to XSS. The corresponding ISL and variable map are shown in the middle columns. The interesting cases are in lines 3 and 4, which are the `if` statement and its true branch. Both 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. In addition, the sequence model will understand that variables from the former may influence those that appear in latter instructions.

7.2 Process of Creating 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 (not 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 corpus is built in 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, 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 module), but the annotation of the tokens is done manually (as in all supervised machine learning approaches).

Example 7.3. Instruction `$var = $_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.

As mentioned in the previous section, 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 7.4. Instruction `$v = htmlentities($_GET['user'])` is translated to `sanit_f input var` and 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 `htmlspecialchars` sanitizes its parameter; the last token is labeled with the N-Taint state, meaning that the ultimate state of the sequence is not tainted.

```

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 lecho $var
10 include($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.

Example 7.5. Listing 2 displays fourteen PHP instructions collected from vulnerable and non-vulnerable slices. The representation of the instructions into ISL is illustrated in Listing 3. It is possible to observe that some instructions may have more than one representation, depending if the extracted slice is vulnerable or not. For example, the instruction 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, it is also visible the difference between the `var` and `var_vv` tokens. 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 above. 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).

7.3 Configuring the HMM

The sequence model was mostly defined in Section 6. The only missing piece of information are the *parameters*, i.e., the various probabilities to arrive to the start-states, to do the state transitions, and to perform the emissions of the observations. 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:

Start-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 7.6. To obtain the start-state probability of the San state, we count how many sequences begin with the San state and divide by the size of the *corpus*.

```

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_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.

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 7.7. 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 (token, state) appears in the corpus. Then, calculate the emission probability by dividing this count by the total number of pairs (token, state) 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.

```

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.

Example 7.8. To obtain the probability that the Taint state emits the `var_vv` token (`(var_vv, Taint)`), first get the number of occurrences of this pair in the corpus, and next divided 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 [13] 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.

7.4 Detecting Vulnerabilities

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).

Example 7.9. Fig. 2 has the PHP code for the two slices composed of lines {1, 2, 3, 4} and {1, 3, 5, 6} respectively. After processing the first slice, 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 is N-Taint because variable *u* is included in CTL. In the other slice, TL = {u, a} and CTL = { } since there is no validation and the final state is Taint.

In our implementation, the Viterbi algorithm was extended to explore the information kept in the variable map and in these artifacts (further details in Section 8.1.1). 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 start-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; 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, 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 7.10. Fig. 1(a) and (b) shows an example of the detection of a bug. It comprises from left to right: the PHP code, the representation in ISL, the variable map, and the TL after observations are classified. In line 1, the Viterbi algorithm is applied and as 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. 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(c) 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.

8 IMPLEMENTATION AND INITIAL ASSESSMENT

Our approach is implemented in the DEKANT tool. A corpus was also created to train the model. This corpus can be extended in the future with additional annotated sequences, allowing the tool to evolve its knowledge and detection capabilities.

8.1 Implementation of DEKANT

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

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 $\langle \text{token}, \text{state} \rangle$ 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 $\langle \text{miss}, \text{Taint} \rangle$ or $\langle \text{miss}, \text{N-Taint} \rangle$) to ensure that all sequences have the same length; (ii) then, the various probabilities of the model are computed as explained in the previous section; (iii) lastly, all relevant information about the model is saved in a plain text file to be loaded by the vulnerability detector module.

Slice collector: uses a lexer and a parser to process PHP code (based on ANTLR²). 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.

²<https://wwwantlr.org/>

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.

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.

8.1.1 Extensions to the Viterbi algorithm. We extended the Viterbi algorithm with the two procedures of Section 7.4 (`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 and SL).

Listing 5 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, it is verified the `obs` position 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 30); (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 32 to 48); (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 50 to 61); (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 63 to 71). 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 72).

Afterwards, the traditional Viterbi algorithm is executed (`decodeVit` step as explained in Section 7.4) and then the post-processing `afterVit` procedure runs.

```

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 statements

```

```

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 5. *beforeVit* extension to the Viterbi algorithm.

Listing 6 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 ** SL - sanitized list
5 ** state - state of the last obs from inst_slice_isl
6 ** value - assignment 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 6. *afterVit* extension to the Viterbi algorithm.

