Various Approaches for Software Vulnerability Assessment and Risk Identification using Soft Computing Techniques

Ali Hussein¹, Azri Azmi², Hafiza Abas³

¹,²,³Faculty of Artificial Intelligence, Universiti Teknologi Malaysia, Jalan Sultan Yahya Petra, 54100 Kuala Lumpur

¹hussein.a@graduate.utm.my, ²azriazmi@utm.my, ³hafiza.kl@utm.my

Abstract

The detection of software flaws is a viable technique for enhancing the quality of technology and testing management by providing rapid detection of deficiency simulation models until the actual testing phase starts. These prediction outcomes help designers of technology effectively devote their available resources to components that are more vulnerable to deficiencies. In this research investigation, a software bug prediction model is proposed using Deep Learning (DL) approach. The Recurrent Neural Network (RNN) is used for classification of source code including numerous soft computing techniques. Numerous preprocessing and data filtration techniques have been carried out for data balancing and normalization. The Term Frequency and Inverse Document Frequency (TF-IDF) and relation features extraction techniques is used to generate Vector Space Model (VSM). The classification has been done using RNN on both training and validation dataset. The evaluation of performance of proposed work is performed using various real-time and synthetic accessible databases. It is observed from the experimental results that the proposed framework performs better when different evaluated with different datasets.

Keywords: Prediction, Software Flaw Prediction, RNN, Deep Learning Classifier

1. Introduction

Recognition of vulnerabilities in source code or software is a popular field of research throughout the software field [1]. In spite of the fact that existing research has demonstrated the usefulness of utilizing different detection techniques, models, and software flaw assessment tools, improving the effectiveness of these detection tools and models continues to be an essential challenge for researchers. Yearly, thousands of security issues are identified in virtual instruments, either released publicly to the general vulnerabilities database exposed in obfuscated code [2]. Threats also occur in indirect ways which are not evident to code inspectors or the programmers involved. There is an incentive to understand the dynamics of vulnerabilities that can lead to system issues directly from raw data, abundance of source code available publicly [3]. In this work, using deep learning, an information approach to security technology is proposed. Inspired by the progress of the design
in these areas of study, a conceptual framework is utilized to investigate the practicability of the design within the context of the area of vulnerability identification. The research provides an overview of the static analytics software and discusses the Deep Neural Network (DNN) based conceptual perspective for detecting attacks.

In order to address the disparity between domains, it is suggested that a potential solution might involve treating each feature within a program in the field of computer vision as a neural network. This approach would enable fault detectors to determine the security status of a feature and provide comprehensive details regarding the positions of vulnerabilities. Thus, a finer-grained interpretation of fault management programs are needed. There are important exceptions to this diagnosis: most comments in a program may not contain any uncertainty, indicating that few samples are susceptible; and multiple comments are not regarded as a whole that is semantically linked to each other.

This paper introduces a novel active tracking on deep learning to automatically learn features to predict runtime environment weaknesses. In source code, where contingent code components are spread far apart, Long Short Term Memory (LSTM) is leveraged to capture long-standing relationships. For example, combinations of code tokens that are needed to appear simultaneously due to the configuration of the computer program (e.g. in Java) or according to the configuration of API use (e.g. Lock() and activate()), but do not accompany each other automatically. The trained features (syntactic features) accurately capture both the interpretation of code symbols (referred to as semantic functionality) and the hierarchical arrangement of source code. The proposed automated feature learning strategy removes a need for automated feature selection in conventional methods, which takes up much effort. Finally, testing the framework from a huge repository on many Java programs for the Desktop version reveals that the proposed methodology is highly accurate in detecting code vulnerabilities.

2. Literature Review

For traditional programming utilizing k-means cluster analysis and the generative adversarial model, Terada and Watanobe [1] introduced to check bugs in large amount random codes. To select the optimal code with an interactive analysis framework and the software code refactor generation that k-means cluster transformation has been used. A system named Tree-LSTM was described by Tai et al. [2] on object-oriented code analysis, in which an LSTM network works as a tree structure. The model is tested by the tasks of conceptual relationship analysis based on coding pairs and identification of sentiments. Pedroni et al. [3] provided a review centered on the form of compilation communication of massive source code. It helps recognize errors for inexperienced developers and the steps need to be taken about source code mistakes. They investigated the form of a message previously often benefits based on the coding system and detect specific code snippets' vulnerability. Saito and Watanobe [4] describes an ideal learning capacity map, that has been suggested an identifying improvements highest distribution for language learning models. the system used the characteristics as input to the system for testing. Teshima [5] proposed a system software vulnerability detection method using LSTM. To evaluate the optimal exasperation and classification accuracy, the
number of neurons of LSTM was modified. The LSTM model obtains a realistic result for system software bug identification.

