

ABSTRACT

Software vulnerabilities are raising the security risks. If any vulnerability is oppressed due to a malicious attack, it will compromise the system's safety. In addition, it may create catastrophic losses. So, automatic classification methods are required to manage vulnerability in software, and then security performance of the system will be improved. It will also mitigate the risk of system being attacked and damaged. In this project, a new model has been proposed with name automatic vulnerability classification model (IGTF-DNN) Information Gain based on Term Frequency - Deep Neural Network.

The model is generated using information gain (IG) which is based on frequency-inverse document frequency (TF- IDF), and deep neural network (DNN): TF- IDF is used to calculate frequency/weight of words taken from vulnerability description; IG is used to select features to gather optimal set of feature words. Then neural network model is used to construct an automatic vulnerability classifier to achieve effective vulnerability classification.

The National Vulnerability Database of the United States has been taken to test this new model's effectiveness. By comparing with KNN, this TFI-DNN model has achieved better performance in evaluation. The project is designed using R Studio 1.0 and language R 3.4.4.

TABLE OF CONTENTS

CHAPTER NO.	TITLE	PAGE NO.
	ABSTRACT	iv
	LIST OF TABLES	vii
	LIST OF FIGURES	viii
	LIST OF ABBREVIATIONS	ix
1	INTRODUCTION	1
	1.1. Introduction	1
	1.2 Objectives	3
2	LITERATURE SURVEY	4
3	SYSTEM ANALYSIS	20
	3.1 Existing System	20
	3.1.1 Drawbacks	20
	3.2 Proposed System	21
	3.2.1 Advantages	21
	3.3 Feasibility Study	21
	3.3.1 Economical Feasibility	22
	3.3.2 Operational Feasibility	22
	3.3.3 Technical Feasibility	22
4	SYSTEM SPECIFICATION	23
	4.1 Hardware Requirements	23
	4.2 Software Requirements	23
5	SOFTWARE DESCRIPTION	24
	5.1 Front End	24
	5.1.1 Features of R Language	24
	5.1.2 Statistical Functions	25
	5.1.3 Interfaces	25
	5.1.4 Basic Features Of R	26

CHAPTER NO.	TITLE	PAGE NO
	5.1.5 Free Software	27
	5.1.6 R Studio	27
6	PROJECT DESCRIPTION	28
	6.1 ProblemDefinition	28
	6.2 Overview ofthe project	29
	6.3 Module Description	30
	6.3.1 Feature word extraction	31
	6.3.2.TF-IDF(Term Frequency/InverseDocu -mentFrequency)	31
	6.3.3. Information Gain (IG)	33
	6.3.4. Optimizations using DNN	34
	6.4 Input Design	35
	6.5 Output Design	35
	6.6 SystemFlow Diagram	36
	6.7 User Case Diagram	37
	6.8 Sequence Diagram	38
7	CONCLUSION	39
	APPENDIX	40
	A. Source Code	40
	B. Screen Shots	55
	REFERENCES	62

LIST OF TABLES

TABLE .NO.	TABLE NAME	PAGE NO.
1	TF-IDF FEATURES	32
2	INFORMATION GAIN (IG)	34

LIST OF FIGURES

FIGNO	FIG NAME	PAGE NO .
6.1	TFI-DNN ALGORITHM	28
6.6	SYSTEM FLOW DIAGRAM	36
6.7	USE CASE DIAGRAM	37
6.8	SEQUENCE DIAGRAM	38

LIST OF ABBREVIATIONS

NN	Neural Network
AI	Artificial Intelligence
ML	Machine Learning
DNN	Deep Neural Network
IG	Information Gain
APT	Advanced Persistent Threat

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

Due to information technology's fast development, its impacts to industries by application of the Internet and computers are powerful. Not only, they brought convenience, but also huge risks and hidden dangers at the same time. With the improvement of digitalization level of industries, information security problems have become increasingly outstanding. Vulnerabilities are nothing but software/hardware defects of system being illegitimately exploitable made by unauthorized people.

As soon as vulnerability of information system is exploited by suspicious attack, the information system's security will be at great risk. It may even create inestimable consequences. In 2017, Windows system vulnerabilities are exploited by hackers to expose 100,000 organizations around the world to Bitcoin ransomware. Again in the same year, Microsoft released a total of 372 vulnerability patches for Office.

Hackers make use of office vulnerabilities to conduct Advanced Persistent Threat (APT) attacks, spread ransomware, botnets and so on. Nowadays, the count and variety of vulnerabilities are gradually increasing, so that the analysis and management of software vulnerabilities are becoming more important.

If the vulnerability can be classified and managed with effectiveness, it may not only enhance the efficiency of vulnerability recovery and management, but also diminish the risk of systems being attacked and collapsed, which is crucially important for security performance of systems. As software security vulnerabilities play a major role in cyber-security assault, more and more researches on vulnerability classification are conducted by applicable security researchers.

The earlier vulnerability classification method RISOS [1], is aimed at the operating system of computer, mainly segments the OS vulnerabilities into 7 categories from the attack perspective, and elaborates how to exploit vulnerabilities instead of triggering the vulnerabilities' situation. The PA vulnerability classification method in [2] not only studies the operating system vulnerabilities, but also classifies the vulnerabilities already present in the application.

Andy Gray vulnerability classification method [3] introduced a vulnerability classification system consisting of ten categories according to the various analysis needs of vulnerability. As the complexity of vulnerabilities increases, limitations of traditional artificial vulnerability classification methods become clearer.

Therefore, researchers give more attention to vulnerabilities' automatic classification. Recently, a large number of machine learning methods have been reported in text classification field [4]. Classifying data by vulnerability description is a kind of text classification.

Therefore, the automatic classification of vulnerabilities problems can be solved using machine learning methods. Shua et al. [5] applied the SVM classification method based on LDA model in vulnerability classification domain.

The SVM based upon topic model makes full use of number of distributed vulnerabilities for classification. The experiment results indicated that SVM has attained good results in vulnerability grouping.

Wijayasekara et al. [6] evaluated Naïve Bayes classification method by using textual information from error descriptions. That analysis illustrated the feasibility of Naïve Bayes classifier for classifying textual information based on vulnerability description.

Sarang et al. [7] introduced a classification method to classify CVE entries which could not give enough information into vulnerability groups using Naïve Bayes classifier. Gawron et al. [8] pertained Naïve Bayes algorithm and simplified artificial neural network (ANN) algorithm for vulnerability classification, and thereby made comparison on same data set. The experimental

results verified that artificial neural network algorithm was finer to Naive Bayes algorithm in vulnerability classification.

1.2 OBJECTIVES

The main objectives of the Vulnerability Classification are

- Using TF-IDF to calculate the frequency and weight of each word from vulnerability description.
- Using IG to select features to obtain an optimal set of feature word.
- Using neural network model to construct an automatic vulnerability classifier to achieve effective vulnerability classification

To achieve the main goal, the specific objectives required are

- To extract vulnerability information from the records using program codes written in R including vulnerability text description, and vulnerability type (category).
- To collect some text information of vulnerabilities from 2015 to 2019 for statistics, 500 of them were used for training data set and 100 for test data set.
- To take Excel work sheet records from 5 excel work sheets for year 2015, 2016, 2017, 2018 and 2019 and store as data frame objects. Then category array is filled with unique values of categories from Vulnerability type column.

CHAPTER 2

LITERATURE REVIEW

2.1 CONTINUOUS INTEGRATION, DELIVERY AND DEPLOYMENT: A SYSTEMATIC REVIEW ON APPROACHES, TOOLS, CHALLENGES AND PRACTICES

MOJTABA SHAHINA, MUHAMMAD ALI BABAR ALI MING ZHU

In this paper [1] the authors stated that continuous practices, i.e., continuous integration, delivery, and deployment, are the software development industry practices that enable organizations to frequently and reliably release new features and products. With the increasing interest in and literature on continuous practices, it is important to systematically review and synthesize the approaches, tools, challenges, and practices reported for adopting and implementing continuous practices.

Objective: This research aimed at systematically reviewing the state of the art of continuous practices to classify approaches and tools, identify challenges and practices in this regard, and identify the gaps for future research. **Method:** They used systematic literature review (SLR) method for reviewing the peer-reviewed papers on continuous practices published between 2004 and 1st June 2016. We applied thematic analysis method for analysing the data extracted from reviewing 69 papers selected using predefined criteria. **Results:** We have identified thirty approaches and associated tools, which facilitate the implementation of continuous practices in the following ways:

- reducing build and test time in continuous integration (CI)
- increasing visibility and awareness on build and test results in CI;
- supporting (semi-) automated continuous test
- detecting violations, flaws and faults in CI;
- addressing security and scalability issues in deployment pipeline, and

improving dependability and reliability of deployment process. They have also determined a list of critical factors such as testing (effort and time), team awareness and transparency, good design principles customer highly skilled and motivated team, application domain, and appropriate infrastructure that should be carefully considered when introducing continuous practices in a given organization. The majority of the reviewed papers were validation (34.7%) and evaluation (36.2%) research types. This review also reveals that continuous practices have been successfully applied to both greenfield and maintenance projects.

Continuous practices have become an important area of software engineering research and practice. Whilst the reported approaches, tools, and practices are addressing a wide range of challenges, there are several challenges and gaps which require future research work for: improving the capturing and reporting of contextual information in the studies reporting different aspects of continuous practices; gaining a deep understanding of how software-intensive systems should be (re-) architected to support continuous practices; addressing the lack of knowledge and tools for engineering processes of designing and running secure deployment pipelines.