8.2 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 [13]. 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 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 applications³ and had flaws from the twelve classes presented in Section 3. The probability matrices that were computed based on this corpus are shown in Fig. 6.

³bayar, 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, phpbtrkplus-2.2, SAMATE, superlinks, vicnum15, ZiPEC 0.32, Wordpress 3.9.1.

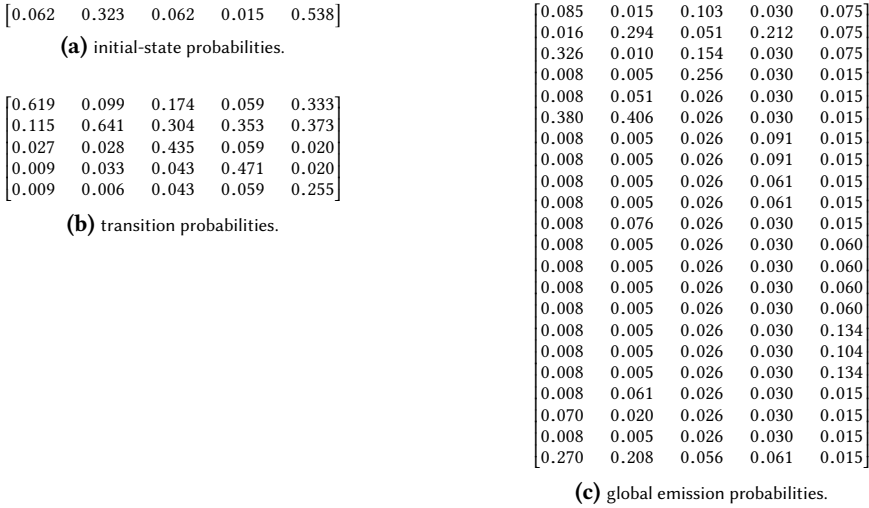


Fig. 6. Parameters of our HMM model extracted from the corpus. Columns correspond to the 5 states (in the same order of column 1 of Table 2). The lines of matrix (c) are the tokens (in the same order of column 1 of Table 1).

To perform a preliminary assessment the model, we applied a *10-fold cross validation* [8]. 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 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 alarms, 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 by DEKANT in the first two phases of the method. For example, the first row says that DEKANT issued 419 alerts in the evaluation phase but that 14 of them were mistakes (columns 2 and 3). In the evaluation phase, the precision and accuracy are very high, around 0.97 and 0.95, and the rates are small (fpr is 0.15 and fnr is 0.02). In the decoding phase, the results are 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 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 DEKANT, something that we will be check more thoroughly in the next section.

9 EXPERIMENTAL EVALUATION

Our experimental evaluation addresses the following questions about DEKANT: (1) Is the tool able to discover novel vulnerabilities? (Section 9.1); (2) Can it classify correctly various classes of vulnerabilities? (Section 9.1); (3) Is DEKANT more accurate and precise than tools that search for

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

vulnerabilities in plugins (Section 9.2); tools that do data mining using standard classifiers (Section 9.3); and, tools that do taint analysis (Section 9.4)?

9.1 Open Source Software Evaluation

This section assesses the ability of DEKANT to classify different vulnerabilities by analyzing 23 WordPress plugins [41] and 23 packages of real web applications. All of these are written in the PHP language. The plugins are used to determine if the tool is useful for the discovery of new (zero-day) vulnerabilities. The applications serve as a ground truth for the evaluation, since they have known vulnerabilities — 13 of the packages contain bugs found by [18] and the other 10 packages have flaws disclosed by various researchers in the past. 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.

9.1.1 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. We selected a diverse set of plugins based on two criteria, the development team and the number of

Table 4. Vulnerable slices in plugins found by DEKANT.

