DEKANT: A Static Analysis Tool that Learns to Detect Web Application Vulnerabilities

Ibéria Medeiros
INESC-ID, Faculdade de Ciências, Universidade de Lisboa – Portugal
ibemed@gmail.com

Nuno Neves
LaSIGE, Faculdade de Ciências, Universidade de Lisboa – Portugal
nuno@di.fc.ul.pt

Miguel Correia
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa – Portugal
miguel.p.correia@ist.utl.pt

ABSTRACT

The state of web security remains troubling as web applications continue to be favorite targets of hackers. Static analysis tools are important mechanisms for programmers to deal with this problem as they search for vulnerabilities automatically in the application source code, allowing programmers to remove them. However, developing these tools requires explicitly coding knowledge about how to discover each kind of vulnerability. This paper presents a new approach in which static analysis tools *learn* to detect vulnerabilities automatically using machine learning. The approach uses a sequence model to learn to characterize vulnerabilities based on a set of annotated source code slices. This model takes into consideration the order in which the code elements appear and are executed in the slices. The model created can then be used as a static analysis tool to discover and identify vulnerabilities in source code. The approach was implemented in the DEKANT tool and evaluated experimentally with a set of open source PHP applications and WordPress plugins, finding 16 zero-day vulnerabilities.

CCS Concepts

•Software and its engineering → Software verification and validation; •Security and privacy → Vulnerability management; Web application security; •Computing methodologies → Machine learning;

Keywords

vulnerabilities, web application, software security, static analysis, sequence models, machine learning

1. INTRODUCTION

The state of web application security continues to be a concern. In the OWASP Top 10 of 2013, vulnerabilities such as SQL injection (SQLI) and cross-site scripting (XSS) maintain a high risk level [32]. Moreover, specific vulnerabilities continue to cause major problems, with allegedly 12 million sites compromised in Oct. 2014 due to an SQLI vulnerability in Drupal [2] and data of 37 million users stolen in Aug. 2015 from the Ashley Madison site using an SQLI attack [29].

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 certain code *slice* (a series of instructions). Therefore, a good practice for web application security is to pass inputs through *sanitization functions* that invalidate dangerous metacharacters or *validation functions* that evaluate their content.

Programmers often use *static analysis tools* to search for vulnerabilities automatically in the application source code, then removing them. However, developing these tools requires explicitly coding knowledge about how each vulnerability is detected [5, 8, 10, 14], which is complex. Moreover, this knowledge may be wrong or incomplete, making the tools inaccurate [6]. For example, if the tools do not understand that a certain function sanitizes inputs, this could lead to a false positive (a warning about an inexistent vulnerability).

This paper presents a new approach for static analysis, leveraging classification models for sequences of observations that are commonly used in the field of natural language processing (NLP). Currently, NLP tasks such as parts-of-speech tagging or named entity recognition are typically modeled 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 estimates given by a structured prediction model that takes word order into consideration. The model's parameters (e.g., symbol emission and class transition probabilities, in the case of hidden Markov models) are typically inferred using supervised machine learning techniques, leveraging annotated corpora. We propose applying the same approach to programming languages. These languages are artifical but they have many characteristics in common with natural languages, such as the existence of words, sentences, a grammar, and syntactic rules. 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 this order. Therefore it involves forms of classification more sophisticated than classification based on *standard classifiers* (e.g., naive Bayes, decision trees, support vector machines) that simply verify the presence of certain observations, without considering any order and relation between them.

ISSTA '16, July 18-22, 2016, Saarbrücken, Germany
© 2016 ACM. ISBN 978-1-4503-4390-9/16/07...$15.00
DOI: http://dx.doi.org/10.1145/2931037.2931041

This paper is the first to propose an approach in which *static analysis tools learn to detect vulnerabilities automatically using machine learning.* The approach involves using 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 used sequence models. PHPMinerII uses machine learning to train standard classifiers, which are then used to verify if certain code elements exist in the code, but not to identify the location of the vulnerabilities [24, 25]. WAP uses a taint analyzer (with no machine learning involved) to search for vulnerabilities and a standard classifier to classify them as true or false positives [14]. Neither of the two tools considers the order of code elements or the relation between them, leading to false positives and false negatives.

We specifically use a *hidden Markov model* (HMM) [20] to characterize vulnerabilities based on a set of source code slices with their *code elements* (e.g., function calls) annotated as tainted or not, taking into consideration the code that validates, sanitizes, and modifies inputs. The model can then be used as a static analysis tool to discover vulnerabilities in source code. A HMM is a Bayesian network composed of nodes representing states and edges representing transitions between states. In a HMM the states are hidden, i.e., are not observed. Given a sequence of observations, the hidden states (one per observation) are discovered following the HMM, taking into account the order of the observations. The HMM can be used to find the sequence of states that *best* explains the sequence of observations (of code elements, in our case). To detect vulnerabilities we introduce the idea of revealing the discovered hidden states of the code elements that compose the slice. This is interesting because the state of the elements determines if they are *tainted*, i.e., if the state may have been defined by an input, which may have been provided by an adversary. This allows the tool to interpret the execution of the slice statically, i.e., without actually running it. Notice that transitioning from a state to another requires understanding how the code elements behave in terms of sanitization, validation and modification, or if they affect the data flow somehow. This understanding is performed by the machine learning algorithm we propose.

The paper also presents the *hidDEn marKov model diAgNosing vulnerabiliTies* (DEKANT) tool that implements our approach. DEKANT first extracts slices from the source code, next translates these slices into an intermediate language – *intermediate slice language* (ISL) – and retrieves their variable map. Then it analyses that representation, with the assistance of its variable map, to understand if there are vulnerabilities or not. Finally, the tool outputs the vulnerabilities, identifying them in the source code.

We evaluated the tool experimentally with 10 plugins of the WordPress content management system (CMS) [34] and it discovered *16 zero-day vulnerabilities* (i.e., 16 previously-unknown vulnerabilities). These vulnerabilities were reported to the developers of the plugins that confirmed their existence and fixed the plugins. Also, we ran the tool with 10 publicly-available open source web applications written in PHP with vulnerabilities disclosed in the past, adding up to more than 4,200 files and 1.5 million lines of code. From the 310 slices analyzed, 211 were classified as containing vulnerabilities and 99 as not. The tool found 21 vulnerabilities reported in the Common Vulnerabilities and Exposures (CVE) [4] and the Open Source Vulnerability Database (OSVDB) [18]. Our results suggest that the tool is capable of relating the elements that characterize a vulnerability in order to classify slices correctly as vulnerable or not.

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 detect vulnerabilities using annotated code slices; (2) a sequence model and an intermediate language used by the model to detect vulnerabilities taking into consideration the order in which the code elements appear in the slices; (3) a static analysis tool that implements the approach, learning to detect vulnerabilities using annotated code slices, then using this knowledge to discover and identify vulnerabilities in web application source code; (4) an experimental evaluation that shows the ability of this tool to detect known and zero-day vulnerabilities.

2. RELATED WORK

Static analysis tools search for vulnerabilities in the code of applications, typically in source code [8, 9, 10, 17, 23, 28, 35]. Many of these tools do taint analysis, i.e., track user inputs to verify if they reach a sensitive sink (a function that can be exploited). Pixy [10] is one of the first tools of the kind for PHP code. phpSAFE [17, 8] is a recent tool that does taint analysis to search for vulnerabilities in CMS plugins (e.g., WordPress plugins) without analysing the CMS source code, so it is configurable with the functions of the CMS that work as entry points and sensitive sinks. Static analysis tools tend to generate many false positives and false negatives due to the complexity of coding knowledge about vulnerabilities. WAP [14, 15] also does taint analysis, but uses data mining to predict false positives, besides also going one step further and correcting automatically the detected vulnerabilities. Yamaguchi et al. [35] presented an approach to do more precise static analysis based on a novel data structure to represent source code (code property graph). We propose an alternative approach that unlike these works does not involve coding knowledge about vulnerabilities, instead extracts this knowledge from annotated code samples.