With increasing competition in software market, organizations pay significant attention and allocate resources to develop and deliver high-quality software at much accelerated pace [5]. Continuous Integration (CI), Continuous DElivery (CDE), and Continuous Deployment (CD), called continuous practices for this study, are some of the practices aimed at helping organisations to accelerate their development and delivery of software features without compromising quality [6]. Whilst CI advocates integrating work-in-progress multiple times per day, CDE and CD are about ability to quickly and reliably release values to customers by bringing automation support as much as possible [7-10].

Continuous practices are expected to provide several benefits such as: (1) getting more and quick feedback from the software development process and customers; (2) having frequent and reliable releases, which lead to improved customer satisfaction and product quality; through CD, the connection between development and operations teams is strengthened and manual tasks can be eliminated. A growing number of industrial cases indicate that the continuous practices are making inroad in software development industrial practices across various domains and sizes of organizations.

At the same time, adopting continuous practices is not a trivial task since organizational processes, practices, and tool may not be ready to support the highly complex and challenging nature of these practices. Due to the growing importance of continuous practices, an increasing amount of literature describing approaches, tools, practices, and challenges has been published through diverse venues. An evidence for this trend is the existence of five secondary studies on CI, rapid release, CDE and CD. These practices are highly correlated and intertwined, in which distinguishing these practices are sometimes hard and their meanings highly depends on how a given organization interprets and employs them. Whilst CI is considered the first step towards adopting CDE practice, truly implementing CDE practice is necessary to support automatically and continuously deploying software to production or customer environments (i.e., CD practice).

They noticed that there was no dedicated effort to systematically analyze and rigorously synthesize the literature on continuous practices in an integrated manner. By integrated manner they meant simultaneously investigating approaches, tools, challenges, and practices of CI, CDE, and CD, which aims to explore and understand the relationship between them and what steps should be followed to successfully and smoothly move from one practice to another. This study aimed at filling that gap by conducting a Systematic Literature Review (SLR) of the approaches, tools, challenges and practices for adopting and implementing continuous practices.

This SLR provides an in-depth understanding of the challenges of adopting continuous practices and the strategies (e.g., tools) used to address the challenges. Such an understanding is expected to help identify the areas where methodological and tool support to be improved. This increases the efficacy of continuous practices for different types of organizations and software- intensive applications.

Moreover, the findings are expected to be used as guidelines for practitioners to become more aware of the approaches, tools, challenges and implement appropriate practices that suit their industrial arrangements. For this review, they have systematically identified and rigorously reviewed 69 relevant papers and synthesized the data extracted from those papers in order to answer a set of research questions that motivated this review.

The significant contributions of this work are:

- A classification of the reported approaches, associated tools, and challenges and practices of continuous practices in an easily accessible format.
- A list of critical factors that should be carefully considered when implementing continuous practices in both software development and customer organizations.
- An evidence-based guide to select appropriate approaches, tools and practices based on the required suitability for different contexts.
- A list of researchable issues to direct the future research efforts for advancing the state-of-the-art of continuous practices.

This work has presented a Systematic Literature Review (SLR) of approaches, tools, challenges and practices identified in empirical studies on continuous practices in order to provide an evidential body of knowledge about the state of the art of continuous practices and the potential areas of research. They selected 69 papers from 2004 to 1st June 2016 for data extraction, analysis, and synthesis based on predefined inclusion and exclusion criteria. A rigorous analysis and systematic synthesis of the data extracted from the 69 papers have enabled us to conclude: The research on continuous practices, in particular continuous delivery and deployment is gaining increasing interest and attention from software engineering researchers and practitioners according to the steady upward trend in the number of papers on continuous practices in the last decade.

2.2 BLACK BOX EVALUATION OF WEB APPLICATION SCANNERS: STANDARDS MAPPING APPROACH

MALIK QASAIMEH, ALA'ASHAMLAWI, TARIQ KHAIRALLA

The Secure Development Life Cycle (SDLC) of web applications aims to enhance the quality attributes of released applications. Security is among of the important attributes during the penetration testing phase. Web Application Vulnerability Scanners (WAVS) help the developers to identify existing vulnerabilities that could compromise the security and privacy of data exchanged between the client and web server during the development and deployment phases.

scanning software for 2017. The method of black box testing was adopted to evaluate the five WAVSs against seven vulnerable web applications. The evaluation is based on different measures such as the vulnerabilities severity level, types of detected vulnerabilities, numbers of false positive vulnerabilities and the accuracy of each scanner.

The evaluation is conducted based on an extracted list of vulnerabilities from OWASP and NIST. The accuracy of each scanner was measured based on the identification of true and false positives.

Secure software development aims to activate security development early during the software development life cycle. In the era of internet of things (IoT) it is expected that web applications will be increasingly integrated with sensor-based systems to provide vital services (e.g. smart homes and cities, transportation and logistics and healthcare services) and web-based applications requiring users and sensors to input critical data in order to complete certain transactions to enable smart services.

Security measures should be properly configured in order to secure the information exchanged over the web application as well as ensuring the security of the hosting web server. It is said that “the best defense against cyber-attacks is a good offense”, which is achieved by testing the web application for any possible vulnerabilities during the verification phase, as described by different Software Security Development Lifecycle (SSDL) institutions, such as Microsoft’s Security Development Lifecycle (SDL), Touch Points and Comprehensive Lightweight Application Security Process (CLASP). Vulnerabilities scanning tools are useful during penetration testing in order to reduce opportunities to exploit potential vulnerabilities early during the SSDL.

WAVSs are defined as automated programs that examine web applications for potential security vulnerabilities such as Cross-Site Scripting (XSS), SQL Injection, directory traversal, insecure configurations, and remote command execution vulnerabilities [7, 8]. Most web application vulnerability scanners classify vulnerabilities into four categories: High, Medium, Low, and Informational. Some scanners also consider Critical as a category. These categories are described below

High: A vulnerability which when exploited allows attackers to take complete control of the web application and server. It allows attackers to access the application's database, modify accounts, and steal sensitive information. XSS and SQL injection are examples of high severity vulnerabilities which should have the utmost priority for fixing if detected by a scanner.

Medium: A vulnerability which when exploited allows attackers to access a logged-in user account to view sensitive content. It allows attackers access to information that helps them exploit other vulnerabilities, or better understand the system so they can refine their attacks. Open redirection is an example of a medium severity vulnerability which allows an attacker to redirect a user to a malicious website. Medium severity vulnerabilities should be addressed at the earliest possible opportunity if detected by a scanner.

Low: A vulnerability that has a minimal impact or cannot be exploited by an attacker. Cookies not marked as Http Only is an example of a low severity vulnerability. Marking Cookies as Http Only makes the cookie unreadable by client-side scripts and hence provides an additional layer of protection against XSS attacks. Low severity vulnerabilities are worth investigating and fixing if the time and budget allows.

Informational: These are not considered vulnerabilities, rather they are alerts that provide information about the web application. Examples include NTLM Authorization Required and Database Detected (MySQL). No action is needed for these informational alerts. Black box and white box testing are the main approaches for the security evaluation of web applications. White box testing studies the internal structures of applications, however web applications usually consist of multiple technologies and programming languages, including the client- and server- side languages, therefore the such testing may fail to capture all security flaws and vulnerabilities due to the code complexity. Black box testing, also known as dynamic security evaluation, includes an analysis of the application execution under a certain conditions and inputs (i.e. functions) to identify possible vulnerabilities. This approach, also known as penetration testing, consists of the following phases:

Crawling and identifications: in this phase the scanner operations include browsing all possible links and web directories. One of the main challenges in this phase is crawling pages that require human input, such as user passwords. The main objective of this phase is to obtain the HTML format of all server replies. By the end of this phase the scanner identifies the entry points that require special input, such as the username and password. The forms and functions such as the GET and POST are also identified in this phase. The scanner should also be able to identify the application structure and functionality to extract information that will be useful in the next phase. Parsing and attack: in this phase the scanner generates or uses already existing data from directories expected to match the real data input required by the web application. Fuzz testing is deployed in this phase to generate and submit data of different sizes. Some scanners use malicious input patterns to identify the possible vulnerable response from the web server. The identified actions from the crawling phase along with input filled are sent to the server to observe its reply.

Analysis: In this phase the scanner analyzes the server response, which is generally influenced by the data submitted. The scanner also classifies the server response and observes the valid ones. The scanners list errors that help in the classification of possible vulnerabilities. For example, if an error was related to XSS, the scanner concludes that XSS vulnerability may exist.

They concluded that the SDLC of web applications consists of many activities that aim to enhance the overall applications' robustness against different types of attack. Black box testing is an important activity used during SDLC to enhance the security of web applications. Developers and penetration testers utilize WAVSs to dynamically scan web applications and investigate any possible vulnerability before it can threaten client services.

Currently, many WAVSs are proposed and they have different capabilities in detecting the true positive vulnerabilities. In this paper, five WAVSs were evaluated by scanning seven intentionally vulnerable web applications. The selected WAVs are among the top ten scanners for the year of 2017. A mapping approach was developed to extract a list of vulnerabilities based on NIST and OWASP standards. The scanners were evaluated using different measures, such as the vulnerabilities severity level, types of detected vulnerabilities, numbers of false positive vulnerabilities and the accuracy of each scanner.

2.3AUTOMATIC CLASSIFICATION METHOD FOR SOFTWARE VULNERABILITY BASED ON DEEP NEURAL NETWORK

**GUOYAN HUANG, YAZHOU LI, QIAN WANG, JIADONG REN,
YONGQIANGCHENG,XIAOLIN ZHAO,**

In this paper [3] the authors stated that software vulnerabilities are the root causes of various security risks. Once a vulnerability is exploited by malicious attacks, it will greatly compromise the safety of the system, and may even cause catastrophic losses. Hence automatic classification methods are desirable to effectively manage the vulnerability in software, improve the security performance of the system and reduce the risk of the system being attacked and damaged.