WordPress Plugin	Slices	Real Vulnerabilities						FP
		SQI	XSS	Files	SCD	HI	CS	
Appointment Booking Calendar	15	3	4					
Login by Auth0	1		1					
Authorizer	2		2					
BuddyPress	4							
Contact formgenerator	14	11						
CP Appointment Calendar	11	2						
Easy2map	13		1	2				
Ecwid Shopping Cart	1		1					
Gantry Framework	3		3					
Google Maps Travel Route	10	1	2					1
Lightbox Plus Colorbox	8		8					
Payment form for Paypal pro	19		2					
Recipes writer	8		4					
ResAds	17		17					
Simple support ticket system	37	18						
The Cart Press eCommerce Shopping	25	8	17					
WebKite	1	1						
WP Easy Cart eCommerce Shopping	78	13	6	29	5	5	2	
WP Marketplace	45	2	24					3
WP Shop	22	7	10					
WP ToolBar Removal Node	1		1					
WP ultimate recipe	7							1
WP Web Scraper	3		3					
Total	345	66	106	31	5	5	2	5

DT & RFI, LFI vulnerabilities

9.1.2 Real Web Applications. To determine if DEKANT is effective at classifying the vulnerabilities belonging to the twelve classes under study, we run the tool with 23 well known vulnerable open source software packages divided into two sets.

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 twelve 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.

Summing-up, the results are reassuring as DEKANT correctly classifies every vulnerability that was described in [18], but actually with less FP. The accuracy and precision are very high, around 0.97, and the FP rate is 0.06 and the FN rate is 0.01.

For the second set, we run DEKANT with ten applications with flaws previously registered in the CVE [5] and NVD [21] databases (Table 6). In total more than 4,200 files and 1.5 million LoC are analyzed. The largest packages are *epesi* and *phpMyAdmin*, with approximately 750 and 250 thousand LoC. Similarly to the first set of applications, we extracted 310 slices, which were then checked manually.

DEKANT classifies 223 slices as having bugs but 12 alarms are invalid (columns 5-6). The vulnerabilities pertain to six classes, where the most common are SQLI and XSS (columns 7-10, with Files aggregating DT, RFI and LFI). The FPs occur in the XSS and PHPCI classes due to equivalent reasons as above. The remaining 87 slices are correctly set as not-vulnerable (not shown in the table). Consequently, we could not find missed bugs (i.e., FN is zero).

Overall, DEKANT had accuracy and precision of 0.96 and 0.95, and a FP rate of 0.12 (and no FNs). 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.

Table 6. Slices in open source software with vulnerabilities disclosed in the past, analyzed by DEKANT.

Web application	Files	LoC	Time (s)	Classif.		Vulnerability classes			
				Vul	FP	SQLI	Files*	CI**	XSS
cacti-0.8.8b	249	95274	7	2	2			1	1
communityEdition	228	217195	21	16		4	4	3	5
epesi-1.6.0	2246	741440	90	25	4		3		22
NeoBill0.9-alpha	620	100139	5	19			2		17
phpMyAdmin-4.2.6	538	241505	12	1					1
refbase-0.9.6	171	109600	8	5	6				5
Schoolmate-1.5.4	64	8411	2	120		69			51
VideosTube	39	3458	2	1					1
Webchess 1.0	37	7704	2	20		6			14
Zero-CMS.1.0	21	1139	2	2		1			1
Total	4213	1525865	151	211	12	80	9	4	118

*DT & RFI, LFI vulnerabilities

**PHPCI vulnerability

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	2955	3	4			1	3	1		3	3	4	2	14	
Login by Auth0	1.3.6	35	3101		1				1					1			
Authorizer	2.3.6	164	159023		2				2					1			1
BuddyPress	2.4.0	574	219690							1			-	-	-	-	-
Contact formgenerator	2.0.1	42	9187	11				11							3		11
CP Appointment Calendar	1.1.7	7	988	2				2					2		9		
Easy2map	1.2.9	16	3193		1				1					1	8	10	
Ecwid Shopping Cart	3.4.6	61	16807		1				1				-	-	-	-	1
Gantry Framework	4.1.6	274	50717		3				1			2		1			2
Google Maps Travel Route	1.3.1	10	1692	1	2	1		1	2					1	7	10	2
Lightbox Plus Colorbox	2.7.2	13	5902		8				6			2	-	-	-	-	8
Payment form for Paypal pro	1.0.1	10	3920		2				2					2	19	2	
Recipes writer	1.0.4	9	2074		4				4					4	5		
ResAds	1.0.1	30	3168		17				2			15		17			
Simple support ticket system	1.2	20	1533	18				18					3		2	7	15
The Cart Press eCommerce Shopping	1.4.7	220	47114	8	17			8	17				-	-	-	-	25
WebKite	2.0.1	13	1267	1				1									1
WP Easy Cart eCommerce Shopping	3.2.3	623	126448	13	6			13	6				-	-	-	-	19
WP Marketplace	2.4.1	88	15485	2	24	3	3		9		1	20	2	27	18	30	
WP Shop	3.5.3	49	9171	7	10				5	1		12	7	10	5	29	
WP ToolBar Removal Node	1839	2	544		1				1					1			
WP ultimate recipe	2.5	284	42774			1	1		1		1				6		1
WP Web Scraper	3.5	89	8116		3				3				-	-	-	-	3
Total		2639	734869	66	106	5	4	55	67	3	2	54	17	70	84	102	89