Machine learning has been used in some works to measure software quality by collecting attributes that reveal the presence of software defects [1, 3, 12]. Other works use machine learning to predict the existence of vulnerabilities in the source code, which is different from identifying precisely their existence, which is what we do in this paper [16, 27, 31, 19]. They use attributes such as past vulnerabilities and function calls [16], or code complexity and developer activities [27], or a combination of code-metric analysis with metadata gathered from code repositories [19].

PhpMinerI and PhpMinerII use machine learning to predict the presence of vulnerabilities in PHP programs [24, 25]. These tools extract a set of attributes from program slices that end in a sensitive sink but do not necessarily start in a entry point. The tools are first trained with a set of annotated slices, then used to assess the presence of vulnerabilities. The tools do not perform directly the data mining process, but instead the user has to use the WEKA tool to do it [33]. Recently, these authors enhanced the detection using traces of program executions [26]. WAP is different because it uses machine learning and data mining to predict if a vulnerability detected by taint analysis is a real vulnerability or a false positive [14, 15]. Furthermore, unlike PhpMiner, it identifies the location of vulnerabilities in the source code, which is required to remove them. The PhpMiner tools and WAP use standard classifiers (e.g., Logistic Regression, Naive Bayes, 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 use machine learning techniques in contexts other than web applications. Chucky discovers vulnerabilities by identifying missing checks in C source code [36]. The tool does taint analysis to identify checks between entry points and sensitive sinks, applies text mining to discover the neighbors of these checks, then builds a model to identify missing checks. Scandariato et al. use text mining to predict vulnerable software components in Android applications [21]. They use text mining techniques to get the terms (words) present in software components (files) and their frequencies, and use a static code analyzer to check if those software components are vulnerable or not. Then, they correlate the term frequencies in vulnerable software components and build a model to predict if a given software component is vulnerable or not.

This paper is the first to explore the use of sequential models, learned from training data, in the context of static analysis for security.

3. SURFACE VULNERABILITIES

The major classes of security flaws in web applications are due to improper handling of user input and may be denominated *surface vulnerabilities* or *input validation vulnerabilities*. Such a vulnerability may be exploited by crafting an input that combines normal characters with metacharacters or metadata (e.g., ', OR). Considering the case of programs in PHP, such input enters the program though an *entry point* like $_POST and reaches a *sensitive sink* like *mysql_query*, *echo* or *include*, exploiting the vulnerability. This section presents the classes of surface vulnerabilities considered in the experimental evaluation of DEKANT: SQLI, XSS, remote and local file inclusion, directory traversal, source code disclosure, operating system and PHP command injection.

SQLI has the highest risk in [32]. The following PHP script has a simple example of a SQLI vulnerability that we found in *SchoolMate 1.5.4 (ValidateLogin.php)* [22]. `$u` takes the username provided by the user (line 1), then 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 obtaining all users' passwords.

```
1 $u = $_POST['username'];
2 $q = "SELECT pass FROM users WHERE user='".$u."'";
3 $query = mysql_query($q);
```

XSS vulnerabilities allow attackers to execute scripts in the users' browsers. There are some varieties of XSS, but we explain only reflected XSS for space reasons. The following code shows an example of this vulnerability that we discovered in *ZeroCMS 1.0 (zero_compose.php)* [37]. If the input is not empty, it is stored in `$user_id` and inserted in the HTML file returned to the user by the *echo* function.

```
$user_id = (isset($_POST['user_id'])) ?
     $_POST['user_id'] : '';
echo '<input type="hidden" name="user_id" value="' .
     $user_id . '">';
```

The other six vulnerabilities are presented briefly. Remote and local file inclusion (RFI/LFI) vulnerabilities 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 several strategies to insert it there). Directory traversal / path traversal (DT/PT) and source code disclosure (SCD) vulnerabilities let an attacker read files from the local file system. An operating system command injection (OSCI) vulnerability lets an attacker inject commands to be executed in a shell. A PHP command injection (PHPCI) vulnerability allows an attacker to supply PHP code that is executed by a PHP *eval* function.

4. THE APPROACH

The approach has two phases: *learning* and *detection*. In the first, an annotated data set is used to acquire knowledge about vulnerabilities. In the second, vulnerabilities are detected using a sequence model, a HMM. The HMM captures how calls to sanitization functions, validation and string modification affect the data flows between entry points and sensitive sinks. These factors may lead state to change from not tainted to tainted or vice-versa. However, we do not tell the model how to understand these functions, but train it automatically using the annotated data set (see Section 6).

The two phases are represented in Figure 1. The *learning phase* is executed when the corpus is first defined or later modified and is composed of the following sequence of steps:

(1) *Building the corpus:* to build the corpus with a set of source code slices annotated either as vulnerable or non-vulnerable, to characterize code with flaws and code that handles inputs adequately (see Section 6.1). Duplicates have to be removed;

(2) *Knowledge extraction:* to extract knowledge from the corpus (the *parameters* of the model) and represent it with probability matrices (see Section 6.2.4).

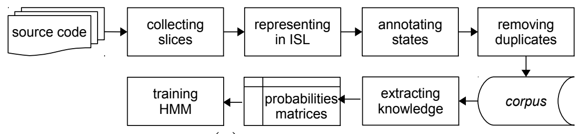
(3) *Training HMM:* to train the HMM to characterize vulnerabilities with knowledge contained in the *parameters*.

The *detection phase* is composed of the following steps:

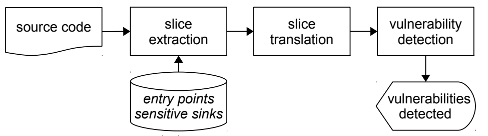
(1) *Slice extraction:* to extract slices from the source code, with each slice starting in an entry point and finishing in a sensitive sink. This is done by the *slice extractor*, which tracks the entry points and their dependencies until they reach a sensitive sink, independently if they are sanitized, validated and/or modified. The resulting slice is a sequence of tracked instructions;

(2) *Slice translation:* to translate the slice into *Intermediate Slice Language* (ISL). We designate the slice in ISL by *slice-isl*. During this translation, a *variable map* is created containing the variables present in the slice source code. ISL is a categorized language with grammar rules that aggregate in categories the functions of the server-side language by their functionality;

(3) *Vulnerability detection:* to use the HMM to find the best sequence of states that explains *slice-isl*. Each *slice-isl* instruction (sequence of observations) is classified by the model after the tainted variables from the previous instruction determine which emission probabilities will be selected for the instruction to be classified. The classification of the last observation from the last instruction of the *slice-isl* will classify the whole slice as containing a vulnerability or not. If a vulnerability is detected, its description (including its location in the source code) is reported.



(a) Learning phase.



(b) Detection phase.

Figure 1: Overview on the proposed approach.

Token	Description	PHP Func.	Taint
input	entry point	$_GET	Yes
var	variable	–	No
sanit_f	sanitization function	htmlentities	No
ss	sensitive sink	mysql_query	Yes
typechk_str	type checking string function	is_string	Yes
typechk_num	type checking numeric function	is_int	No
contentchk	content checking function	preg_match	No
fillchk	fill checking function	isset, is_null	Yes
cond	*if* instruction presence	if	No
join_str	join string function	implode, join	No
erase_str	erase string function	trim	Yes
replace_str	replace string function	preg_replace	No
split_str	split string function	str_split	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 vulnerable	–	Yes

Table 1: Intermediate Slice Language tokens.