In this paper, a new automatic vulnerability classification model (TFI-DNN) has been proposed. The model is built upon term frequency- inverse document frequency (TF-IDF), information gain(IG) and deep neural network (DNN): the TF-IDF is used to calculate the frequency and weight of each word from vulnerability description; the IG is used for feature selection to obtain an optimal set of feature word; and the DNN neural network model is used to construct an automatic vulnerability classifier to achieve effective vulnerability classification. The National Vulnerability Database (NVD) of the United States has been used to validate the effectiveness of the proposed model. Compared to SVM, Naive Bayes and KNN, the TFI-DNN model has achieved better performance in multi-dimensional evaluation indexes including accuracy, recall rate, precision and F1-score.

With the rapid development of information technology, the impacts brought to industries by application of the Internet and computers are twofold. They bring convenience but also huge risks and hidden dangers. Recently, with the improvement of the digitalization level of various industries, information security issues have become increasingly prominent. Vulnerabilities are defined as software and hardware defects of the system being illegally exploitable by unauthorized personnel. Once the vulnerability of information system is exploited by malicious attack, the security of information system will be at great risk and

May cause inestimable consequences. For example, in 2017, hackers exploited Windows system vulnerabilities to expose 100,000 organizations worldwide to Bitcoin ransomware.

In the same year, Microsoft released a total of 372 Office vulnerability patches. Hackers can use the office vulnerabilities to conduct Advanced Persistent Threat (APT) attacks, spread botnets, ransomware and so on. In recent years, the number and variety of vulnerabilities have gradually increased, so the management and analysis of software vulnerabilities has become more and more important.

If the vulnerability can be classified and managed effectively, it can not only improve the efficiency of vulnerability recovery and management, but also reduce the risk of system being attacked and damaged, which is vitally important for the security performance of the system. As software security vulnerabilities play an important role in cyber security attacks, more and more researches on vulnerability classification are conducted by relevant security researchers. The earlier vulnerability classification method RISOS, which is aimed at the operating system of the computer, mainly divides the operating system vulnerabilities into seven categories from the perspective of attack, and describes how to exploit the vulnerabilities instead of triggering these vulnerabilities' condition. The PA vulnerability classification method not only studied the vulnerabilities of the operating system, but also classified the vulnerabilities existing in the application. Andy Gray vulnerability classification method proposed a vulnerability classification system consisting of ten categories according to the different analysis needs of the vulnerability.

However, as the complexity of vulnerabilities increases, the limitations of traditional artificial vulnerability classification methods become more and more obvious. Therefore, researchers pay more attention to automatic classification of vulnerabilities. A large number of machine learning methods have been recently reported in the field of text classification. Classifying them by vulnerability description is also a kind of text classification.

Therefore, the problem of automatic classification of vulnerabilities can also be solved using machine learning methods. Davari M et al. proposed an automatic vulnerability classification framework based on conditions that activate vulnerabilities, different machine

learning techniques (Random Forest, C4.5 Decision Tree, Logistic Regression, and Naive Bayes) are employed to construct a classifier with the highest F-measure. The 580 software security flaws of the Firefox project were analyzed through experiments to evaluate the effectiveness of the classification.

The SVM classification method based on LDA model is applied in the domain of vulnerability classification by Bo Shuai et al. The SVM based on the topic model can make full use of the number of distributed vulnerabilities for classification. The experiment results indicated that SVM has achieved good results in vulnerability classification. Dumidu Wijayasekara et al. tested the Naïve Bayes classifier by using textual information from the error description. The analysis illustrates the feasibility of the Naïve Bayes classifier to classify textual information based on the vulnerability description. Sarang Na et al. proposed a classification method for classifying CVE entries that could not provide enough information into vulnerability categories using the Naïve Bayes classifier. Marian Gawron et al. applied Naive Bayes algorithm and simplify artificial neural network algorithm to vulnerability classification, and made comparison on the same data set. The experimental results showed that the artificial neural network algorithm was superior to Naive Bayes algorithm in vulnerability classification.

A data-driven approach to machine learning for vulnerability detection was proposed by Harer J A et al. They also compare the application of deep neural network models with more traditional models such as random forests, and find that the best performance comes from combining the features of deep model learning with tree-based models. Finally, the highest performance model proposed in the paper achieves an area under the precision-recall curve of 0.49 and an area under the ROC curve of 0.87. It can be seen that a large number of machine learning algorithms have been well applied in the field of vulnerability classification. Although these machine learning classification algorithms have achieved promising results in many fields, due to the large amount of vulnerability data and short description, the generated word vector space presents the characteristics of high dimension and sparse. These machine learning algorithms are not very effective in dealing with high and sparse problems.

At the same time, they ignore specific vulnerability information and the classification accuracy is not high. However, in recent years, deep learning has been applied in many fields

and has achieved success, such as the field of speech and image recognition, the error rate in speech recognition is reduced by 20%-30%, and the error rate in the Image Net evaluation task is reduced by 26%-15%. For face image classification based on GoogLe Net network, The classification of age and gender is achieved with high precision. Deep learning also has a significant impact in the field of natural language. Hwiyeol et al. studied the classification problem in the field of natural language, and applied convolution neural networks (CNN) and recurrent neural networks (RNN) to the field of large-scale text classification and achieved success. Aziguli W et al. proposed a novel text classifier using DNN model to improve the computational performance of processing large text data with mixed outliers.

Wu Fetal. proposed a deep learning method for vulnerability detection [22], namely, convolution neural network (CNN), long short term memory (LSTM) and convolution neural network- long short term memory (CNN- LSTM), and the vulnerability prediction accuracy reached 83.6%, which is superior to the traditional method. Li Z et al. designed and implemented a deep learning-based vulnerability detection system, which alleviated the cumbersome and subjective tasks of human experts manually defining features.

Therefore, deep learning can also be applied to the field of software vulnerability detection and achieved good results. Therefore, in order to better deal with the high and sparse word vector space and take advantage of the automatic feature extraction by deep learning, this paper proposes an automatic vulnerability classification model TFI-DNN based on term frequency-inverse document frequency (TF- IDF), information gain (IG) and deep neural network (DNN).

In the model, they first used TF-IDF-IG (TFI) algorithm to extract the feature of the description text and reduce the dimension of the generated high-dimensional word vector space, then construct a DNN neural network model based on deep learning. The TFI-DNN model was trained and tested using vulnerability data from the National Vulnerability Database (NVD). The test results show that the automatic vulnerability classification model in this paper effectively improves the performance of vulnerability classification.

They concluded that in order to better analyze and manage vulnerabilities according to their belonging classes, improve the security performance of the system, and reduce the risk of the system being attacked and damaged, this paper applied deep neural network to software vulnerability classification. The analysis of the method and construction process of TFI and DNN are discussed in detail. We compared the vulnerability classification model TFI-DNN to TFI-SVM, TFI-Naïve Bayes and TFI-KNN on the NVD dataset.

The results show that the proposed TFI-DNN model outperforms others in accuracy, precision and F1-score and performs well in recall rate. And it is superior to SVM, Naïve Bayes and KNN on comprehensive evaluation indexes. The work in this paper shows the effectiveness of TFI-DNN in vulnerability classification, and provides a basis for their future research using the benchmark vulnerability dataset.

2.4 GENERALIZED VULNERABILITY EXTRAPOLATION USING ABSTRACT SYNTAX TREES

FABIAN YAMAGUCH, MARKUS LOTTMANN, KONRAD RIECK.

In this paper [4] the authors stated that the discovery of vulnerabilities in source code is a key for securing computer systems. While specific types of security flaws can be identified automatically, in the general case the process of finding vulnerabilities cannot be automated and vulnerabilities are mainly discovered by manual analysis. In this paper, they proposed a method for assisting a security analyst during auditing of source code.

Their method proceeds by extracting abstract syntax trees from the code and determining structural patterns in these trees, such that each function in the code can be described as a mixture of these patterns. This representation enables us to decompose a known vulnerability and extrapolate it to a code base, such that functions potentially suffering from the same flaw can be suggested to the analyst. We evaluate our method on the source code of four popular open-source projects: LibTIFF, FFmpeg, Pidgin and Asterisk. For three of these projects, they were able to identify zero-day vulnerabilities by inspecting only a small fraction of the code bases.

The security of computer systems critically depends on the quality of its underlying code. Even minor flaws in a code base can severely undermine the security of a computer system and make it an easy victim for attackers. There exist several examples of vulnerabilities that

have led to security incidents and the proliferation of malicious code in the past. A drastic case is the malware Stuxnet that featured code for exploiting four unknown vulnerabilities in the Windows operating system, rendering conventional defense techniques ineffective in practice. The discovery of vulnerabilities in source code is a central issue of computer security. Unfortunately, the process of finding vulnerabilities cannot be automated in the general case. According to Rice’s theorem a computer program is unable to generally decide whether another program contains vulnerable code.

Consequently, security research has focused on devising methods for identifying specific types of vulnerabilities. Several approaches have been proposed that statically identify patterns of specific vulnerabilities, such as the use of certain insecure functions. Moreover, concepts from the area of software verification have been successfully adapted for tracking vulnerabilities, for example, in form of fuzz testing, taint analysis and symbolic execution. Many of these approaches, however, are limited to specific conditions and types of vulnerabilities. The discovery of vulnerabilities in practice still mainly rests on tedious manual auditing that requires considerable time and expertise.