9.2 Comparison with Plugin Analysis Tools

The section tests plugin analysis tools, namely WAPe [18] and phpSAFE [20], 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 23 plugins previously presented, which have a total of 349 slices (the 345 slices of Section 9.1.1 plus 4 extra slices that were extracted by the other two tools). The results are summarized in Table 7.

DEKANT evaluates 345 slices (columns 5-8) and outputs 177 of them as potentially vulnerable to SQLi and XSS. Out of this group, 172 of them have real bugs and 5 are FPs. The remaining 168 slices are correctly classified as not vulnerable. While processing the results, we observed that: (i) there are four 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 code that end up being inspected.

WAPe discovers 122 bugs but misses 54 (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, three cases are deemed FPP, leaving only two FPs. In the case of DEKANT, these five slices are placed in the non-vulnerable set. WAPe and DEKANT extract 126 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 17 plugins (out of 23) and three of them partially (columns 14 to 18). For this reason, only 234 slices out of 349 are examined. Within the group of analyzed slices,

there are 87 vulnerabilities that are found and 33 that are missed. However, phpSAFE finds three errors that no other tool is able to discover. The 84 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* and *prepare* from WordPress (the last one protects a SQL statement from SQLi attacks, providing similar functionality as prepared statements).

phpSAFE scans 102 extra slices (aside from the 349 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, RIPS, and Pixy for the detection of SQLi and XSS.

Metric	Plugins			WebApps – Data mining			WebApps – Taint analysis		
	DEKANT	WAPe	phpSAFE	DEKANT	WAPe	PhpMiner II	DEKANT	RIPS	Pixy
acc	0.97	0.84	0.50	0.97	0.96	0.83	0.97	0.80	0.54
pr	0.97	0.98	0.51	0.98	0.96	0.57	0.98	0.43	0.23
fpr	0.03	0.01	0.49	0.004	0.01	0.04	0.004	0.09	0.48
fnr	0.02	0.31	0.51	0.14	0.15	0.74	0.14	0.69	0.37

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

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 102 PFPs of phpSAFE are disregarded from the calculations.

Table 9. Comparison of results between DEKANT, WAPe, PhpMinerII, RIPS and Pixy with open source projects.

Web application	DEKANT				WAPe				PhpMinerII				RIPS				Pixy				
	SQLi	XSS	oth	FP	FN	SQLi	XSS	oth	FPP	FP	FN	SQLi	XSS	oth	FP	FN	SQLi	XSS	FP	FN	
Admin Control Panel Lite 2	9	72		1	9	72		8	1	9	23	1	49	9	7	7	65	9	67	12	5
Clip Bucket		10	12	3	9		10	12	2	4	9		19	20	-	-	-	31		19	47
Clip Bucket	4	10	12	3	9	4	10	12	2	4	9	3	19	17	1	-	-	35	3	19	47
Ldap address book		39		1		3		36		1			2	6			39		1	4	5
Minutes		9		1		9		6		1			12		5	7	11		7	3	10
Mle Moodle		5	1	14		5	1	3		14		10	27	8		6	2	7	12		18
Php Open Chat		10	1	7		9	1			8		9	7	8		17		43	1		2
Pivotx		1	3	10		1	3	9		10		4	1	6		6	4	7	4		4
Play sms		5		7		5	2			7		10	12			6	2	31	4		10
RCR AESir		9	4			9	4	1				3		6		8	4	2	1		2
SAE		61	65	22	5		61	65	20		10	2		8	2	118		2	5		141
Tomahawk Mail		2	1			2	1			3		1	1	2		2		6	1		1
vfront		32	68	34	2	11	32	68	34	24	2	11		1		105		74	39	114	32
Total		117	295	91	14	79	114	291	89	54	23	88	12	112	95	353	9	136	63	232	374
																				12	289
																				1031	181

9.3 Comparison with Data Mining Tools

A few other tools have implemented data mining mechanisms for tasks related with bug discovery, namely WAPe and PhpMinerII [28, 29]. 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. Similarly, the same limitation applies to the vulnerability detection tools that will be studied in the next section, and so we will focus on these applications for the rest of the evaluations.