5. INTERMEDIATE SLICE LANGUAGE

As explained, slices are translated into ISL. All slices begin with an entry point and end with a sensitive sink; between them there can be other entry point assignments, input validations, sanitizations, modifications, etc. A slice contains all instructions (lines of code) that manipulate an entry point and the variables that depend on it, but no other instructions. These instructions are composed of *code elements* (e.g., entry points, variables, functions) that are categorized in *classes* of elements with the same purpose (e.g., class *input* contains PHP entry points like `$_GET` and `$_POST`). The classes are the *tokens* of the ISL language. ISL is essentially a representation of the instructions in terms of these classes. Therefore, the representation of a slice in ISL is an abstraction of the original slice, which is simpler to process. Next we present the ISL, assuming the language of the code inspected is PHP, but the approach is generic and other languages could be considered.

5.1 ISL tokens and grammar

To define the ISL tokens, we studied which PHP code elements could manipulate entry points and be associated to vulnerabilities or prevent them (e.g., functions that do sanitization or replace characters in strings). Moreover, we examined many slices (vulnerable and not) to check the presence of these code elements. The code elements representing PHP functions were carefully studied to understand which of their parameters are relevant for vulnerability detection. Some code elements are represented by more than one token. For instance, the *mysql_query* function and its parameter are represented by two tokens: *ss* (sensitive sink) and *var* (variable; or *input* if the parameter is an entry point).

Table 1 shows the 21 ISL tokens (column 1). The first 20 represent code elements and their parameters, whereas the last is specific for the corpus (see Section 6). Each of the 20 tokens represents one or more PHP functions. Column 2 says the purpose of the functions and column 3 gives function examples (full list at [13]). Some remarks on some tokens. *cond* corresponds to an *if* statement with validation functions over variables (user inputs) from the slice. This token allows the correlation and verification of the relation between the validated variables and the variables that appear inside the *if* branches. *char5* and *char6* represent the amount of characters from a string manipulated by functions that extract or replace the user input contents. *start_where* represents the place in the string (begin, middle or end) where the user input contents suffers modifications by functions that extract or replace characters. *var_vv*, used in the corpus, represents variables that have *Taint* state (that are tainted, i.e., that have values that depend on inputs).

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

Figure 2: ISL grammar rules.

The ISL grammar is composed of the rules shown in Figure 2. It is used to translate a slice, composed of code elements (Table 1, column 3), into what we designate by *slice-isl*, composed of tokens (column 1). Each rule denotes how each code element is represented, as exemplified above for the *mysql_query* function and its parameter, where the *sensitive_sink* rule was applied (line 4 on Figure 2). A HMM processes observations from left to right and a PHP assignment instruction assigns the right-hand side to the left-hand side; the assignment rule in ISL follows the HMM scheme. This means, for example, that the PHP instruction `$u = $_GET['user'];` is translated to *input var*, where *input* is the right-hand side and *var* the left one.

5.2 Variable map

A *slice-isl* does not contain information about the variables represented by the *var* token. However, this information is crucial for the vulnerability detection process as *var* may apply to different variables and the existence of a vulnerability may depend on that information. Therefore, during slice translation a data structure called *variable map* is populated. This map associates each occurrence of *var* in the *slice-isl* with the name of the variable that appears in the source code. This allows tracking how input data propagates to different variables or is sanitized/validated or modified. Each line of the variable map starts with 1 or 0, indicating if the instruction is an assignment or not. The rest of the line contains one item per token in a *slice-isl* instruction.

5.3 Slice translation process

The process of slice translation consists in representing the slice using ISL and creating the corresponding variable map. This section presents this process with two examples.

The slice extractor analyses the source code, extracting slices that start in entry points and end in sensitive sinks. The instructions between these points are those that handle entry points and variables depending on them. The slice extractor performs intra- and inter-procedural analysis, as it tracks the entry points and their dependencies along the source code, walking through different files and functions. The analysis is also context-sensitive as it takes into account the results of function calls.

Figure 3(a) shows PHP code (a slice) vulnerable to SQLI and Figure 3(b) shows this code translated into ISL and the corresponding variable map (ignore the right-hand side for now). The first line represents the assignment of an input to a var: *input var* in ISL. The variable map entry starts with 1 (assignment) and has two items, one for *input* (-) and the other for *var* (*u*, the variable name without the $ character). The next line is a variable assignment represented by *var var* in ISL and by *1 u q* in the variable map. The last line contains a sensitive sink (*ss*) and two variables.

The second example is in Figure 4. The slice extractor takes from that code two slices: lines {1, 2, 3} and {1, 2, 4}. The first has input validation, but not the second that is vulnerable to XSS. The corresponding ISL and variable map are shown in the middle columns. The interesting cases are lines 2 and 3 that represent the *if* statement and its true branch. Both are prefixed with the *cond* token and the former also ends with the same token.

6. THE MODEL

This section presents the *model* used to learn and detect vulnerabilities. The section covers the two phases of the proposed approach (Section 4). The learning phase is mainly presented in Sections 6.1 and 6.2.4. The detection phase is presented in Section 6.3. In the learning phase, the corpus (a set of annotated sequences of observations) is used to set the *parameters* of the sequence model (matrices of probabilities). In the detection phase, a sequence of observations represented in ISL is processed by the model using the Viterbi algorithm [11] with some adaptations to decode the sequence of states that explains those observations. This algorithm is often used in NLP to decode (i.e., discover) the states given the observations. The states classify the observations as tainted or not; and in particular the last state of the sequence indicates if the slice is vulnerable or not.

6.1 Building the corpus

Our approach involves configuring the model automatically using machine learning. The *corpus* is a set of sequences of observations annotated with states, that contains the knowledge that will be learned by the model. The corpus is crucial for the approach as it includes the information about which sequences of instructions lead to vulnerabilities or not.

The corpus is built in four steps: *collecting* a set of (PHP) instructions associated with slices vulnerable and not vulnerable; *representing* these instructions in ISL (sequences of observations); *annotating* manually the state to each observation (to each ISL token) of the sequences; and *removing* duplicated sequences of observations annotated with states. The upper part of Figure 1(a) represents these steps.

The most critical step is the first, in which a set of slices representing existing vulnerabilities (and non-vulnerabilities) with different combinations of code elements has to be obtained. In practice we used a large number of slices from open source applications (see Section 7).

A sequence of the corpus is composed of two or more pairs ⟨`token,state`⟩. The instruction `$var = $_POST['paramater']`, for instance, translated into ISL becomes *input var* and is represented in the corpus as ⟨`input,Taint`⟩ ⟨`var_vv,Taint`⟩. Both states are *Taint* (compromised) because the *input* is always *Taint* (*input* is the source of attacks we consider).

In the corpus, the sequences of observations are annotated according to their taintdness status and type, as presented in column 4 of Table 1, and the tokens representing some class of functions from that table. For instance, the PHP instruction `$var = htlmentities($_POST['parameter'])` is translated to *sanit_f input var* and represented in the corpus by the sequence ⟨`sanit_f,San`⟩ ⟨`input, San`⟩ ⟨`var,N-Taint`⟩. The first two tokens were annotated with the *San* state, because the sanitization function sanitizes its parameter, and the last token was annotated with *N-Taint* state, meaning that the operation and the final state of the sequence are not tainted.

Notice that in the previous examples the state of the last observation is the final state of the sequence. In the sanitization example that state is *N-Taint*, indicating that the sequence is not-tainted (not compromised), while in the other example that state is *Taint*, indicating that the sequence is tainted (compromised).

As mentioned above, 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 token `var`. In the corpus, if the state of the variable is annotated as *Taint*, the variable is represented by `var_vv`, forming the pair ⟨`var_vv,Taint`⟩.

6.2 Sequence model

6.2.1 Hidden Markov model

A hidden Markov model (HMM) is a dynamic Bayesian network with nodes that represent random variables and edges that represent probabilistic dependencies between these variables [11]. These variables are divided in two sets: observed variables – *observations* – and hidden variables – *states*. The edges are the transition probabilities, i.e., the probabilities of going from one state to another. States are said to emit observations. A HMM is composed of: (1) a vocabulary, a set of symbols or tokens that compose the sequence of observations; (2) a set of states; (3) parameters, a set of probabilities: (i) the start-state or initial probabilities, which specify the probability of a sequence of observations starting in each state of the model; (ii) the transition probabilities; (iii) and the emission probabilities, which specify the probability of a state emitting a given observation. The parameters are calculated – *learned* – by counting observations and state transitions over the training corpus, afterwards normalizing the counts in order to obtain probability distributions, and using some smoothing procedure (e.g., add-one smoothing) to deal with rare events in the training data.