In this paper, they proposed a method for assisting a security analyst during auditing of source code. Instead of striving for an automated solution, we aim at rendering manual auditing more effective by guiding the search for vulnerabilities. Based on the idea of vulnerability extrapolation, their method proceeds by extracting abstract syntax trees from the source code and determining structural patterns in these trees, such that each function in the code can be described as a mixture of the extracted patterns. The patterns contain sub trees with nodes corresponding to types, functions and syntactical constructs of the code base. This representation enables our method to decompose a known vulnerability and to suggest code with similar properties—potentially suffering from the same flaw—to the analyst for auditing.

They evaluated the efficacy of our method using the source code of four popular open-source projects: LibTIFF, FFmpeg, Pidgin and Asterisk. We first demonstrate in an quantitative evaluation how functions are decomposed into structural patterns and how similar code can be identified automatically. In a controlled experiment we are able to narrow the search for a given vulnerability to 8.7% of the code base and consistently outperform non-structured approaches for vulnerability extrapolation. They also studied the discovery of real vulnerabilities in a qualitative

evaluation, where they were able to discover 10 zero-day vulnerabilities in the source code of the four open-source projects.

The concept of vulnerability extrapolation builds on the observation that source code often contains several vulnerabilities linked to the same flawed programming patterns, such as missing checks before or after function calls. Given a known vulnerability, it is thus often possible to discover previously unknown vulnerabilities by finding functions sharing similar code structure. In practice, such extrapolation of vulnerabilities is attractive for two reasons: First, it is a general approach that is not limited to any specific vulnerability type. Second, the extrapolation does not hinge on any involved analysis machinery: a robust parser and an initial vulnerability are sufficient for starting an analysis. However, assessing the similarity of code is a challenging task, as it requires analyzing and comparing structured objects, such as subtrees of syntax trees. Previous work has thus only considered flat representations, such as function and type names, for extrapolating vulnerabilities. To tackle the challenge of structured data, our method combines concepts from static analysis, robust parsing and machine learning.

They concluded that a key to strengthening the security of computer systems is the rigorous elimination of vulnerabilities in the underlying source code. To this end, we have introduced a method for accelerating the process of manual code auditing by suggesting potentially vulnerable functions to an analyst. Their method extrapolates known vulnerabilities using structural patterns of the code and enables efficiently finding similar flaws in large code bases. Empirically, they have demonstrated this capability by identifying real zero-day vulnerabilities in open-source projects, including Pidgin and FFmpeg. The concept of vulnerability extrapolation is orthogonal to other approaches for finding vulnerabilities and can be directly applied to complement current instruments for code auditing.

For example, if a novel vulnerability is identified using fuzz testing or symbolic execution, it can be extrapolated to the entire code base, such that similar flaws can be immediately patched. This extrapolation raises the bar for attackers, as they are required to continue searching for novel vulnerabilities, once an existing flaw has been sufficiently extrapolated and related holes have been closed in the source code.

2.5 A SYSTEMATIC LITERATURE REVIEW ON SOFTWARE VULNERABILITY DETECTION USING MACHINE LEARNING APPROACHES

Software vulnerabilities are security flaws, defects, or weaknesses in software architecture, design, or implementation. With the explosion of open source code available for analysis, there is a chance to learn about bug patterns that can lead to security vulnerabilities to assist in the discovery of vulnerabilities. Recent advances in deep learning in natural language processing, speech recognition, and image processing have demonstrated the great potential of neural models to understand natural language. This has encouraged researchers in the cyber security sector and software engineering to utilize deep learning to learn and understand vulnerable code patterns and semantics that indicate vulnerable code properties. In this paper, we review and analyze the recent state-of-the-art research adopting machine learning and deep learning techniques to detect software vulnerabilities, aiming to investigate how to leverage neural techniques for learning and understanding code semantics to facilitate vulnerability. All of the above models are proposed to obtain better fitting and predicting models for different vulnerability datasets but most of them are against of certain vulnerability datasets. These models try to get a better model under a certain condition, but it cannot give good results with every vulnerability dataset. From this paper's results, 12 primary studies were found from the search processes. 7 out of them were published in IEEE, 2 were published in ACM, 2 were published in Springer and the rest of them were published in different conferences and journals. Most primary studies worked on NVD and SARD datasets, and others used open-source projects. Results show that machine learning and deep learning techniques give promising results in the automatic detection of vulnerabilities, but there are still some gaps in existing models that need to be addressed in future research. The procedure of the study search consisted of selecting digital sources, constructing a search string, IEEE Xplore Springer Link ACM Other journals & conferences Search Strings Construction After selecting the sources, the search string needs to be constructed to perform a comprehensive search to select the most relevant search studies to the topic as follows: (Vulnerability OR Software Vulnerability) AND (Detection OR

Automatic Detection OR Discovery) AND (Machine Learning OR Deep Learning OR Deep Neural Networks) (Automatic AND Vulnerability AND Detection OR Discovery) OR (Automatically AND Detect AND Vulnerabilities) OR (Automated AND Vulnerability AND Detection) AND (Machine Learning OR Deep Learning) Inclusion and Exclusion Criteria Search strings are used to retrieve all available research papers in the digital sources mentioned above. In order to select the primary studies from the initial search result, inclusion and exclusion criteria were designed. Inclusion criteria: the text is written in the English language. relevant to the detection of software vulnerabilities. a paper that has been published in a journal or at a conference. publications that have been peer- reviewed. Exclusion criteria: research studies that are irrelevant to the search string. without any empirical research or results. outdated search papers..Although there is no universal definition for software vulnerability, previous studies have given varied explanation of the concept. Kuang et al. defined software vulnerability as the “fault that can be viciously used to harm security of software systems”.Software developers have developed a lot of methods and tools by using several approaches to detect and report these vulnerabilities that pose security threat to systems and users. The CERT/CC (Computer Emergency Response Team Coordination Center) reported that the economic loss caused by the intrusion events has reached about 6.66 billion US dollars in 2003 and this figure is still on the ascendancy with the passage of time. For example, there were a total of 7236 vulnerabilities in 2007, and this number has increased to 4110 by the end of the first two quarter of 2008 .Security of the software system is a prime focus area for software development teams. This paper explores some data science methods to build a knowledge management system that can assist the software development team to ensure a secure software system is being developed. Various approaches in this context are explored using data of insurance domain-based software development.

CHAPTER 3

SYSTEM ANALYSIS

3.1 EXISTING SYSTEM

The automatic classification model of vulnerability (IGTF-DNN) based on TF-IDF is constructed in existing system. The relevant definitions are as follows.

A.TF-IDF: (Term Frequency/Inverse Document Frequency) is a common weighted technology which is found out based on statistical methods. For example, consider there are a set of documents and each document contains a number of terms/words. TF-IDF is used to measure the terms' importance word to a document in the document set or in a corpus. The terms' importance increases proportionally with number of times it appears in the document, but also decreases inversely with frequency it appears in corpus.

B.INFORMATION GAIN (IG): refers to that, if a feature X in class Y is known already, information uncertainty of class Y decrease, and so reduced uncertainty degree will reflect importance of feature X to class Y. Based on feature selection method of information gain criterion, each feature's information gain is measured, and the features with larger IG value are selected. In addition, neural network is used to automatically classify the vulnerability document content in the vulnerability type.

3.1.1 DRAWBACKS

- Fixed data set files are given as input. Only vulnerability details from year 2014 to 2018 are taken for processing.
- Back propagation technique is not applied so accuracy is limited. i.e, initial weight values for hidden layer is not re-calculated and does not improve based on their outputs.
- Limited vulnerability category is selected for classification.

3.2 PROPOSED SYSTEM

In proposed system, all the existing methodology is carried out. Deep neural network is used to automatically classify the vulnerability document content in the vulnerability type with back propagation technique so that hidden layer weights are readjusted with effective values.

3.2.1 ADVANTAGES

- More data set files can be given as input with new vulnerability types. Vulnerability details from year 2014 to 2019 are taken for processing with 500 records in each year.
- Back propagation technique is applied so accuracy is improved. i.e, initial weight values for hidden layer is recalculated and improve based on their outputs.
- 27 vulnerability categories are selected for classification. Can be increased with increase in output layer neuron count.

3.3 FEASIBILITY STUDY

The feasibility study deals with all the analysis that takes up in developing the project. Each structure has to be thought of in the developing of the project, as it has to serve the end user in a user-friendly manner. One must know the type of information to be gathered and the analysis consist of collecting, Organizing and evaluating facts about a system and its environment .The main objective of the system analysis is to study the existing operation and to learn and accomplish the processing activities. The **TF-IDF** (Term Frequency/Inverse DocumentFrequency) and Information Gain finding through R needs to be analyzed well. The details are processed through coding themselves. It will be controlled by the programs alone.

3.3.1 ECONOMIC FEASIBILITY

The organization has to buy a personal computer with a keyboard and a mouse, this is a direct cost. There are many direct benefits of covering the manual system to computerized system. The user can be given responses on asking questions, justification of any capital outlay is that it will reduce expenditure or improve the quality of service or goods, which in turn may be expected to provide the increased profits.

3.3.2 OPERATIONAL FEASIBILITY

The Proposed system accessing process to solves problems what occurred in existing system. The current day-to-day operations of the organization can be fit into this system. Mainly operational feasibility should include on analysis of how the proposed system will affects the organizational structures and procedures.

3.3.3. TECHNICAL FEASIBILITY

The cost and benefit analysis may be concluded that computerized system is favorable in today's fast moving world. The assessment of technical feasibility must be based on an outline design of the system requirements in terms of input, output, files, programs and procedure.