We observed that the various tools (from this and next 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 2609 slices. This set was then manually examined to determine which slices are vulnerable, and it serves as a ground truth. Overall there are 582 slices with vulnerabilities (117 SQLI, 360 XSS, and 105 others) and 2027 slices without problems. This second group was divided in a few subsets, namely, slices with sanitized input, slices with validated or modified input, and 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.

9.3.1 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 75 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 better than WAPe, namely because it shows superior FP and FN rates.

9.3.2 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

Table 10. Confusion matrix of DEKANT, WAPe and RIPS for the detection of all vulnerability classes.

Predicted	Observed						Metric	DEKANT	WAPe	RIPS
	DEKANT		WAPe		RIPS					
	Vul	N-Vul	Vul	N-Vul	Vul	N-Vul				
Vul	503	14	494	23	208	232	acc	0.96	0.96	0.77
N-Vul	79	2013	88	2004	374	1795	pr	0.97	0.96	0.47
							fpr	0.007	0.01	0.11
							fnr	0.13	0.15	0.64

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

had to build a dataset for that purpose. The training dataset was constructed by recreating the procedure explained in [28, 29], where the WEKA package implemented the data mining tasks [40]. 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 353 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 65 slices with bugs (out of the 79 shown in table). WAPe classifies 405 vulnerabilities, but with 16 FPs (of the 23 presented in table) and 72 FNs (out of the 88). 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 0.97 and 0.98 on the metrics. In addition, DEKANT has better rates for false positives and false negatives.

9.4 Comparison with Taint Analysis Tools

There have been tools proposed in the past that perform taint analysis to locate vulnerabilities, and two notable examples are RIPS [6] and Pixy [12]. They track data arriving at the entry points to determine if it reaches a sensitive sink, taking sanitization operations in consideration. RIPS detects the same classes of vulnerabilities as DEKANT, but Pixy only looks for SQLI and reflected XSS. Our evaluation compares the three tools while processing the same applications of the previous section (i.e., the dataset with 2609 slices), with results being displayed in Table 9.

9.4.1 All Vulnerability Classes. The RIPS tool only outputs information about slices that are regarded as vulnerable. Therefore, when no result appears for a particular slice, this could occur because the slice was considered valid or due to the inability to extract the slice. Since we are unable to separate the two situations, this brings some level of uncertainty to the analysis.

RIPS generates alerts for a total of 440 slices in 11 applications (of the 13). Out of this group, 208 correspond to slices with real bugs and the remaining 232 to false alerts. These FPs occur essentially in slices with functions that change the data received at the entry points (such as, *substr* and *preg_replace*) or in slices with validation functions. 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). RIPS does not catch 374 vulnerabilities from the ground truth. We speculate that the reason for this high number of FNs is probably related to the extractor being unable to gather many slices.

Table 10 has the confusion matrix and the metrics. RIPS is outperformed by both DEKANT and WAPe in the dataset with all classes of flaws. Its accuracy and precision are 0.77 and 0.47, which are not as high as the other tools.

9.4.2 Just SQLI and XSS. Pixy only searches for SQLI and XSS vulnerabilities. Therefore, our evaluation just covers these two classes (meaning that the 105 slices with *other* vulnerabilities are treated as being true negatives).

Pixy results are displayed in the last four columns of Table 9. The tool raises 1332 alerts, but the majority of them are mistakes⁵. Only 301 reported vulnerabilities are real (12 SQLI and 289 XSS). There are 176 undetected bugs. Curiously, Pixy has around half the FNs of RIPS, and for some applications it is the tool that detects more XSS vulnerabilities (e.g., the *Minutes* application) but at the cost of a high FP rate. Section 9.3.2 presents the details about the DEKANT evaluation for SQLI and XSS bugs. In what concerns RIPS, the tool finds 145 buggy slices but misses 328 (out of the 374 in the table). It also wrongly reports 192 slices as being vulnerable (of the 232 in the table).