Sequence models correspond to a chain structure [11] (e.g., the sequence of observations of tokens in a *slice-isl*). These models use sequential dependencies in the states, meaning that the *i-th* state depends of the *i-1* previously generated states. In a HMM, the states are generated according to a

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='".$u."'";`	`var var`	`1 u q`	`TL = {u, q}`	⟨`var_vv_u,Taint`⟩ ⟨`var_vv_q,Taint`⟩
3 `$result = mysql_query($q);`	`ss var var`	`1 - q result`	`TL = {u, q, result}`	⟨`ss,N-Taint`⟩ ⟨`var_vv_q,Taint`⟩ ⟨`var_vv_result,Taint`⟩

(a) code with SQLI vulnerability (b) *slice-isl* (c) outputting the final classification

Figure 3: Code vulnerable to SQLI, translation into ISL, and detection of the vulnerability.

PHP code	slice-isl	variable map	list
1 `$u = $_POST['name'];`	`input var`	`1 - u`	`TL = {u}; CTL = {}`
2 `if (isset($u) && preg_match('/[a-zA-Z]+/', $u))`	`cond fillchk var contentchk var cond`	`0 - - u - u -`	`TL = {u}; CTL = {u}`
3 `echo $u;`	`cond ss var`	`0 - - u`	`TL = {u}; CTL = {u}`
4 `echo $u;`	`ss var`	`0 - u`	`TL = {u}; CTL = {}`

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

Figure 4: Code with a slice vulnerable to XSS (lines {1, 2, 4}) and a slice not vulnerable (lines {1, 2, 3}), with translation into ISL.

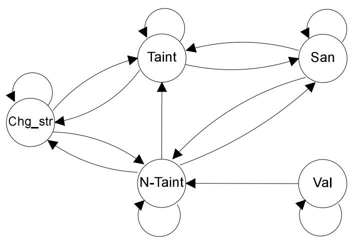


Figure 5: Model graph of the proposed HMM.

first order Markov process, in which the *i-th* state depends only of the previous state.

In the context of NLP, a HMM is often used to find the sequence of states that best explains a new sequence of observations, given the learned parameters. This is known as the *decoding problem*, which can be solved by the Viterbi decoding algorithm [30], by picking the best global hidden state sequence through dynamic programming. In a nutshell, the algorithm iteratively obtains the probability distribution for the *i-th* state based on the probabilities obtained for the *(i-1)-th* state and the learned parameters.

6.2.2 Vocabulary and states

The HMM vocabulary consists in the 21 ISL tokens. The HMM contains the 5 states in Table 2. The final state of *slice-isl* will be vulnerable (*Taint*) or not vulnerable (*N-Taint*), but for correct detection it is necessary to take into account sanitization (*San*), validation (*Val*) and modification (*Chg_str*) of the user inputs. Therefore these three factors are represented as intermediate states in the model.

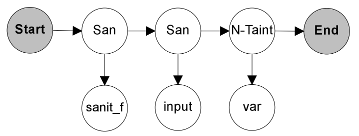
State	Description	Emitted observations
Taint	Tainted	input, var, var_vv, conc
N-Taint	Not tainted	Input, var, var_vv, ss, cond, conc
San	Sanitization	input, var, var_vv, sanit_f
Val	Validation	input, var, var_vv, typechk_str, typechk_num, contentchk, fillchk
Chg_str	Change string	input, var, var_vv,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

Table 2: HMM states and the observations they emit.

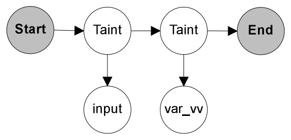
6.2.3 Model graph

The model uses the knowledge in the corpus to discover the states of new sequences of observations, detecting vulnerabilities. The knowledge that we want to be learned can be expressed as a graph, which represents the model to detect vulnerabilities. Figure 5 shows the graph for the specific HMM we use, where the nodes represent the states and the edges the transitions between them. Table 2 shows the observations that can be emitted in each state (column 3).

A sequence of observations can start in any state except *Val*, and end in the states *Taint* or *N-Taint*. The exception is due to validated instructions that begin with the *cond* observation (e.g., lines 2-3 in Figure 4), which is emitted by the *N-Taint* state, but after this observation the state transits to the *Val* state. In relation to the final state, an instruction (a sequence of observations) from *slice-isl* is classified for all its observations, where the state of the last observation will be the final state of all observations, meaning that an instruction is always classified as *Taint* or *N-Taint*. Therefore, the final state of the last instruction of *slice-isl* gives the final classification, i.e., says if the *slice-isl* is vulnerable or not. State outputs and transitions depend on the previously processed observations and the knowledge learned.



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



(b) PHP instruction: `$u = $_GET['user']`
ISL instruction: `input var`
Sequence: ⟨`input,Taint`⟩ ⟨`var_vv,Taint`⟩

Figure 6: Models for two example corpus sequences.

Figure 6 shows the instantiation of the graph for two sequences. The sanitization code of Figure 6(a) 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 (non-tainted variable). Figure 6(b) depicts the assignment of an entry point to a variable, turning this one tainted (*Taint*) and emitting *var_vv* (tainted variable).

6.2.4 Parameters

The *parameters* of the model are probabilities for the initial states, the state transitions, and symbol emissions (Section 6.2.1). The parameters are calculated using the corpus and the add-one smoothing technique to ensure that all probabilities are different from zero.

The probabilities are calculated from the corpus counting the number of occurrences of observations and/or states for each type of probability. The result are 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. For our model these numbers are 5 and 21, resulting in matrices of dimensions (1×5), (5×5) and (21×5). They are calculated as follows:

Initial-state probabilities: count how many sequences start in each state. Then, calculate the probability for each state dividing these counts by the number of sequences of the corpus, resulting in a matrix with the dimension (1×5).

Transition probabilities: count how many times in the corpus a certain state transits to another state (or to itself). Recall that we consider pairs of states. We can calculate the transition probability by dividing this count by the number of pairs of states from the corpus that begin with the start state. For instance, the transition probability from the *N-Taint* state to *Taint* state is the number of occurrences of this pair of states divided by the number of pairs of states starting in the *N-Taint* state. The resulting matrix has a dimension of (5×5), that represents the possible transitions between the 5 states.

Emission probabilities: count how many times in the corpus a certain token is emitted by a certain 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 of pairs ⟨token,state⟩ for that specific state. The resulting matrix – called *global emission probabilities matrix* – has a dimension of (21×5), representing the 21 tokens emitted by the 5 states.

Zero-probabilities have to be avoided because the Viterbi algorithm uses multiplication to calculate the probability of the next state, and therefore we need to ensure that this multiplication is never zero. The *add-one smoothing* technique [11] is used to calculate the parameters, avoiding zero probabilities. This technique adds a unit to all counts, making zero-counts equal to one and the associated probability different from zero.

6.3 Detecting vulnerabilities

This section describes the *detection* phase of Figure 1(b).

6.3.1 Detection

A sequence of observations in ISL is processed by the model using the Viterbi algorithm to decode the sequence of states. For each observation, the algorithm calculates the probability of each state emitting that observation, taking for this purpose the emission and transition probabilities and the maximum of probabilities calculated for the previous observation in each state, i.e., the order in which the observation appears in the sequence and the previous knowledge. For the first observation of the sequence the initial-state probabilities are used, whereas for the rest of the probabilities these are replaced by the maximum of probabilities calculated for each state for the previous observation. For emission probabilities, the matrix for the observations to be processed is retrieved from the global emission probabilities matrix. The multiplication of these probabilities is calculated for each state – *score of state* – and the maximum of scores is selected, assigning it the state with bigger score to the observation. The process is repeated for all observations and the last observation is the one with the highest probability of the states of the sequence. In our case, this probability classifies the sequence as *Taint* or *N-Taint*.