The project aims to provide classification type of the given vulnerability description into pre-defined categories. The feature word present in the given description 1 is set otherwise 0 is set. So the vector encoded is given to Deep Neural Networks. The outputs of the NN are the matching type of the vulnerability. The current system aims to overcome the problems of the existing system. The current system is to reduce the technical skill requirements so that more number of data can be mined to find the fraudulent users.

CHAPTER - 4

SYSTEM SPECIFICATION

4.1 HARDWARE REQUIREMENTS

This section gives the details and specification of the hardware on which the system is expected to work.

Processor	:	Dual Core 2.1 GHz
RAM	:	2 GB SD RAM
Monitor	:	17" Color
Hard disk	:	500 GB
Keyboard	:	Standard 102 keys
Mouse	:	Optical mouse

4.2 SOFTWARE REQUIREMENTS

This section gives the details of the software that are used for the development.

Operating System	:	Windows 10 Pro
Environment	:	R Studio 1.0
Language	:	R 3.4.4

CHAPTER – 5

SOFTWARE DESCRIPTION

5.1 FRONT END R LANGUAGE

R is an open source programming language and software environment for statistical computing and graphics that is supported by the R Foundation for Statistical Computing. The R language is widely used among statisticians and data miners for developing statistical software and data analysis. Polls, surveys of data miners, and studies of scholarly literature databases show that R's popularity has increased substantially in recent years. R is a GNU package. The source code for the R software environment is written primarily in C, Fortran, and

R. R is freely available under the GNU General Public License, and pre-compiled binary versions are provided for various operating systems. While R has a command line interface, there are several graphical front-ends available.

5.1.1 FEATURES OF R LANGUAGE

R is an integrated suite of software facilities for data manipulation, calculation and graphical display. It includes

- Aneffective data handling and storage facility,
- A suite of operators for calculations on arrays, in particular matrices,
- A large, coherent, integrated collection of intermediate tools for data analysis,
- graphical facilities for data analysis and display either on-screen or on hardcopy, and
- A well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities.
- The term “environment” is intended to characterize it as a fully planned and coherent system, rather than an incremental accretion of very specific and inflexible tools, as is frequently the case with other data analysis software.

5.1.2 STATISTICAL FUNCTIONS

R, like S, is designed around a true computer language, and it allows users to add additional functionality by defining new functions. Much of the system is itself written in the R dialect of S, which makes it easy for users to follow the algorithmic choices made. For computationally- intensive tasks, C, C++ and Fortran code can be linked and called at run time. Advanced users can write C code to manipulate R objects directly.

Many users think of R as a statistics system. User prefers to think of it of an environment within which statistical techniques are implemented. R can be extended (easily) via packages. There are about eight packages supplied with the R distribution and many more are available through the CRAN family of Internet sites covering a very wide range of modern statistics. R has its own LaTeX-like documentation format, which is used to supply comprehensive documentation, both on-line in a number of formats and in hardcopy.

5.1.3 INTERFACES

The most commonly used graphical integrated development environment for R is RStudio. A similar development interface is R Tools for Visual Studio. Interfaces with more of a point- and-click approach include Rattle GUI, R Commander, and RKWard. Some of the more common editors with varying levels of support for R include Eclipse, Emacs (Emacs Speaks Statistics), Kate, LyX, Notepad++, WinEdt, and Tinn-R. R functionality is accessible from several scripting languages such as Python, Perl, Ruby, F#, and Julia.

The R language came to use quite a bit after S had been developed. One key limitation of the S language was that it was only available in a commercial package, S-PLUS. Ross's and Robert's experience developing R is documented in a 1996 paper in the Journal of Computational and Graphical Statistics. an important contribution by convincing Ross and Robert to use the GNU General Public License to make R free software. This was critical because it allowed for the source code for the entire R system to be accessible to anyone who wanted to tinker with it.

5.1.4 BASIC FEATURES OF R

Basic key feature of R was that its syntax is very similar to S, making it easy for S-PLUS users to switch over. While the R's syntax is nearly identical to that of S's, R's semantics, while superficially similar to S, are quite different. In fact, R is technically much closer to the Scheme language than it is to the original S language when it comes to how R works under the hood. R runs on almost any standard computing platform and operating system. Its open source nature means that anyone is free to adapt the software to whatever platform they choose. Indeed, R has been reported to be running on modern tablets, phones, PDAs, and game consoles. R has over many other statistical packages (even today) its sophisticated graphics capabilities. R's ability to create "publication quality" graphics has existed since the very beginning and has generally been better than competing packages. Today, with many more visualization packages available than before, that trend continues. R's base graphics system allows for very fine control over essentially every aspect of a plot or graph. Other newer graphics systems, like lattice and ggplot2 allow for complex and sophisticated visualizations of high-dimensional data.

R has maintained the original S philosophy, which is that it provides a language that is both useful for interactive work, but contains a powerful programming language for developing new tools. This allows the user, who takes existing tools and applies them to data, to slowly but surely become a developer who is creating new tools.

R has nothing to do with the language itself, but rather with the active and vibrant user community. In many ways, a language is successful inasmuch as it creates a platform with which many people can create new things. R is that platform and thousands of people around the world have come together to make contributions to R, to develop packages, and help each other use R for all kinds of applications. The R-help and R-devel mailing lists have been highly active for over a decade now and there is considerable activity on web sites like Stack Overflow.

5.1.5 FREE SOFTWARE

A major advantage that R has over many other statistical packages and is that it's free in the sense of free software (it's also free in the sense of free beer). The copyright for the primary source code for R is held by the R Foundation and is published under the GNU General Public License version 2.0. According to the Free Software Foundation, with free software, you are granted the following four freedoms.

- The freedom to run the program, for any purpose.
- The freedom to study how the program works, and adapt it to need. Access to the sourcecode is a precondition for this.
- The freedom to redistribute copies so help the neighbor package.
- The freedom to improve the program, and release improvements to the public, so that the whole community benefits (freedom 3). Access to the source code is a precondition for this.

5.1.6 R STUDIO

R is a freely available environment for statistical computing. R works with a command-line interface, meaning you type in commands telling R what to do. RStudio is a convenient interface for using R, which can either be accessed online or downloaded to local computer.

- RScript to the console either by highlighting and clicking this icon: or else by typing CTRL+ENTER at the end of the line. Different RScripts can be saved in different tabs.
- The top right is user Workspace and is where user will see objects (such as datasets and variables). Clicking on the name of a dataset in user workspace will bring up a spreadsheet of the data.
- The bottom right serves many purposes. It is where plots will be appear, where user manages files, where install packages, and where the help information appears. Use the tabs to toggle back and forth between these screens as needed.

CHAPTER 6

PROJECT DESCRIPTION

6.1 PROBLEM DEFINITION

Although the machine learning classification algorithms have achieved promising results in many fields, due to the large amount of vulnerability data and short description, the generated word vector space presents the characteristics of high dimension and sparse. These machine learning algorithms are not very effective in dealing with high and sparse problems. At the same time, they ignore specific vulnerability information and the classification accuracy is not high. However, in recent years, deep learning has been applied in many fields and has achieved success, such as software testing. Therefore, in order to better deal with the high and sparse word vector space and take advantage of the automatic feature extraction by deep learning, this project proposes an automatic vulnerability classification model TFI-DNN based on term frequency- reverse document frequency(TF-IDF), information gain(IG) and deep neural network(DNN)

6.1.1 PROBLEM STATEMENT

THE IGTF-DNN ALGORITHM

The vulnerability automatic classification model IGTF-DNN is composed of IG, TF and DNN. The original vulnerability data is first preprocessed, and then TFI is used to grab features of vulnerability description text and lower the dimensionality of generated higher- dimensional word vector space, and later the DNN is build to comprehend automatic training and classification of vulnerability.

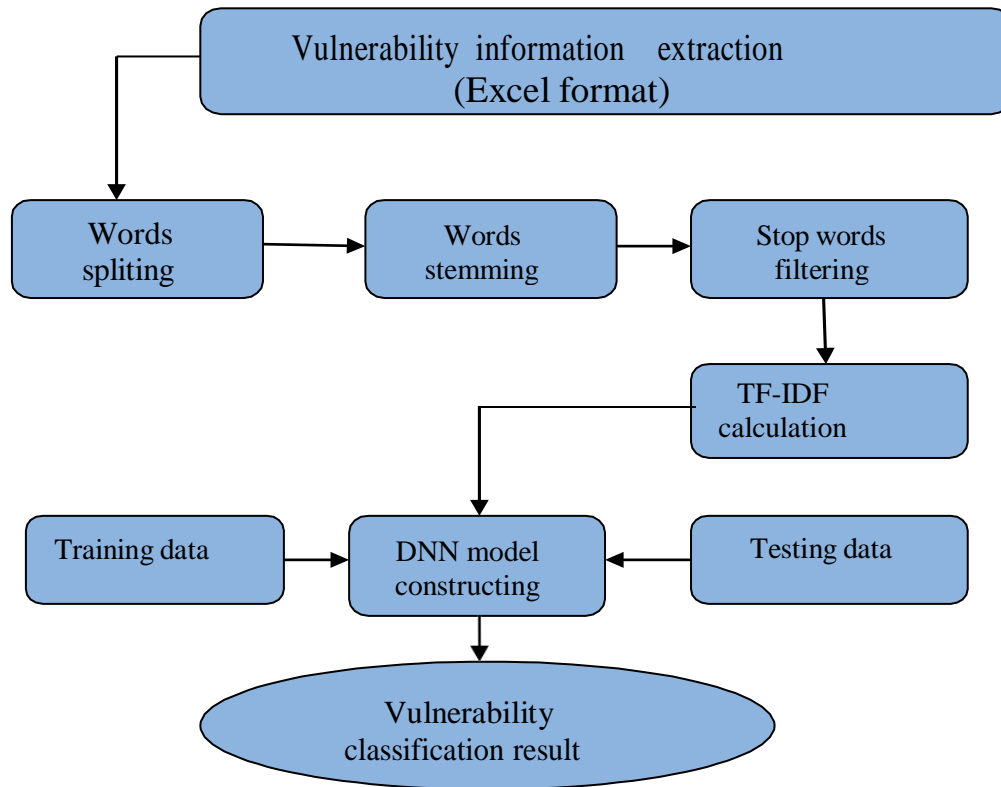


Figure 6.1 TFI-DNN algorithm

A. FEATURE SELECTION USING TFI: TFI is used to grab feature word set. The steps are given in Algorithm 1 which is found in modules description chapter.