Table 8 presents the metrics for these tools. The results corroborate the promising detection capabilities of DEKANT, as the tool has the best accuracy and precision and the lowest FP and FN rates. RIPS is second, but with an accuracy and precision reasonably below. For our dataset, the weakest values are obtained by Pixy, with a small precision due to the many FP.

10 CONCLUSION

The paper explores a new approach to detect web application vulnerabilities inspired in 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 can be used as a static analysis tool to discover vulnerabilities in source code and identify their location.

ACKNOWLEDGMENTS

This work was partially supported by the national funds through Fundação para a Ciência e a Tecnologia (FCT)/MCTES (PIDDAC)/FEDER with reference to project AAC-2/SAICT/2017-029058 (SEAL), and through FCT with references UID/CEC/00408/2019 (LASIGE) and UID/CEC/50021/2019 (INESC-ID).

REFERENCES

- [1] Erik Arisholm, Lionel C Briand, and Eivind B Johannessen. 2010. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software* 83, 1 (2010), 2–17.
- [2] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *EuroS&P*. 334–349.
- [3] Leonard E. Baum and Ted Petrie. 1966. Statistical Inference for Probabilistic Functions of Finite State Markov Chains. *The Annals of Mathematical Statistics* 37, 6 (1966), 1554–1563.
- [4] BBC Technology. 2014. Millions of websites hit by Drupal hack attack. <http://www.bbc.com/news/technology-29846539>.
- [5] CVE. [n. d.]. <http://cve.mitre.org>.
- [6] Johannes Dahse and Thorsten Holz. 2014. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *Proceedings of the 21st Network and Distributed System Security Symposium*.
- [7] Johannes Dahse and Thorsten Holz. 2015. Experience Report: An Empirical Study of PHP Security Mechanism Usage. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 60–70.
- [8] Janez Demšar. 2006. Statistical Comparisons of Classifiers over Multiple Data Sets. *The Journal of Machine Learning Research* 7 (Dec 2006), 1–30.
- [9] José Fonseca and Marco Vieira. 2014. A Practical Experience on the Impact of Plugins in Web Security. In *Proceedings of the 33rd IEEE Symposium on Reliable Distributed Systems*. 21–30.
- [10] HELPNETSECURITY. 2017. Hacker breached 60+ unis, govt agencies via SQL injection. <https://www.helpnetsecurity.com/2017/02/16/hacker-govt-agencies-via-sql-injection/>.

⁵In fact, a significant number of non-vulnerable slices included in our ground truth dataset comes from Pixy.