A *slice-isl* is composed by a set of sequences of observations. The model is applied to each sequence, classifying each one as tainted or not (*Taint*, *N-Taint*). However, for the classification to be correct the model needs to know which variables are tainted and propagate this information between the sequences processed. For this purpose, three artefacts are used in the model: the lists of tainted variables (*tainted list*, TL) (explained next), inputs and tainted variables validated by validation functions (*conditional tainted list*, CTL), and sanitized variables (*sanitized list*, SL) (Section 6.3.3).

There are two relevant interactions between the *variable map*, the emission probabilities and *var_vv* to fill the three lists in two moments of the sequence processing: *after* and *before*. *After:* if the sequence represents an assignment, i.e., the last observation of the sequence is a *var*, the variable map is visited to get the variable name for that *var*, then TL is updated: (i) inserting the variable name if the state is *Taint*; or (ii) removing it if its state is *N-Taint* and the variable belongs to TL. In case (ii) and in the presence of a sanitization sequence, SL is updated inserting the variable name; if the sequence represents an *if* condition (the first and last observations of the sequence must be *cond*), for each *var* and *var_vv* observation, the variable map is visited to get the variable name, next TL to verify if it contains the variable name, and then, in that case, CTL is updated inserting that variable name. *Before:* for each *var* observation, the variable map is visited to get the variable name, then TL and SL are accessed to verify if they contain that variable name. CTL is also accessed if the sequence starts with the token *cond*; in case of variable name only belong to TL, the *var* observation is updated to *var_vv*, then the emission probabilities matrix for the observations from the sequence is retrieved from the global emission probabilities matrix.

In order to detect vulnerabilities, the Viterbi algorithm was modified with these artefacts and interactions. Our model processes each sequence of observations from *slice-isl* as follows: (1) *"before"* is performed; (2) the decoding step of the Viterbi algorithm is applied; (3) *"after"* is performed.

6.3.2 Detection example

Figure 3 shows an example of detection. The figure contains from left to right: the code, the *slice-isl*, the variable map, and TL after the model classifies the sequence of observations. Observing TL, it is visible that it contains the tainted variables and that they propagate their state to the next sequences, influencing the emission probability of the variable. In line 1, the *var* observation is vulnerable because by default the *input* observation is so; the model classifies it correctly; and in TL the variable *u* is inserted. Next, line 2, before the Viterbi algorithm is applied the first *var* observation is updated to *var_vv* because it represents the *u* variable which belongs to TL. The *var_vv var* sequence is classified by the Viterbi algorithm, resulting in *Taint* as final state, and the variable *q* is inserted in TL. The process is repeated in the next line.

Figure 3(c) presents the decoding of *slice-isl*, where it is possible to observe the replacement of *var* by *var_vv*, with the variable name as suffix. Also, the states of each observation are presented and the state of the last observation indicates the final classification (there is a vulnerability). Looking for the states generated it is possible to understand the execution of the code without running it, why the code is vulnerable, and which variables are tainted.

6.3.3 Validation and sanitization

The *conditional tainted list* (CTL) is an artefact used to help interpret inputs and variables that are validated. This list will contain the *validated inputs* and *variables*, i.e., the inputs (token *input*) and tainted variables that belong to TL, and that are validated by validation functions (tokens *typechk_num* and *contentchk*). Therefore, when line 2 of Figure 4 is processed, this list is created and will be passed to the other sequences. That figure contains two *slice-isl* executed alternatively, depending on the result of the condition in line 2: {1, 2, 3} and {1, 2, 4}. When the model processes the former, it sets TL = {u} and CTL = {u}, as the variable {u} is the parameter of the *contentchk* token. The final state of the *slice-isl* (corresponding to line 3) is *N-Taint*, as the variable is in CTL. In the other slice there is no interaction with CTL and the final state is *Taint*.

The *sanitized list* (SL) is a third artefact. Its purpose is essentially the same as CTL, except that SL will contain variables sanitized using sanitization functions or modified using functions that, e.g., manipulate strings.

7. DEKANT AND THE CORPUS

To evaluate our approach and model we implemented them in the DEKANT tool. Moreover, we defined a corpus that we used to train the model before running the experiments. This corpus can be later extended with additional knowledge (remember that the tool is able to learn, so also to evolve).

7.1 DEKANT

The DEKANT tool was implemented in Java. The tool has four main modules: *knowledge extractor*, *slice extractor*, *slice translator*, and *vulnerability detector*.

The *knowledge extractor* module is independent of the other three and executed just when the corpus is first created or later modified. It runs in three steps. (1) *Corpus processing:* the sequences of the corpus are loaded from a plain text file; each sequence is separated in pairs ⟨`token,state`⟩ and the elements of each pair are inserted in the matrices of *observations* and *states*. (2) *Parameter calculation:* the parameters (probabilities) of the model are computed using the two matrices, and inserted in auxiliary matrices. (3) *Parameter storage:* the parameters are stored in a plain text file to be loaded by the *vulnerability detector* module.

The *slice extractor* extracts slices from PHP code by tracking data flows starting at entry points and ending at sensitive sinks, independently if the entry points are sanitized, validated and modified.

The *slice translator* parses the slices, translates them into ISL applying the grammar, and generates the variable maps.

The *vulnerability detector* works in three steps. (1) *Parameter loading:* the parameters (probabilities) are loaded from a text file and stored in matrices. (2) *Sequence of observations decoding:* the modified Viterbi algorithm is executed. (3) *Evaluation of sequences of observations:* the probability of a sequence of observations to be explained by a sequence of states is estimated, the most probable is chosen, and a vulnerability flagged if it exists.

7.2 Model and corpus assessment

A concern when specifying a HMM is to make it accurate and precise, i.e., to ensure that it classifies correctly sequences of observations or, in our case, that it detects vulnerabilities correctly. *Accuracy* measures the total of slices well-classified as vulnerable and non-vulnerable, whereas *precision* measures the fraction of vulnerabilities identified that are really vulnerabilities. The objective is high accuracy

		Observed	
		Vulnerable	Not Vulnerable
Predicted	Vulnerable	412	16
	Not Vulnerable	2	80

Table 3: Confusion matrix of the model tested with the corpus. *Observed* is the reality (414 vulnerable slices, 96 not vuln.). *Predicted* is the output of DEKANT with our corpus (428 vuln., 82 not vuln.).

and precision or, equivalently, minimum rates of false positives (inexistent vulnerabilities classified as vulnerabilities) and false negatives (vulnerabilities not classified as vulnerabilities). The model is configured with the corpus, so its accuracy and precision depend strongly on that corpus containing correct and enough information.

We created a corpus with 510 slices: 414 vulnerable and 96 non-vulnerable. These slices were extracted from several open source PHP applications[1] and contained vulnerabilities from the eight classes presented in Section 3.

To evaluate the accuracy and precision of the model configured with this corpus, we did *10-fold cross validation* [7], a common technique to validate training data. This form of validation involves dividing the training data (the corpus of 510 slices) in 10 folds. Then, the tool is trained with a pseudo-corpus of 9 of the folds and tested with the 10th fold. This process is repeated 10 times to test every fold with the model trained with the rest. This estimator allows assessing the quality of the corpus without the bias of testing data used for training or just a subset of the data.

The confusion matrix of Table 3 presents the results of this estimator. The precision and accuracy of the model were around 96%. The rate of false positives was 17% and the rate of false negatives almost null (0.5%). There is a tradeoff between these two rates and it is better to have a very low rate of false negatives that leads to some false positives (non-vulnerabilities flagged as vulnerabilities) than the contrary (missing vulnerabilities). These results show that the model has good performance using this corpus.