B. OPTIMIZATIONS USING DNN: DNN includes single input layer, one hidden layer (can be made to two if required) and single output layer, whose input is the instance's feature vector and output is instances' category. It includes one propagation process, forward propagation and back propagation. The propagation process is in Algorithm 2 which is found in modules description chapter.

6.2 OVERVIEW OF PROJECT

Software vulnerabilities are raising the security risks. If any vulnerability is oppressed due to a malicious attack, it will compromise the system's safety. In addition, it may create catastrophic losses. So, automatic classification methods are required to manage vulnerability in software, and then security performance of the system will be improved. It will also mitigate the risk of system being attacked and damaged. In this project, a new model has been proposed with name automatic vulnerability classification model (IGTF-DNN) Information Gain based on TermFrequency- Deep NeuralNetwork.

The model is generated using information gain (IG) which is based on frequency-inverse document frequency (TF-IDF), and deep neural network (DNN): TF-IDF is used to calculate frequency/weight of words taken from vulnerability description; IG is used to select features to gather optimal set of feature words. Then neural network model is used to construct an automatic vulnerability classifier to achieve effective vulnerability classification.

The National Vulnerability Database of the United States has been taken to test this new model's effectiveness. By comparing with KNN, this TFI-DNN model has achieved better performance in evaluation.

6.3 MODULE DESCRIPTION

The following modules are present in the project

- FEATURE WORDS EXTRACTION
- TERM FREQUENCY – INVERSE DOCUMENT
- FREQUENCY FEATURE WORD EXTRACTION
- OPTIMIZATIONS USING DNN

6.3.1 FEATURE WORDS EXTRACTION

In this module, the following algorithm is worked out.

- Traversing each word in the word_list.
- Word frequency statistics for word_list, stored in the doc_frequency list.
- Traversing the word frequency list doc_frequency.
- Calculate the TF value of each word according to (1) and store it in the word_tf dictionary.
- Calculate the IDF value of each word according to (2) and store it in the word_idf dictionary.

- The word set is sorted in descending according to the TF-IDF value.
- Select the first n words as an important feature set.
- Save important words in the feature list (features_vocabSet).
- Traverse features_vocabSet, divide features_vocabSet and store the subset into the sub Data Set.
- Calculate probability of sub Data Set.
- Calculate the empirical conditional entropy of each word according to (4) and (5) and store it in new Entropy.
- Calculate the IG value of each word according to (6).
- Save each word and the corresponding IG value in the dictionary.
- The word set is sorted descending by IG value.
- Select the first words as features and store them in the feature_words.
- Return feature_words.

6.3.2 TF-IDF(Term Frequency/Inverse Document Frequency)

It is a common weighted technology which is found out based on statistical methods [17]. For example, consider there are a set of documents and each document contains a number of terms/words. It is defined that the word I's importance in document j as follows.

$$Tf_{ij} = n_{ij} / \sum_k n_{k,j} \quad (1)$$

Where both I and j are positive integers, $n_{i,j}$ denotes the term I's frequency in document j. The

IDF formula is as follows.

$$idf_i = \log (|F| / |\{ j: t_i \in d_j \}|)$$

(2) where $|F|$ is the total number of documents in corpus, f_j is the j_{th} document, and $|\{j: t_i \in f_j\}|$ is the number of documents containing the term t_i .

The TF-IDF formula is as follows

$$TF-IDF = tf_{ij} * idf_i \quad (3)$$

TF-IDF is used to measure the terms' importance word to a document in the document set of a corpus. The terms' importance increases proportionally with number of times it appears in the document, but also decreases inversely with frequency it appears in corpus.

TABLE 1- TF-IDF FEATURES:

S.No	Word	frequency
1	Xss	0.005312172
2	Service	0.005032584
3	Via	0.004962687
4	Denial	0.004892790
5	Web	0.004683099
6	Attackers	0.004543305
7	Remote	0.004473408
8	Cause	0.004473408
9	Scripting	0.004403511
10	Crosssite	0.004333614

6.3.3 INFORMATION GAIN (IG)

It refers to that, if a feature X in class Y is known already, information uncertainty of class Y decrease, and so reduced uncertainty degree will reflect importance of feature X to class Y. Set the training data set to D, |D| shows the count of samples in D. Suppose there are K classes C_k , $k = 1, 2, \dots, K$ | C_k | is the count of samples fit in to class C_k . $\sum_{k=1}^K |C_k| = |D|$. If feature A has n different values $\{a_1, a_2, \dots, a_n\}$, D is segmented into 'n' sub groups according to feature A values, represented as $D = (D_1, D_2, \dots, D_n)$, where | D_i | is the samples count in D_i , $\sum_{i=1}^n |D_i| = |D|$. The samples set fit into class C_k in D_i is D_{ik} , $D_{ik} = D_i \cap C_k$, | D_{ik} | is the samples count of D_{ik} .

The empirical entropy $H(D)$ of data set D is calculated as follows.

$$H(D) = - \sum_{k=1}^K \frac{|C_k|}{|D|} \log_2 \frac{|C_k|}{|D|}$$

The empirical conditional entropy $H(D|A)$ of feature A for dataset D is calculated

$$H(D|A) = - \sum_{i=1}^n \frac{|D_i|}{|D|} \sum_{k=1}^K \frac{|D_{ik}|}{|D_i|} \log_2 \frac{|D_{ik}|}{|D_i|}$$

The information gain calculation formula for each feature is as follows.

$$g(D, A) = H(D) - H(D|A) \quad (5)$$

Based on feature selection method of information gain criterion, each feature's information gain is measured, and the features with larger IG value are selected.

TABLE 2- INFORMATION GAIN (IG):

S.No	Feature	Gain Value
1	xss	0.018107652
2	service	0.106077030
3	via	0.735451987
4	denial	0.081597715
5	web	0.090538261
6	attackers	0.610320229
7	remote	0.606898196
8	cause	0.130556345
9	scripting	0.018107652
10	Cross site	0.022634565
11	allows	0.767428160
12	inject	0.022634565
13	arbitrary	0.389475942
14	script	0.022634565
15	html	0.013580739

6.3.4. OPTIMIZATIONS USING DNN

In this module, DNN is used which consists of one input layer, multiple hidden layers and one output layer, whose input is the feature vector of instance and output is the category of instance. It mainly includes two propagation processes, forward propagation and back propagation. The propagation process is in Algorithm2.

6.4 INPUT DESIGN

Input design is the process of converting user-originated inputs to a computer understandable format. Input design is one of the most expensive phases of the operation of computerized system and is often the major problem of a system. A large number of problems with a system can usually be tracked back to fault input design and method. Every moment of input design should be analyzed and designed with utmost care.

The system takes input from the users, processes it and produces an output. Input design is link that ties the information system into the world of its users. The system should be user-friendly to gain appropriate information to the user. The decisions made during the input design are

- To provide cost effective method of input.
- To achieve the highest possible level of accuracy.
- To ensure that the input is understood by the user.

System analysis decide the following input design details like, what data to input, what medium to use, how the data should be arranged or coded, data items and transactions needing validation to detect errors and at last the dialogue to guide user in providing input. Input data of a system may not be necessarily is raw data captured in the system from scratch. These can also be the output of another system or subsystem.

The design of input covers all the phases of input from the creation of initial data to actual entering of the data to the system for processing. The design of inputs involves identifying the data needed, specifying the characteristics of each data item, capturing and preparing data from computer processing and ensuring correctness of data. Any Ambiguity in input leads to a total fault in output. The goal of designing the input data is to make data entry as easy and error free as possible.

6.5 OUTPUT DESIGN

Output design generally refers to the results and information that are generated by the system for many end-users; output is the main reason for developing the system and the basis on which they evaluate the usefulness of the application.

The output is designed in such a way that it is attractive, convenient and informative. As the outputs are the most important sources of information to the users, better design should improve the system's relationships with user and also will help in decision-making. Form design elaborates the way output is presented and the layout available for capturing information.