- [11] Imperva. 2017. The State of Web Application Vulnerabilities in 2017. (Dec. 2017).
- [12] N. Jovanovic, C. Kruegel, and E. Kirda. 2006. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*. 27–36.
- [13] Daniel Jurafsky and James H. Martin. 2008. *Speech and Language Processing*. Prentice Hall.
- [14] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering* 34, 4 (2008), 485–496.
- [15] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *Annual Network and Distributed System Security Symposium*.
- [16] Ibéria Medeiros, Nuno F. Neves, and Miguel Correia. 2016. DEKANT: a static analysis tool that learns to detect web application vulnerabilities. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*.
- [17] Ibéria Medeiros, Nuno F. Neves, and Miguel Correia. 2016. Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining. *IEEE Transactions on Reliability* 65, 1 (March 2016), 54–69.
- [18] Ibéria Medeiros, Nuno F. Neves, and Miguel Correia. 2016. Equipping WAP with Weapons to Detect Vulnerabilities. In *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.
- [19] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. 2007. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*. 529–540.
- [20] Paulo Nunes, José Fonseca, and Marco Vieira. 2015. phpSAFE: A Security Analysis Tool for OOP Web Application Plugins. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.
- [21] NVD. [n. d.]. <http://nvd.nist.org>.
- [22] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. 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 (CCS '15)*. 426–437.
- [23] Lawrence R Rabiner. 1989. A tutorial on hidden Markov models and selected applications in speech recognition. *Proc. IEEE* 77, 2 (1989), 257–286.
- [24] S. Rasthofer, S. Arzt, and E. Bodden. 2014. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS)*.
- [25] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In *Proceedings of the International Conference on Machine Learning and Application (ICMLA)*.
- [26] R. Scandariato, J. Walden, A. Hovsepian, and W. Joosen. 2014. Predicting Vulnerable Software Components via Text Mining. *IEEE Transactions on Software Engineering* 40, 10 (2014), 993–1006.
- [27] Umesh Shankar, Kunal Talwar, Jeffrey S Foster, and David Wagner. 2001. Detecting format-string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*.
- [28] Lwin Khin Shar and Hee Beng Kuan Tan. 2012. Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities. In *Proceedings of the 34th International Conference on Software Engineering*. 1293–1296.
- [29] Lwin Khin Shar and Hee Beng Kuan Tan. 2012. 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*. 310–313.
- [30] Lwin Khin Shar, Hee Beng Kuan Tan, and Lionel C. Briand. 2013. Mining SQL Injection and Cross Site Scripting Vulnerabilities using Hybrid Program Analysis. In *Proceedings of the 35th International Conference on Software Engineering*. 642–651.
- [31] Sink. 2017. XSS Attacks: The Next Wave. <https://snyk.io/blog/xss-attacks-the-next-wave/>.
- [32] Noah A. Smith. 2011. *Linguistic Structure Prediction*. Graeme Hirst.
- [33] Soeul Son and Vitaly Shmatikov. 2011. SAFERPHP: Finding Semantic Vulnerabilities in PHP Applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*.
- [34] The Hacker News. 2017. It's 3 Billion! Yes, Every Single Yahoo Account Was Hacked In 2013 Data Breach. <https://thehackernews.com/2017/10/yahoo-email-hacked.html>.
- [35] The Hacker News. 2017. WordPress Plugin Used by 300,000+ Sites Found Vulnerable to SQL Injection Attack. <https://thehackernews.com/2017/06/wordpress-hacking-sql-injection.html>.
- [36] threatpost. 2017. Million-Plus WordPress Sites Exposed by Vulnerable Plugin. <https://threatpost.com/million-plus-wordpress-sites-exposed-by-vulnerable-plugin/123983/>.
- [37] A. Viterbi. 1967. Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm. *IEEE Transactions on Information Theory* 13, 2 (April 1967), 260–269.

- [38] James Walden, Maureen Doyle, Grant A Welch, and Michael Whelan. 2009. Security of open source web applications. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*. 545–553.
- [39] J. Williams and D. Wichers. 2017. OWASP Top 10 2017 – The Ten Most Critical Web Application Security Risks.
- [40] Ian H. Witten, Eibe Frank, and Mark A. Hall. 2011. *Data Mining: Practical Machine Learning Tools and Techniques* (3rd ed.). Morgan Kaufmann.
- [41] WordPress. [n. d.]. <https://wordpress.org/>.
- [42] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. 590–604.
- [43] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. 2015. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. 797–812.
- [44] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck. 2013. Chucky: Exposing Missing Checks in Source Code for Vulnerability Discovery. In *Proceedings of the 20th ACM SIGSAC Conference on Computer Communications Security*. 499–510.

Ibéria Medeiros is an Assistant Professor in the Department of Informatics, at the Faculty of Sciences of University of Lisbon. She is a member of the Large-Scale Informatics Systems (LASIGE) Laboratory, and the Navigators research group. She holds a PhD in Computer Science by the Faculty of Sciences of University of Lisbon. Currently, she is the principal investigator of the SEAL national project, has been participating in DiSIEM European project and REDBOOK national project, and participate in SEGRID European project. She is author of tools for software security, which WAP (Web Application Protection) is the most known and an OWASP project. Her research interests are concerned with software security, source code static analysis, vulnerability detection, data mining and machine learning, and security. 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 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 national and EU 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 on the editorial board of the International Journal of Critical Computer-Based Systems. More information about him can be found at <http://www.di.fc.ul.pt/~nuno/>.

Miguel Correia is an Associate Professor with Habilitation 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 SPARTA, QualiChain, Safe-Cloud, PCAS, TLOUDS, ReSIST, CRUTIAL, and MAFTIA European projects. He has more than 150 publications and is Senior Member of the IEEE. More information about him at <http://www.gsd.inesc-id.pt/~mpc/>.