8. EXPERIMENTAL EVALUATION

The objective of the experimental evaluation was to answer the following questions using DEKANT and the corpus presented in the previous section: (1) Is *a tool that learns to detect vulnerabilities* able to detect vulnerabilities in plugins and real web applications? (Section 8.1) (2) Can it be more accurate and precise than other tools that do data mining using standard classifiers? (Section 8.2) (3) Can it be more accurate and precise than other tools that do taint analysis? (Section 8.3) (4) Is it able to classify correctly vulnerabilities independently of their class? (Section 8.1)

8.1 Open source software evaluation

To demonstrate the ability of DEKANT to classify vulnerabilities, we run it with 10 WordPress plugins [34] and 10 packages of real web applications, all written in PHP, using the corpus of the previous section. *The code used in the evaluation was not the same used to build the corpus.*

8.1.1 Zero-day vulnerabilities in plugins

WordPress is the most adopted CMS worldwide and supports plugins developed by different teams. Plugins are interesting because they are often less scrutinized than full

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

Plugin	Slices	Real vulnerabilities			N-Vul	FP
		SQLI	XSS	DT & LFI		
appointment-booking-calendar 1.1.7*	12	1	3	–	6	2
		CVE-2015-7319, CVE-2015-7320				
calculated-fields-form 1.0.60	3	–	–	–	2	1
contact-form-generator 2.0.1	5	–	–	–	4	1
easy2map 1.2.9*	6	–	1	2	3	0
		CVE-2015-7668, CVE-2015-7669				
event-calendar-wp 1.0.0	6	–	–	–	6	0
payment-form-for-paypal-pro 1.0.1*	11	–	2	–	8	1
		CVE-2015-7666				
resads 1.0.1*	2	–	2	–	0	0
		CVE-2015-7667				
simple-support-ticket-system 1.2*	20	5	–	–	15	0
		CVE-2015-7670				
wordfence 6.0.17	6	–	–	–	6	0
wp-widget-master 1.2	9	–	–	–	6	3
Total	80	6	8	2	56	8

*confirmed and fixed by the developers and registered in CVE

Table 4: Vulnerabilities found by DEKANT in WordPress plugins.

applications. We selected 10 plugins based on two criteria: development team and number of downloads. For the former, we choose 5 plugins developed by companies and the other 5 by individual developers. For the second, we choose 5 with less than 1000 downloads and the other 5 with more than 21,000 downloads. The plugins with less downloads were not always those developed by individual developers.

WordPress has a set of functions that sanitize and validate different data types, which are used by some of the plugins. Therefore, to run DEKANT with the source code of the plugins but without the WordPress code base, we added the information about those functions to the tool. Notice that the entry points and sensitive sinks remain mostly the same, except for sinks that handle SQL commands (*$wpdb* class). We configured DEKANT with these functions, mapping them to the ISL tokens. Recall that ISL abstracts the PHP instructions, so it can capture behaviors such as sanitization and validation even for the functions that were added.

DEKANT discovered 16 new vulnerabilities as shown in Table 4. 80 slices were extracted and translated into ISL. The tool classified 24 slices as vulnerable and 56 as not vulnerable (N-Vul), but 8 of the vulnerable were false positives (FP). This classification was confirmed by us manually. The 16 real vulnerabilities detected (columns 3-5) were 6 SQLI, 8 XSS, and 2 DT/LFI. These vulnerabilities were reported to the developers, who confirmed and fixed them, releasing new versions. The plugins *appointment-booking-calendar 1.1.7*, *easy2map 1.2.9*, *payment-form-for-paypal-pro 1.0.1*, *resads 1.0.1* and *simple-support-ticket-system 1.2* were fixed thanks to this work. We registered the vulnerabilities in CVE with the IDs shown in the table.

The 16 zero-day vulnerabilities were found in 5 plugins: 2 developed by companies and 3 by individual developers; plus 2 having more than 21,000 downloads. These results show that, independently of the development teams and the number of downloads, the WordPress plugins are vulnerable and may contain more vulnerabilities than other web applications, as recent research suggests [17].

8.1.2 Real web applications

To demonstrate the ability of DEKANT to classify vulnerabilities from the 8 classes, we run it with 10 open source software packages with vulnerabilities disclosed in the past. These packages were not used to build the corpus.

DEKANT classified 310 slices of the 10 applications. The results are in Table 5, columns 10-13. After this process we confirmed this classification manually in order to assess the results of DEKANT and the other tools (columns 2-5; Vul stands for vulnerable, San for sanitized, and VC for validated and/or changed). The 4 right-hand columns of the table show that DEKANT correctly classified 211 slices as being vulnerable (Vul) and the remaining as not-vulnerable (N-Vul), except 12 wrongly classified as vulnerable (false positives – FP). This misclassification is justified by the presence of validation and string modification functions (e.g., *preg_match* and *preg_replace*) with context-sensitive states. In such cases we set DEKANT to classify the slices as vulnerable but printing a warning on a possible false positive. Table 6 shows the confusion matrix summarizing these values. Overall, DEKANT had accuracy and precision of 96% and 95%, 12% of false positives, and no false negatives.

Web application	Slices				WAP				DEKANT			
	Vul	San	VC	Total	Vul	FPP	FP	FN	Vul	N-Vul	FP	FN
cacti-0.8.8b	2	0	8	10	2	2	6	0	2	6	2	0
communityEdition	16	36	8	60	16	6	2	0	16	44	0	0
epesi-1.6.0-20140710	25	1	8	34	25	6	2	0	25	5	4	0
NeoBill0.9-alpha	19	0	0	19	19	0	0	0	19	0	0	0
phpMyAdmin-4.2.6-en	1	6	7	14	1	0	7	0	1	13	0	0
refbase-0.9.6	5	4	3	12	5	0	3	0	5	1	6	0
Schoolmate-1.5.4	120	0	0	120	117	0	0	3	120	0	0	0
VideosTube	1	0	2	3	1	1	1	0	1	2	0	0
Webchess 1.0	20	0	0	20	18	0	0	2	20	0	0	0
Zero-CMS.1.0	2	5	11	18	2	5	6	0	2	16	0	0
Total	211	52	47	310	206	20	27	5	211	87	12	0

Table 5: Results of running the slice extractor, WAP and DEKANT in open source software.

	Observed							
	DEKANT		WAP		Original		Analyzed	
Predicted	Vul	N-Vul	Vul	N-Vul	Vul	N-Vul	Vul	N-Vul
Vul	211	12	206	27	182	36	50	218
N-Vul	0	87	5	72	86	821	109	748

Table 6: Confusion matrix of DEKANT, WAP and C4.5/J48 in PhpMinerII data set (original and analyzed).

Table 7 summarizes the results and presents additional metrics. For the 10 packages, more than 4,200 files and 1,525,865 lines of code were analyzed and 223 vulnerabilities found (12 false positives). The largest packages were *epesi* and *phpMyAdmin* (741 and 241 thousand lines of code).

Table 8 presents the 223 slices classified by DEKANT as vulnerable (12 false positives) distributed by the 6 classes of vulnerabilities. Interestingly, all false positives were PHPCI and XSS vulnerabilities. The tool correctly classified the sanitized slices as not vulnerable. The vulnerabilities correctly classified by DEKANT correspond to 21 entries of vulnerabilities that appear in CVE [4] and OSVDB [18].

8.2 Comparison with data mining tools

To answer the second question, DEKANT was compared with WAP and PHPMinerII with the 10 packages of the previous section. We opted by evaluating these tools with those packages and not with the plugins, because they are not configurable for the plugins. When run with the plugins these tools provide much worse results than DEKANT.