6.6 SYSTEM FLOW DIAGRAM:

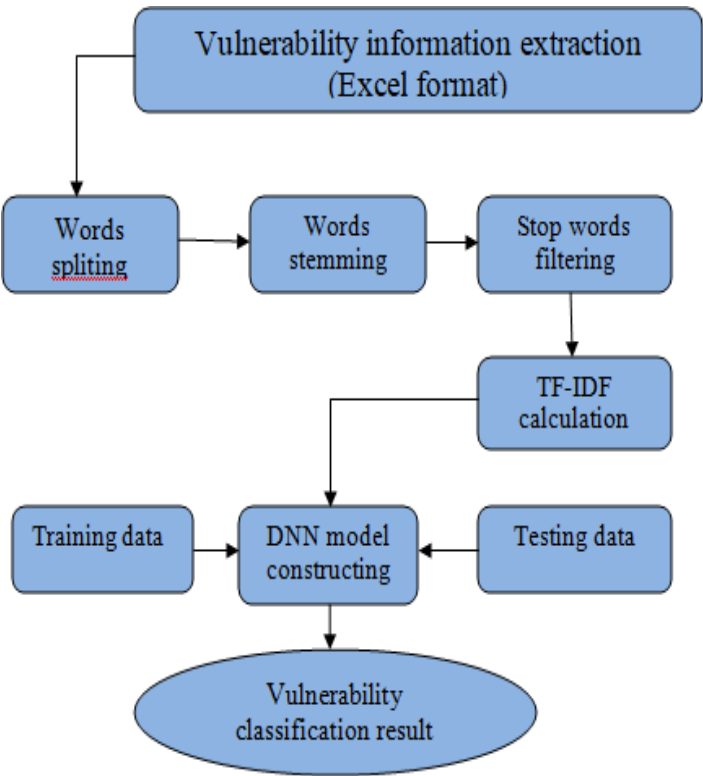


FIG 6.6- SYSTEM FLOW DIAGRAM

6.7 USE CASE DIAGRAM

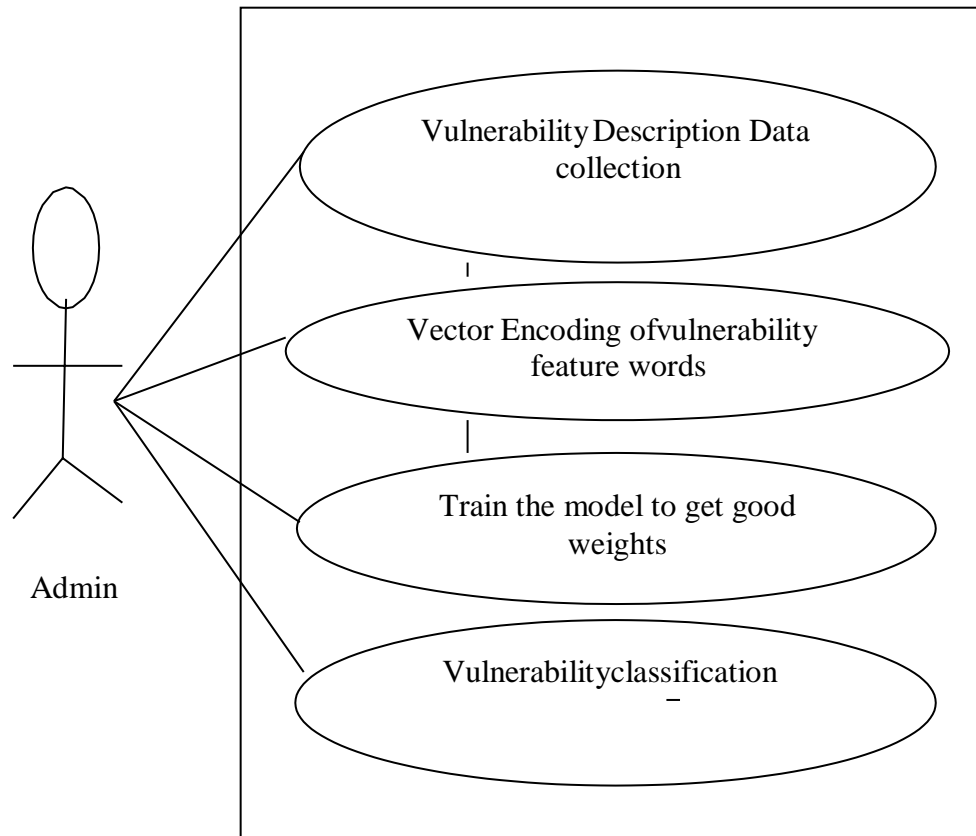


Fig 6.7 Use Case Diagram

6.8 SEQUENCE DIAGRAM

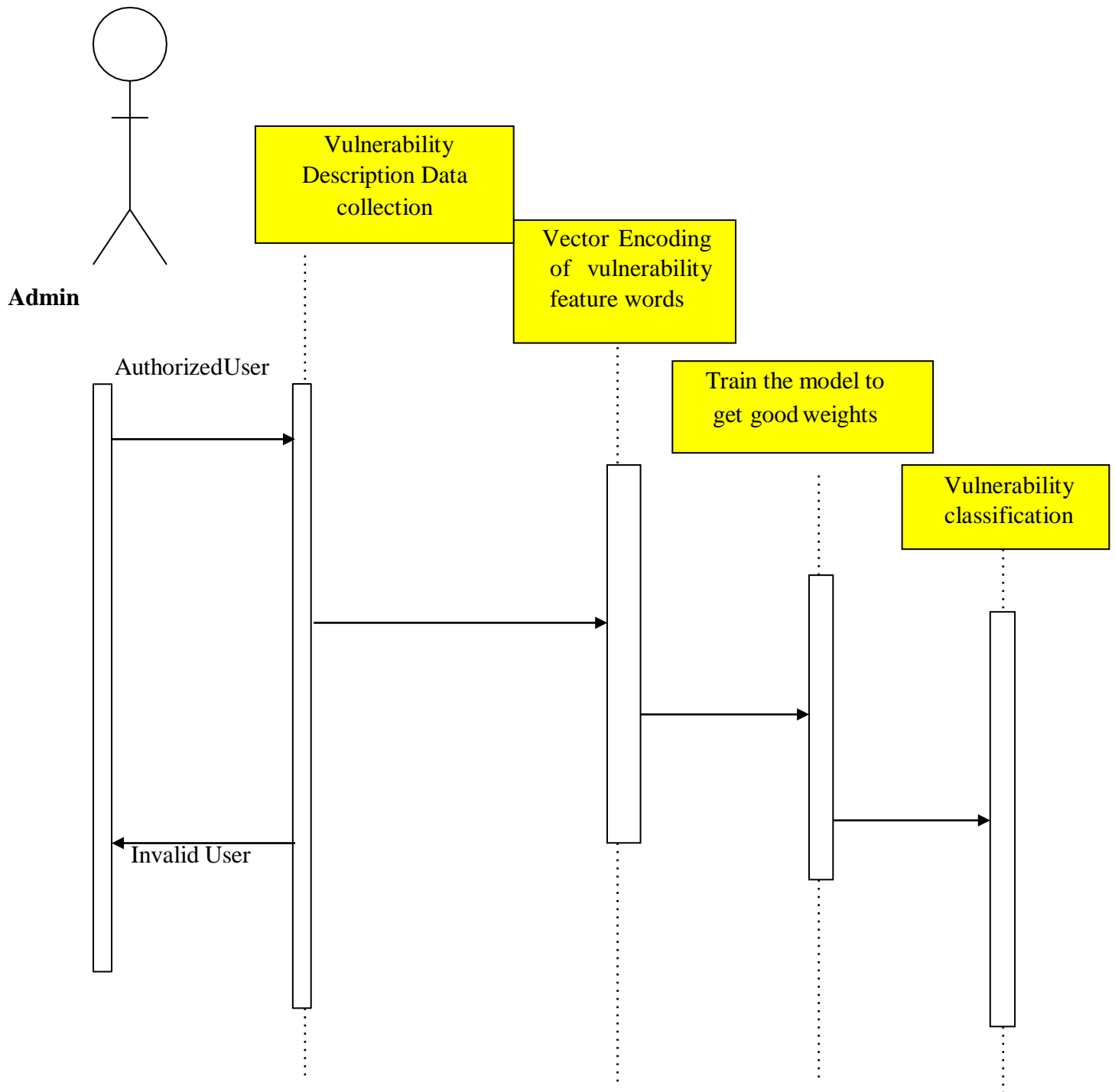


Fig 6.8 Sequence Diagram

CHAPTER 7

CONCLUSION

In order to better analyze and manage vulnerabilities according to their belonging classes, improve the security performance of the system, and reduce the risk of the system being attacked and damaged, this paper applied deep neural network to software vulnerability classification. The analysis of the method and construction process of TFI and DNN are discussed in detail. The comparison is made with the vulnerability classification model TFI-DNN to TFI-SVM, TFI- Naïve Bayes and TFI-KNN on the NVD dataset. The results show that the proposed TFI-DNN model outperforms well in preparing weights and biases. And it is superior to general TF-IDF on comprehensive evaluation indexes. The work in this paper shows the effectiveness of TFI-DNN in vulnerability classification, and provides a basis for our future research using the benchmark vulnerability dataset.

