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

Ibíria 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 compromissed 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 sanitation 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 which 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 artificial 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.
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 analyser (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 (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.

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.

```php
<?php
$user_id = (isset($_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.

3. 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 5).

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 5.1). 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 5.2.4).

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.

4. 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.

4.1 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: as (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 5). Each of the 20 tokens represents one or more PHP functions. Col-
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: \texttt{$u = \_\_\_GET['user'];$}. The variable map entry starts with 1 (assignment) and has two items, one for \texttt{$\_\_\_GET($\texttt{user}$);} and the other for \texttt{var} ($u$, the variable name without the \$ character). The next line is a variable assignment represented by \texttt{var} in ISL and by \texttt{\texttt{var} u q} in the variable map. The last line contains a sensitive sink (\texttt{az}) 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.

5. THE MODEL

This section presents the model used to learn and detect vulnerabilities. The section covers the two phases of the proposed approach (Section 3). The learning phase is mainly presented in Sections 5.1 and 5.2.4. The detection phase is presented in Section 5.3. 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 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.

5.1 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
null
number of states and tokens of the model. For our model these numbers are 5 and 21, resulting in matrices of dimensions \((1 \times 5), (5 \times 5)\) and \((21 \times 5)\). 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 \((1 \times 5)\).

*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 \(N\)-\text{Taint} state to \text{Taint} state is the number of occurrences of this pair of states divided by the number of pairs of states starting in the \(N\)-\text{Taint} state. The resulting matrix has a dimension of \((5 \times 5)\), 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 \((\text{token}, \text{state})\) appears in the corpus. Then, calculate the emission probability by dividing this count by the total of pairs \((\text{token}, \text{state})\) for that specific state. The resulting matrix – called global emission probabilities matrix – has a dimension of \((21 \times 5)\), 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.

### 5.3 Detecting Vulnerabilities

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

#### 5.3.1 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 \text{Taint} or \(N\)-\text{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 \( (\text{Taint}, \overline{\text{Taint}}) \). 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 \( \text{(tainted list, TL)} \) (explained next), inputs and tainted variables validated by validation functions \( \text{(conditional tainted list, CTL)} \), and sanitized variables \( \text{(sanitized list, SL)} \) (Section 5.3.3).

There are two relevant interactions between the \textit{variable map}, the emission probabilities and \textit{var\_vv} to fill the three lists in two moments of the sequence processing: \textit{after} and \textit{before}. \textit{After:} if the sequence represents an assignment, i.e., the last observation of the sequence is a \textit{var\_vv}, the variable map is visited to get the variable name for that \textit{var\_vv}, then TL is updated: (i) inserting the variable name if the state is \textit{Taint}; or (ii) removing it if its state is \textit{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 \textit{if} condition (the first and last observations of the sequence must be \textit{cond}), for each \textit{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. \textit{Before:} for each \textit{var\_vv} 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 \textit{cond}; in case of variable name only belong to TL, the \textit{var\_vv} observation is updated to \textit{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 \textit{slice-isl} as follows: (1) \textit{“before”} is performed; (2) the decoding step of the Viterbi algorithm is applied; (3) \textit{“after”} is performed.

5.3.2 Detection Example

Figure 3 shows an example of detection. The figure contains from left to right: the code, the \textit{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 \textit{var\_vv} observation is vulnerable because by default the \textit{input} observation is so; the model classifies it correctly; and in TL the variable \textit{u} is inserted. Next, line 2, before the Viterbi algorithm is applied the first \textit{var\_vv} observation is updated to \textit{var\_vv} because it represents the \textit{u} variable which belongs to TL. The \textit{var\_vv} \textit{var\_vv} sequence is classified by the Viterbi algorithm, resulting in \textit{Taint} as final state, and the variable \textit{q} is inserted in TL. The process is repeated in the next line.

Figure 3(c) presents the decoding of \textit{slice-isl}, where it is possible to observe the replacement of \textit{var\_vv} by \textit{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.

5.3.3 Validation and Sanitization

The \textit{conditional tainted list} (CTL) is an artefact used to help interpret inputs and variables that are validated. This list will contain the \textit{validated inputs and variables}, i.e., the inputs (token \textit{input}) and tainted variables that belong to TL, and that are validated by validation functions (tokens \textit{typechk\_num} and \textit{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 \textit{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 = \{\textit{u}\} and CTL = \{\textit{u}\}, so the variable \{\textit{u}\} is the parameter of the \textit{contentchk} token. The final state of the \textit{slice-isl} (corresponding to line 3) is \textit{N-Taint}, as the variable is in CTL. In the other slice there is no interaction with CTL and the final state is \textit{Taint}.

The \textit{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.

6. 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).

6.1 DEKANT

The DEKANT tool was implemented in Java. The tool has four main modules: \textit{knowledge extractor, slice extractor, slice translator, and vulnerability detector}. The \textit{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) \textit{Corpus processing}: the sequences of the corpus are loaded from a plain text file; each sequence is separated in pairs \{\textit{token, state}\} and the elements of each pair are inserted in the matrices of \textit{observations and states}. (2) \textit{Parameter calculation}: the parameters (probabilities) of the model are computed using the two matrices, and inserted in auxiliary matrices. (3) \textit{Parameter storage}: the parameters are stored in a plain text file to be loaded by the \textit{vulnerability detector} module.