Both tools also classify slices previously extracted, but using data mining based on standard classifiers, which do not consider order. WAP performs taint analysis to extract the slices that start in an entry point and reach a sensitive sink, with attention to sanitization, then uses data mining to predict if they are false positives or real vulnerabilities. The tool deals with the same vulnerability classes as DEKANT. PhpMinerII uses data mining to classify slices as being vulnerable or not, without considering false positives. This tool handles only SQLI and reflected XSS vulnerabilities.

8.2.1 Comparison for all vulnerability classes

Columns 6 to 9 of Table 5 present WAP's results for the 8 vulnerability classes. WAP reported 206 vulnerabilities (Vul), 20 false positives predicted (FPP), with 27 false positives and 5 false negatives (vulnerabilities not de-

Web application	Files	Lines of code	Analysis time (s)	Vuln. files	Vulner. found
cacti-0.8.8b	249	95,274	7	7	4
communityEdition	228	217,195	21	11	16
epesi-1.6.0-20140710	2246	741,440	90	13	29
NeoBill0.9-alpha	620	100,139	5	5	19
phpMyAdmin-4.2.6-en	538	241,505	12	1	1
refbase-0.9.6	171	109,600	8	5	11
Schoolmate-1.5.4	64	8,411	2	41	120
VideosTube	39	3,458	2	1	1
Webchess 1.0	37	7,704	2	5	20
Zero-CMS.1.0	21	1,139	2	2	2
Total	4,213	1,525,865	151	91	223

Table 7: Summary of results of DEKANT with open source code.

Web application	SQLI	RFI, LFI DT/PT	PHPCI	XSS	Total
cacti-0.8.8b	0	0	2	2	4
communityEdition	4	4	3	5	16
epesi-1.6.0-20140710	0	3	4	22	29
NeoBill0.9-alpha	0	2	0	17	19
phpMyAdmin-4.2.6-en	0	0	0	1	1
refbase-0.9.6	0	0	0	11	11
Schoolmate-1.5.4	69	0	0	51	120
VideosTube	0	0	0	1	1
Webchess 1.0	6	0	0	14	20
Zero-CMS.1.0	1	0	0	1	2
Total	80	9	9	125	223

Table 8: Results of the classification of DEKANT considering different classes of vulnerabilities extracted by the slice extractor.

tected). WAP identified the same 258 slices without sanitization (columns 2 and 4 from Table 5) than the slice extractor and detected the same 206 vulnerabilities than DEKANT (5 less than DEKANT, false negatives, FN). Moreover and as expected, from the 47 slices classified as not vulnerable by DEKANT, WAP predicted correctly 20 of them as false positives (FPP), meaning that 27 slices were wrongly classified as vulnerabilities (FP), reporting 27 false positives.

This difference of false positives is justified by: (1) the presence of symptoms in the slice which are not contemplated by WAP as attributes in its data set; (2) lack of verification of the relations between attributes, once the data mining mechanism only verifies the presence of the attributes in the slice, does not relates them. The false negatives are justified by reason (2) plus the importance of the order of the code elements in the slice. The misclassification was based in the concatenation of variables tainted with not-tainted (variables validated or modified), in that order; then data mining matches the presence of symptoms related with validation and classified the slices as false positives. In these 5 slices is evident the importance of the order of code elements for a correct classification and detection. DEKANT implements a sequence model that takes into account that order, prevailing in these cases.

Columns 4 and 5 of Table 6 present the confusion matrix with these values. WAP had an accuracy of 90%, a precision of 88%, 2% of false negatives and 27% of false positives (Table 10, third column).

8.2.2 *Comparison for SQLI and reflected XSS*

For a fair comparison with PHPMinerII, only SQLI and reflected XSS vulnerabilities classes considered. Table 9 shows the results; columns 2 to 4 are the 158 vulnerabilities classified by DEKANT (80 SQLI, 78 XSS) and 6 false positives. The next four columns are about WAP, with the 153 vulnerabilities (77 SQLI, 76 XSS), but with 21 false positives and 5 false negatives. The next 12 columns present the PHPMinerII results.

PhpMinerII does not come trained, so we had to create a data set to train it. For that purpose, PhpMinerII extracts slices that end in a sensitive sink, but that do not have to start in an entry point. It outputs the slices, the vector of attributes of each slice, and a preliminary classification as

Metric	DEKANT	WAP	PhpMinerII original	PhpMinerII analyzed	Pixy
acurracy	96%	90%	89%	71%	18%
precision	95%	88%	83%	19%	13%
false positive	12%	27%	4%	23%	87%
false negative	0%	2%	32%	69%	24%

Table 10: Evaluation metrics of DEKANT, WAP, PhpMinerII, Pixy.

vulnerable or not. Then a classification has to be assigned to each attribute vector manually. This data set is used to train the data mining part of the tool. We present experimental results of the tool running it both without and with data mining. Table 9 shows the analysis without data mining and the intersection of both sets of slices for SQLI and XSS with the DEKANT slices. For these two classes, columns 9, 10, 15 and 16 (Yes, No) show the number of slices classified by the tool, columns 11 and 17 (Y - Y) show the intersection (the number of vulnerabilities detected by both tools), whereas columns 12 and 18 (Y - N) depict the number of vulnerabilities that DEKANT classified correctly but PHPMinerII did not report. We observe that from the SQLI vulnerabilities detected by DEKANT, PHPMinerII only detected correctly approximately 16%, presenting high rates of false negatives and false positives. For XSS, PHPMiner II presents again an elevated rate of false negatives and false positives, besides a small number of true positives compared with the number of vulnerabilities detected by DEKANT.

To perform the data mining process the WEKA tool was used [33] with the same classifiers as PhpMinerII [24, 25]. The best classifier was the C4.5/J48. Columns 6 to 9 of Table 6 show the results of this classifier. The first two columns of these four are relative to the slices flagged by the tool without data mining, while the last two columns are relative to the data mining process presented above. The accuracy and precision are equal to 71% and 19%, and the false positives and negatives rates are 23% and 69%, justifying the very low precision rate.

Table 10 summarizes the comparison between DEKANT, WAP and PhpMinerII. DEKANT was the best of all. WAP was the second, also with low false negatives but high false positives. Despite PhpMinerII presenting the lowest false positive rate, it had the highest rate of false negatives and lower accuracy and precision rates, making it the weakest tool (false negatives are specially problematic as they represent vulnerabilities that were not found).

8.3 Comparison with taint analysis tools

We compare DEKANT with Pixy [10], a tool that performs taint analysis to detect SQLI and reflected XSS vulnerabilities, taking sanitization functions in consideration. The last four columns of Table 9 are related to the analysis made with Pixy. Despite Pixy reporting 902 vulnerabilities in 10 packages, they are mostly false positives. Those vulnerabilities were 421 SQLI and 481 XSS (first two columns of the last 4). The same process of the previous section was executed over the results of Pixy. In summary, only 120 vulnerabilities are the same as for DEKANT, while the rest are false positives and some false negatives (last 2 columns).