Through system software bug prediction models, and an attention-based RNN was introduced by Fan et al. [6]. The model performance measures were often the F-measure performance and the coefficient of determination. This model has strengthened the classification approach of the data files. Accordingly, the F-measure performance and the area under curve were 14% and 7% higher than the province versions. A software source code classification algorithm that utilizes Convolutional Neural Network (CNN) was suggested by Ohashi et al. [7]. Dependent on the specific algorithm throughout the code, the model segregates the source code. All reference codes are translated into a basic code configuration during the CNN classification process without even any parameters, features, phrases, etc. The reliability of the CNN model for identification is very strong. A hybrid strategy to optimize the mechanism of identification was suggested by Zhou et al. [8]. The strategies for data and text analytics were merged in this research investigation. In the beginning, the overview of the bug report was obtained by utilizing text analytics, and this was followed by its categorization into phases. Then, other frameworks and functions were collected and provided to machine learner at the second level. To association of these two steps, numerous data grafting strategies were used. Finally, the bug report has generated based on the supplied test dataset. Jin et al. [9] concluded with improved returns using the same sample and used it in a original assessment. Compared to a text and schema of the standard bug intensity report, the importance of understanding utilizing Naive Bayes (NB) was developed. The use of standard bugs and additional fields was not factored in other research. Accordingly, this same F-measures of both the Eclipse and Mozilla configurations were measured to be 80% and 83%. In order to characterize the defects into extreme as well as non-severe groups, various machine learning architectures such as NB, Linear Regression, Support Vector Machine (SVM), Random Forest (RF), including k-nearest Neighbours (K-NN) were implemented. The researchers indicated that perhaps the classifier's output depends mostly on subset and achieves reliability in the 75 to 83% range.

In their analysis of the NASA dataset, Goseva et al. [10] demonstrate both supervised as well as unsupervised classification techniques. In all approaches, the TF-IDF, TF and Bag of Word (BoW) occurrence feature vector techniques have been utilized to generate the feature vectors. In addition, numerous classification algorithms, like SVM, KNN, RF and Decision Tree, were used for unsupervised classification approach to classification model and anomalies detection process. The findings showed that the managed strategy worked for good than unsupervised manner. Kukkar et al. [11] suggested a technique to enhance the K-NN classification model's effectiveness by using TF-IDF with bi-gram matrix factorization methods. For testing and training results, the research assessed additional three fields with both the textual field. While using the F-score, the efficiency was calculated. The output was found to rely on the repository. The estimation of software defects is a useful approach to minimize testing costs for the software component. By supervised learning from fault datasets of the previous version, it can effectively classify defect-prone software modules. According to Tong et al. [12] It is possible to divide current SDP studies through four categories:
identification, correlation, guidelines of the mining organization and rating. First-class experiments use categorized architectures (also referred to as classifiers) as predictive models to classify software systems into deficiency classes and non-defect classes or different traumatic experiences of defects. The training discrepancy we concentrate on is focused on the analysis of the classification problem in this system. A significant number of tools have been proposed to solve challenging to understand. Most of these techniques are grouped into two categories: the amount of data and stage of the algorithm. The data augmentation methods primarily research the impact of adjusting classes' distribution to deal with extracted features. According to More et al. [13] implementing a preprocessing mechanism to rebalance classification performance is probably a positive approach. The primary benefit of research methods is that they are autonomous of the classification model. In addition, data level approaches could be easily implemented as application-level approaches in distributed machine learning. Numerous existing researchers have used much soft computing and machine learning approaches in these frameworks to predict failure structures and lower software construction and preservation time. The neural network model is the most common among them. Most software estimation approaches make designs using measurements and defective data from an earlier implementation or similar artifacts and then use the method to forecast whether defects are currently being developed by Rawat et al. [14] throughout the modules, called an ensemble learning methodology. For JavaScript use a hybrid, compressed analysis, Wei et al. [15] introduced blended taint evaluation. The collection of data was performed for even those difficult conditions to dynamically evaluate by implementing a static response. A static network, which also incorporates a request builder, is perpetuated with complex results. The formidable challenge calls are used in this call-graph builder module. In pure topology optimization, they caught up with the WALA instrument to create a static call-graph. The proposed method works similarly as previously stated and encourages external call diagram builder resources to be used in the research flow.