The \textit{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 \textit{slice translator} parses the slices, translates them into ISL applying the grammar, and generates the variable maps.

The \textit{vulnerability detector} works in three steps. (1) \textit{Parameter loading}: the parameters (probabilities) are loaded from a text file and stored in matrices. (2) \textit{Sequence of observations decoding}: the modified Viterbi algorithm is executed. (3) \textit{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.

6.2 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. \textit{Accuracy} measures the total of slices well-classified as vulnerable and non-vulnerable, whereas \textit{precis-
7. 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 7.1) (2) Can it be more accurate and precise than other tools that do data mining using standard classifiers? (Section 7.2) (3) Can it be more accurate and precise than other tools that do taint analysis? (Section 7.3) (4) Is it able to classify correctly vulnerabilities independently of their class? (Section 7.1)

7.1 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.

7.1.1 Zero-day Vulnerabilities in Plugins

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.

7.1.2 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 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.

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

<table>
<thead>
<tr>
<th>Predicted</th>
<th>Vul</th>
<th>N-Vul</th>
<th>Observed</th>
<th>Vul</th>
<th>N-Vul</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vul</td>
<td>211</td>
<td>26</td>
<td>116</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>N-Vul</td>
<td>8</td>
<td>27</td>
<td>152</td>
<td>50</td>
<td>218</td>
</tr>
<tr>
<td>Total</td>
<td>219</td>
<td>33</td>
<td>268</td>
<td>52</td>
<td>438</td>
</tr>
</tbody>
</table>

Table 7: Results of running the slice extractor, WAP and DEKANT in open source software.

<table>
<thead>
<tr>
<th>Web application</th>
<th>Slices</th>
<th>Vul</th>
<th>N-Vul</th>
<th>Vul</th>
<th>N-Vul</th>
<th>Vul</th>
<th>N-Vul</th>
</tr>
</thead>
<tbody>
<tr>
<td>communityEdition</td>
<td>228</td>
<td>217</td>
<td>175</td>
<td>1</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>epsesi-1.6.0-20140710</td>
<td>2246</td>
<td>741</td>
<td>440</td>
<td>90</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NeoBill-9-alpha</td>
<td>620</td>
<td>100</td>
<td>139</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>phpMyAdmin-4.2.6-en</td>
<td>538</td>
<td>241</td>
<td>540</td>
<td>12</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>refbase-0.9.6</td>
<td>171</td>
<td>169</td>
<td>600</td>
<td>8</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Schoolmate-1.5.4</td>
<td>64</td>
<td>8</td>
<td>43</td>
<td>2</td>
<td>120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VideosTube</td>
<td>39</td>
<td>3</td>
<td>45</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Webchess 1.0</td>
<td>37</td>
<td>7</td>
<td>704</td>
<td>2</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-CMS 1.0</td>
<td>21</td>
<td>2</td>
<td>1,139</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>2246</td>
<td>741</td>
<td>241,505</td>
<td>1</td>
<td>20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8: Summary of results of DEKANT with open source code.

<table>
<thead>
<tr>
<th>Web application</th>
<th>Files</th>
<th>Lines of code</th>
<th>Analysis time (s)</th>
<th>Vul</th>
<th>Vulner-Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>communityEdition</td>
<td>228</td>
<td>217,195</td>
<td>21</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>epsesi-1.6.0-20140710</td>
<td>2246</td>
<td>741,440</td>
<td>90</td>
<td>13</td>
<td>29</td>
</tr>
<tr>
<td>NeoBill-9-alpha</td>
<td>620</td>
<td>100,139</td>
<td>5</td>
<td>5</td>
<td>19</td>
</tr>
<tr>
<td>phpMyAdmin-4.2.6-en</td>
<td>538</td>
<td>241,505</td>
<td>12</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>refbase-0.9.6</td>
<td>171</td>
<td>169,600</td>
<td>8</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>Schoolmate-1.5.4</td>
<td>64</td>
<td>8,411</td>
<td>2</td>
<td>41</td>
<td>120</td>
</tr>
<tr>
<td>VideosTube</td>
<td>39</td>
<td>3,458</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Webchess 1.0</td>
<td>37</td>
<td>7,704</td>
<td>2</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>Zero-CMS 1.0</td>
<td>21</td>
<td>1,139</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td>2246</td>
<td>741,505</td>
<td>241,505</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

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

7.2 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.

7.2.1 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 detected). 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).

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

<table>
<thead>
<tr>
<th>Metric</th>
<th>DEKANT</th>
<th>WAP</th>
<th>PhpMinerII original</th>
<th>PhpMinerII mined</th>
<th>Pixy</th>
</tr>
</thead>
<tbody>
<tr>
<td>accuracy</td>
<td>98%</td>
<td>96%</td>
<td>96%</td>
<td>75%</td>
<td>18%</td>
</tr>
<tr>
<td>precision</td>
<td>95%</td>
<td>88%</td>
<td>93%</td>
<td>19%</td>
<td>13%</td>
</tr>
<tr>
<td>false positive</td>
<td>12%</td>
<td>27%</td>
<td>4%</td>
<td>23%</td>
<td>87%</td>
</tr>
<tr>
<td>true negative</td>
<td>0%</td>
<td>2%</td>
<td>3%</td>
<td>69%</td>
<td>24%</td>
</tr>
</tbody>
</table>

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

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 (SEG-RID), 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).

9. REFERENCES

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


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