9. DISCUSSION

DEKANT is a static analysis tool because it searches for vulnerabilities in source code, without execution. DEKANT has two main parts: one programmed, another learned. The former corresponds to the slice extractor that does part of what other static analysis tools do: parses the code and ex-

Web application	DEKANT			WAP				PhpMinerII - SQLI						PhpMinerII - XSS						Pixy			
	SQLI	XSS	FP	SQLI	XSS	FP	FN	Yes	No	Y - Y	Y - N	FP	FN	Yes	No	Y - Y	Y - N	FP	FN	SQLI	XSS	FP	FN
cacti-0.8.8b	0	2	0	0	2	6	0	0	0	0	0	0	0	6	11	2	0	4	0	0	6	4	0
communityEdition	4	5	0	4	5	0	0	5	0	0	0	5	4	43	521	0	0	43	5	5	8	13	9
epesi-1.6.0-20140710	0	18	0	0	18	0	0	0	0	0	0	0	0	1	1	1	0	0	16	0	1	0	17
NeoBill0.9-alpha	0	17	0	0	17	0	0	0	0	0	0	0	0	20	3	17	0	3	0	0	20	3	0
phpMyAdmin-4.2.6-en	0	1	0	0	1	6	0	0	0	0	0	0	0	24	74	0	0	24	1	–	25	24	0
refbase-0.9.6	0	5	6	0	5	3	0	3	0	0	0	3	0	82	115	0	1	82	5	3	93	96	5
Schoolmate-1.5.4	69	14	0	66	14	0	3	41	11	11	0	30	58	2	0	2	0	0	12	303	113	339	6
VideosTube	0	1	0	0	1	0	0	10	19	0	0	10	0	2	28	1	0	1	0	12	2	13	0
Webchess 1.0	6	14	0	6	12	0	2	1	0	1	0	0	5	13	7	13	0	0	1	92	206	279	1
Zero-CMS.1.0	1	1	0	1	1	6	0	6	2	1	1	5	0	9	65	1	0	8	0	6	7	11	0
Total	80	78	6	77	76	21	5	66	32	13	1	53	67	202	825	37	1	165	40	421	481	782	38

Table 9: Comparison of results between DEKANT, WAP, PHPMinerII and Pixy with open source projects.

tracts slices. The latter uses the sequence model we propose, configured with knowledge extracted from the corpus.

In classic static analysis tools this knowledge was programmed, involving several data structures and variables representing and relating the code elements that create and avoid vulnerabilities. Programming this knowledge is a hard, complex task, for the programmers, who may leave errors that lead to false positives and false negatives [6].

Taking this difficulty into account, machine learning started to be used to reduce the effort required to programming static analysis tools. Table 10 compares the results of WAP and PHPMinerII (both use machine learning) with Pixy (an older tool that does not use it). In that table it is possible to see that tools based on machine learning can provide good results. The application of data mining requires a definition of a data set with the knowledge about vulnerabilities, making it a crucial part of the process for correct detection.

This paper presents the first static analysis approach and tool that learns to detect vulnerabilities automatically using machine learning (WAP has most knowledge programmed and PHPMinerII does not identify vulnerabilities, only predicts if they exist). Furthermore, we go one step further by using for the first time in this context a sequence model instead of standard classifiers. This model not only considers the code elements that appear in the slices, but also their order and relations between them. Again, similarly to what happens with standard classifiers, the definition of the corpus for the sequence model is crucial. Table 10 compares the results of DEKANT with WAP and PHPMinerII, showing that this approach indeed improves the results.

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

We thank the anonymous reviewers and Bruno Martins for their comments on a previous version of the paper. This work was partially supported by the EC through project FP7-607109 (SEGRID), and by national funds through Fundação para a Ciência e a Tecnologia (FCT) with references UID/CEC/50021/2013 (INESC-ID) and UID/CEC/00408/2013 (LaSIGE).

11. REFERENCES

[1] Arisholm, E., Briand, L.C., Johannessen, E.B.: A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. Journal of Systems and Software 83(1), 2–17 (2010)

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

[3] Briand, L.C., Wüst, J., Daly, J.W., Porter, D.V.: Exploring the relationships between design measures and software quality in object-oriented systems. Journal of Systems and Software 51(3), 245–273 (2000)

[4] CVE: http://cve.mitre.org

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

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

[7] Demšar, J.: Statistical comparisons of classifiers over multiple data sets. The Journal of Machine Learning Research 7, 1–30 (Dec 2006)

[8] Fonseca, J., Vieira, M.: A practical experience on the impact of plugins in web security. In: Proceedings of the 33rd IEEE Symposium on Reliable Distributed Systems. pp. 21–30 (Oct 2014)

[9] Huang, Y.W., Yu, F., Hang, C., Tsai, C.H., Lee, D.T., Kuo, S.Y.: Securing web application code by static analysis and runtime protection. In: Proceedings of the 13th International World Wide Web Conference. pp. 40–52 (2004)

[10] Jovanovic, N., Kruegel, C., Kirda, E.: Precise alias analysis for static detection of web application vulnerabilities. In: Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security. pp. 27–36 (Jun 2006)

[11] Jurafsky, D., Martin, J.H.: Speech and Language Processing. Prentice Hall (2008)

[12] Lessmann, S., Baesens, B., Mues, C., Pietsch, S.: Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Transactions on Software Engineering 34(4), 485–496 (2008)

[13] Medeiros, I., Neves, N.F., Correia, M.: Additional info on the paper to ISSTA 2016 (Jan 2016), https://sites.google.com/site/siteaddinfo/issta2016

[14] Medeiros, I., Neves, N.F., Correia, M.: Detecting and removing web application vulnerabilities with static analysis and data mining. IEEE Transactions on Reliability 65(1), 54–69 (March 2016)

[15] Medeiros, I., Neves, N.F., Correia, M.: Equipping WAP with weapons to detect vulnerabilities. In: Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (2016)

[16] Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A.: Predicting vulnerable software components. In: Proceedings of the 14th ACM Conference on Computer and Communications Security. pp. 529–540 (2007)

[17] Nunes, P., Fonseca, J., Vieira, M.: phpSAFE: A security analysis tool for OOP web application plugins. In: Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (Jun 2015)

[18] OSVDB: http://osvdb.org

[19] Perl, H., Dechand, S., Smith, M., Arp, D., Yamaguchi, F., Rieck, K., Fahl, S., Acar, Y.: 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. pp. 426–437. CCS '15 (Oct 2015)

[20] Rabiner, L.R.: A tutorial on hidden Markov models and selected applications in speech recognition. Proceedings of the IEEE 77(2), 257–286 (1989)

[21] Scandariato, R., Walden, J., Hovsepyan, A., Joosen, W.: Predicting vulnerable software components via text mining. IEEE Transactions on Software Engineering 40(10), 993–1006 (2014)

[22] SchoolMate: Http://sourceforge.net/projects/schoolmate/

[23] Shankar, U., Talwar, K., Foster, J.S., Wagner, D.: Detecting format-string vulnerabilities with type qualifiers. In: Proceedings of the 10th USENIX Security Symposium (Aug 2001)

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

[25] Shar, L.K., Tan, H.B.K.: 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. pp. 310–313 (2012)

[26] Shar, L.K., Tan, H.B.K., Briand, L.C.: Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis. In: Proceedings of the 35th International Conference on Software Engineering. pp. 642–651 (2013)

[27] Shin, Y., Meneely, A., Williams, L., Osborne, J.A.: Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Transactions on Software Engineering 37(6), 772–787 (2011)

[28] Son, S., Shmatikov, V.: SAFERPHP: Finding semantic vulnerabilities in PHP applications. In: Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security (2011)

[29] TheGuardian: Ashley Madison condemns attack as experts say hacked database is real (2015), http://www.theguardian.com/technology/2015/aug/19/ashley-madisons-hacked-customer-files-posted-online-as-threatened-say-reports

[30] Viterbi, A.: Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. IEEE Transactions on Information Theory 13(2), 260–269 (Apr 1967)

[31] Walden, J., Doyle, M., Welch, G.A., Whelan, M.: Security of open source web applications. In: Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement. pp. 545–553 (2009)

[32] Williams, J., Wichers, D.: OWASP Top 10 2013 – the ten most critical web application security risks (2013)

[33] Witten, I.H., Frank, E., Hall, M.A.: Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann, 3rd edn. (2011)

[34] WordPress: https://wordpress.org/

[35] Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: Proceedings of the 2014 IEEE Symposium on Security and Privacy. pp. 590–604 (May 2014)

[36] Yamaguchi, F., Wressnegger, C., Gascon, H., Rieck, K.: Chucky: Exposing missing checks in source code for vulnerability discovery. In: Proceedings of the 20th ACM SIGSAC Conference on Computer Communications Security. pp. 499–510 (Nov 2013)

[37] ZeroCMS: Content management system built using PHP and MySQL, http://www.aas9.in/zerocms/