To determine the essential elements likely to be caused by a shift in technology, Musco et al. [16] utilized three kinds of call-graphs. However, to determine the effect of a shift in the software, they used feedback on different. The same approach should be used, and with a small modification, it may add a vulnerability or mitigate a vulnerabilities transformation rather than using an unpredictable change. Munaiah et al. [17] through the method, "Proximity" but "Dangerous Walks" metrics have presented new novel patch management measurements. The call-graph interpretation of the system distinguishes all of these. Their current studies have shown that utilizing their measurements to construct a forecasting model can help forecast more accurately because their measurements are statistically linked to susceptible components. A novel paradigm to detecting influence weaknesses called VGDetector has been developed by Cheng et al. [18]. In a combined analysis, the latest graph fully convolutional neural network framework is used to embed source code.

The fully automated way to map weaknesses to development tools and an interpersonal therapy Vulture was introduced by Neuhaus et al. [19] which can generate determinants in a new module to predict weaknesses. They found that dependencies and calls for features affect whether or not a part is susceptible. They
also performed a Mozilla compiled code assessment that revealed that their methodology is precise. Lee et al. [20] proposed a method to produce semantic fingerprints from intrusion prevention programs. The call-graph, termed the software graph, was retrieved by both the API call pattern created by malicious. For the conceptual signature, this diagram is used. Even when the malware is confused, or the brief software moment from its older iterations, semantic fingerprints allow users to upload.

Punia et al. [21] proposed a request algorithm to estimate and improve survival in a specified number, is probably the most comparable to our research. The connection between their measurements and many forms of bugs has been examined. It is observed that there is a connection in software engineering between call-graph dependent measurements and bugs. The frequency response machine learning methods were compared to J84 and SMO, and tested deep learning methods to identify possible bugs in the program. It has also concentrated on various source code metrics and also introduced pairing metrics for a so composite request. Pu et al. [22] suggested a source code corrections approach based on LSTM by utilizing returns generated similarities. For both the code major revision, the research utilized the pattern (seq2seq) neural network architecture with machine learning activities. White et al. [23] suggested a deep RNNs-based computer language model. The extensive experiments have shown that in a Java repository, the model exceeds conventional transfer learning called n-gram and directory listing n-gram. The software learning algorithm in the context of software development looks very promising. Terada et al. [24] proposed an LSTM-based methodology for software training in which the framework, by analyzing unfinished, raw data, predicts the following term. When developing a complete program, software engineers usually begin the process from the very start. The model suggests the next term finish a program in order to support everyone. The Classifier model achieved a high level of predictive power.

In summary, several promising approaches have been suggested. As classification algorithms for web application bug identification and vulnerability classification, most investigators used conventional semi-supervised classifiers, RNNs, LSTM, or CNNs. RNNs are far better than the standard language models, including such n-gram, but they have drawbacks in charge of understanding long sequence data. LSTM is an RNN version that transcends the limitations of RNNs. The model described in the present study blends the process of awareness with LSTM (LSTM-RNN). The LSTM-RNN network is used as a feature vector for source code evaluation and identification based on the predicted upper bound. The LSTM network is surpassed by the LSTM-AttM network, despite the fact that the former uses only estimates based on the results of the most recent hidden state. In comparison, for prediction, LSTM-RNN considers all previously hidden state effects. Based on faults, functional programming identification, archive code identification, and basic error detection, almost all of the studies used various system software, classification models. The proposed scheme explicitly defines logic, grammar, and other system software errors. Additionally, in place of the error spot, the model is used to predict the correct terms.
3. Methodology

3.1 Proposed Architecture

A system overview describing the flow of the execution procedure is displayed in the following figure 1. In the initial stages, a dataset consisting of many different software codes is taken into consideration. These programs comprise a wide range of processes and functions. The data set has been handled by Natural Language Processing using certain basic methods, including tokenization, which has resulted in the data being separated into individual words. The Porter stemmer algorithm was utilized in order to extract features, and ultimately, a filtration strategy was applied in order to get rid of instances that had been classified incorrectly or null values.

![Figure 1. Proposed System Architecture Design](image-url)
The TF-IDF features have been extracted by considering the density of relevant tokens; this method for obtaining features is employed in both the training and testing phases, correspondingly. The vector space model has developed with the objective of feature selection and enhancing with data acquisition in order to obtain the best feature possible from the model’s vector space. There are three distinct machine learning algorithms that have been demonstrated during the training and testing. After the training has been completed, the system will begin to generate some background knowledge in accordance with a method of supervised learning. Utilizing this method, it is possible to test data sets on a variety of platforms and improve detection accuracy for different types of data. For detecting the bug across the full dataset RNN classifier is utilized.

3.2 Algorithm Design

The algorithm which is described below have been used throughout the process of features extraction using natural language processing. However, the complete set of NLP features was insufficient for classification, thus these algorithms also generated the train modules. Certain attributes are extracted by utilizing the following algorithm, while certain NLP features provide an enhanced vector space for the selected features.

**Algorithm Design**

The algorithm which is described below have been used throughout the process of features extraction using natural language processing. However, the complete set of NLP features was insufficient for classification, thus these algorithms also generated the train modules. Certain attributes are extracted by utilizing the following algorithm, while certain NLP features provide an enhanced vector space for the selected features.

**Input:** Code line snippet which comprises of Term[1…..m]

**Output:** TF-IDF weight for every Term

**Steps:**

1. Data_Vec = \{Tk1, Tk2, Tk3… Tkm\}

2. for every (Tr into Data_Vector)
   
3. Term_Frequency tf (tr,doc) = (tr,doc)

4. \( idf = \text{tr} \rightarrow \sum(\text{doc}) \)

5. return \( tf \times idf \)

**Figure 2. Term Frequency and Inverse Document Frequency**

4. Result and Discussion

The RNN classifier is used for fault anticipating which comprises unlabeled datasets, to validate the assessment of the recommended bug forecast method. The confusion matrix, which is displayed in Table 1, is used as a basis for the performance analyses of software flaw prediction.
Table 1. Confusion matrix evaluation

<table>
<thead>
<tr>
<th>Confusion matrix</th>
<th>Predicted</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Positive</td>
</tr>
<tr>
<td>Actual Positive</td>
<td>True Positive (TP)</td>
</tr>
<tr>
<td>Actual Negative</td>
<td>False Positive (FP)</td>
</tr>
</tbody>
</table>

TN: It provides negative prediction for actual negative classes.

FP: It provides positive as prediction to every negative classes.

FN: It provides negative to every positive classes.

TP: It provides positive prediction for positive classes.

The accuracy of predictions is measured as the proportion of times the prediction was correct out of the total number of instances it was utilized.

\[
\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}
\]  

The performance evaluation of the proposed system was carried out utilizing the proposed classification strategies with ten different synthetic datasets. The following Table 2, provides an illustration of the number of instances of bugs that have been accurately identified in each individual dataset, together with the proportion of bug records that are present in each dataset.

Table 2. Analysis of Bug Accuracy Rate

<table>
<thead>
<tr>
<th>Dataset Name</th>
<th>Bug Accuracy Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intercafe [1]</td>
<td>77.80%</td>
</tr>
<tr>
<td>Synapse-1.0 [2]</td>
<td>66.45%</td>
</tr>
<tr>
<td>Berek [5]</td>
<td>88.18%</td>
</tr>
<tr>
<td>Intercafe [7]</td>
<td>80.68%</td>
</tr>
<tr>
<td>Termo [9]</td>
<td>77.5%</td>
</tr>
<tr>
<td>Pbean2 [13]</td>
<td>83.46%</td>
</tr>
<tr>
<td>Safe [16]</td>
<td>88.07%</td>
</tr>
<tr>
<td>Apache [18]</td>
<td>86.92%</td>
</tr>
<tr>
<td>Forrest-0.8 [19]</td>
<td>92.12%</td>
</tr>
<tr>
<td>Camel1.6 [21]</td>
<td>90.91%</td>
</tr>
</tbody>
</table>
It is observed from the experimental findings that the bug predictive performance using RNN classifier is best. Due to the limitations imposed by the data size, several datasets fail to adequately capture actual software problems and new bugs. When implemented in actual software programs, the approach that has been offered can be evaluated with additional testing.
5. Conclusion and Recommendation

The identification of vulnerabilities in imbalanced source codes is an exhausting and time-consuming task. Vulnerable code makes it possible to execute software attacks on remote users. At times, susceptible code develops internal attacks such as session hijacking, buffer overflow, bypass authorization, and so on while it is being executed. Sometimes flaws manifest themselves in subtle ways that software testers are unable to uncover or opt to overlook. Constructing predictive models to detect the vulnerability is a common application of machine learning methods, which are a prominent way. The proposed methodology demonstrates the data-driven approach to identify flaws through the application of deep learning technique along with a potential fix. The proposed RNN is utilized to develop a code vulnerability evaluation and bug cleanup. The system is capable of working with a variety of datasets to acquire features and identify vulnerabilities. RNN provides a classification that is superior to that provided by conventional ML classifiers. A further need in software engineering is clone detection and recommendation to improve code quality. When designing software, a high level of design excellence can be reached with the support of bugs-free code clones.

References