APPENDIX

A.SAMPLE SOURCE CODE

```
#https://www.cvedetails.com
#https://www.cvedetails.com/vulnerability-list/year-1999/month-1/January.html
library(tm)
#r1=read.xlsx("health10_3.xlsx",sheetName="Sheet1")
r1=read.csv("./CorrectedCVEDataset/DataSet2015.csv")
r2=read.csv("./CorrectedCVEDataset/DataSet2016.csv")
r3=read.csv("./CorrectedCVEDataset/DataSet2017.csv")
r4=read.csv("./CorrectedCVEDataset/DataSet2018.csv")
r5=read.csv("./CorrectedCVEDataset/DataSet2019.csv")#r1
df1<-data.frame(r1) df2<-data.frame(r2) df3<-data.frame(r3) df4<-data.frame(r4) df5<-
data.frame(r5)df<-rbind(df1,df2) df<-rbind(df,df3) df<-rbind(df,df4) df<-rbind(df,df5)

catarray<-unlist(unique(df['Vulnerability.Type.s.']))catarray<-as.vector(catarray)
catlength<-length(catarray)

#mydatframes<-list()
mydatframes<-vector("list", catlength)myTDMat<-vector("list", catlength)
```

```

categorywiselists<-vector("list", catlength)totalrecords<-nrow(df)
catcount<-c()
DataSet<-data.frame(Feature=character(0),Category=character(0))AllWords<-c()
leng<-5 #Five documents are taken from folder 'for(i3 in 1:catlength)
{
mydatframes[[i3]]<-subset(df,Vulnerability.Type.s==catarray[i3])
mydatframes[[i3]]$Vulnerability.Type.s.
rows<-nrow(mydatframes[[i3]])catcount<-c(c(catcount),rows)
tmp1<- paste(mydatframes[[i3]]$Description , collapse=' ')

tmp1 <- Corpus(VectorSource(tmp1))
tmp1 <- tm_map(tmp1, removePunctuation)
tmp1 <- tm_map(tmp1, function(x)removeWords(x,stopwords())) tmp1 <- tm_map(tmp1,
removeWords, c("RT", "are", "that")) myTDMat[[i3]]<-TermDocumentMatrix( tmp1,
control=list(removePunctuation=TRUE,stopwords=TRUE,removeNumbers=TRUE))#,weight
ing
=weightBin()))
myMatrices[[i3]]<-as.matrix(myTDMat[[i3]])
myMat[[i3]] <- sort(rowSums(myMatrices[[i3]]),decreasing=TRUE)#class(myMat[[i3]])
myMat[[i3]] <-data.frame(word=names(myMat[[i3]]),freq=myMat[[i3]])
myMat[[i3]][['Category']]<-"
myMat[[i3]][['Category']]<-catarray[i3]#singlecategorycontents<- paste
AllWords<-c(c(AllWords), myTDMat[[i3]]$dimnames$Terms)if(i3==1)
{

```

```

Mat <- myMatrices[[i3]]
} else
{
Mat<-rbind(Mat, myMatrices[[i3]])
}
}

AllWordsCount<-length(AllWords)AllDocumentsCount<-1

TFMat<-matrix(nrow=AllWordsCount,ncol=      AllDocumentsCount)      IDFVector<-
replicate(AllWordsCount,0)
TFIDF<-replicate(AllWordsCount,0)for(i in 1:AllWordsCount)
{
for(j in 1:AllDocumentsCount)
{

nume<- Mat[i,j]
deno <-colSums(Mat)
val1<-round(nume/deno ,4)[1]TFMat[i,j]<-val1
}
}

TFIDFMatrix<-matrix(nrow =AllWordsCount,ncol=2)

for(i in 1:AllWordsCount)
{

vect<-Mat[i,]
val2<-length(vect[vect>0])

```



```
IDFVector[i]<- log10(round( leng /val2,4)[1])
```

```
TFIDF[i]<- rowSums(TFMat)[i] * IDFVector[i] TFIDFMatrix[i,1]<-AllWords[i]
TFIDFMatrix[i,2]<-TFIDF[i]
}
```

```
#Line 7k<-0
```

```
AllWordsSorted<-AllWords AllWordsCountMinusOne<-AllWordsCount-1 for(i in
1:AllWordsCountMinusOne)
```

```
{
```

```
k<-i+1
```

```
for(j in k:AllWordsCount)
```

```
{
```

```
if(TFIDF[i] <TFIDF[j])
```

```
{
```

```
tmp1<- TFIDF[i] TFIDF[i]<-TFIDF[j]TFIDF[j]<-tmp1
```

```
tmp1<- AllWordsSorted[i] AllWordsSorted[i]<-AllWordsSorted[j]AllWordsSorted[j]<-tmp1
```

```
}
```

```
}
```

```
}
```

```
tfidf_df_sorted <-data.frame(word=AllWordsSorted,freq=TFIDF)#Line 8 and 9
```

```
if( nrow(tfidf_df_sorted ) >200)
```

```

tfidf_df_sorted<-tfidf_df_sorted[1:200,]tfidf_df_sorted
v1<-as.vector(tfidf_df_sorted$word)v1[1]
v1leng<-length(v1)str1<-"
for(i in 1:v1leng)
{
str1 <-paste(str1,trimws(as.character( v1[i])),'\n',collapse=",sep=")
}
#write_lines(x=str1,path="TFIDFfileoutput.txt")v2<-as.vector(tfidf_df_sorted$freq)
v2leng<-length(v2)str2<-"
for(i in 1:v2leng)
{
str2 <-paste(str2,trimws(as.character( v2[i])),'\n',collapse=",sep=")
}
library(readr) #write_lines(x=str2,path="TFIDFValuefileoutput.txt")

#write.csv(as.vector(tfidf_df_sorted$freq),file="TFIDFValuefileoutput.txt")      #TEXT
PROCESSINGENDS HERE=====

```

```

#=====
==
=====

```

#empiricalentropyH (D) ofdata set D is calculated as follows. (4)

```

AllWords<- as.vector(tfidf_df_sorted$word)AllWordsCount<-length(AllWords)

FeaturesinCategories<- vector("list",catlength) #3 for hw,sw and swt DataSet<-
data.frame(Feature=character(0),Category=character(0))for(i in 1:AllWordsCount)
{
for(j in 1:catlength)
{
if(AllWords[i] %in% myMat[[j]]$word)
{
FeaturesinCategories[[j]]<- c(c(FeaturesinCategories[[j]]),AllWords[i])
DataSet <- rbind(DataSet, data.frame(Feature=AllWords[i],Category=catarray[j]))

}
}
# if (AllWords[i] %in% catarray[1])# {
# FeaturesinCategories[[1]]<- c(c(FeaturesinCategories[[1]]),AllWords[i])
# DataSet <- rbind(DataSet, data.frame(Feature=AllWords[i],Category='Hardware'))# }
# if (AllWords[i] %in% swlist)# {
# FeaturesinCategories[[2]]<- c(c(FeaturesinCategories[[2]]),AllWords[i])
# DataSet <- rbind(DataSet, data.frame(Feature=AllWords[i],Category='Software'))# }
# if (AllWords[i] %in% swtlist)# {
# FeaturesinCategories[[3]]<- c(c(FeaturesinCategories[[3]]),AllWords[i])
# DataSet <- rbind(DataSet, data.frame(Feature=AllWords[i],Category='Testing'))# }

```

```

} #=====
DSSize<-nrow(DataSet)      #AllWordsCount      AllWords<-as.vector(DataSet$Feature)
AllWordsCount<-DSSize FeaturesCount<-c()
for(i in 1:catlength)
{
FeaturesCount<-c(c(FeaturesCount), length( FeaturesinCategories[[i]] ) )
}
#Features1Count<-length( FeaturesinCategories[1])op1<-0
tmpsum<-0
for(i in 1:catlength)
{
if(FeaturesCount[i]/DSSize>0)
{
tmpsum<-tmpsum + ((FeaturesCount[i]/DSSize)*log2(FeaturesCount[i]/DSSize) )
}
}
op1 <- -tmpsum
#op1 <- - ( ((Features1Count/DSSize)*log2(Features1Count/DSSize) )
+((Features2Count/DSSize)*log2( Features2Count/DSSize) )
+((Features3Count/DSSize)*log2( Features3Count/DSSize) ) )EntropyHofD <- (op1)

cat('EntropyH(D):\n')print(EntropyHofD)

#-((3/11) *log2(3/11) ) - (5/11 *log2(5/11)) - ((3/11) *log2(3/11) )

```

```

#- ( (14/30) * log2(14/30) ) -((8/30) * log2(8/30) ) -((8/30) * log2(8/30) )
#-(( (1/4) * log2(1/4) ) +((2/4) * log2(2/4) ) +((1/4) * log2(1/4) ))
#-(( (3/11) * log2(3/11) ) +((5/11) * log2(5/11) ) +((3/11) * log2(3/11) ))
overallsum<-0 #n<-DSSize KCate<-catlengthHDofA<-c()

InformationGainForAllWords<-c()

for(i in 1:DSSize)
{
#Calculation ofChild Entropy
#Refer HelpMust-InfoGain.pdf in this folder 7th pageWord<-AllWords[i]
deno1<- floor(DSSize /2) deno2<- DSSize - deno1 child1<-DataSet[ 1:deno1 ,]
child2<-DataSet[ deno1+1:deno2,]#.....
Child1Size<-nrow(child1) child1Subsets<-vector("list",catlength)child1SubsetsCount<-c()
for(j in 1:catlength)
{
child1Subsets[[j]] <-subset ( child1, Category==catarray[j] & Feature==Word)
child1SubsetsCount<-c(c(child1SubsetsCount), nrow(child1Subsets[[j]]))
}

#child1HwSubset <- subset ( child1, Category=='Hardware' & Feature==Word)

```

```

#child1SwSubset <- subset ( child1, Category=='Software' & Feature==Word)
#child1SwtSubset <- subset ( child1, Category=='Testing' & Feature==Word)
#child1Hwcount<-nrow(child1HwSubset)
#child1Swcount<-nrow(child1SwSubset)          #child1Swtcount<-nrow(child1SwtSubset)
#child1Hwcount
#child1Swcount#child1Swtcount

```

```

op1<-0 tmpsum<-0
for(k in 1:catlength)
{
op1<-0
if( is.infinite( log2( child1SubsetsCount[k]/Child1Size ) ))
{

}else
{
op1<-( child1SubsetsCount[k]/Child1Size ) * log2( child1SubsetsCount[k]/Child1Size )
}
tmpsum<- tmpsum+ op1
}
child1Entropy<- - tmpsum

```

```

Child2Size<-nrow(child1) child2Subsets<-vector("list",catlength)child2SubsetsCount<-c()
for(j in 1:catlength)
{

```

```

child2Subsets[[j]] <-subset ( child2, Category==catarray[j] & Feature==Word)
child2SubsetsCount<-c(c(child2SubsetsCount), nrow(child2Subsets[[j]]))
}

#child1HwSubset <- subset ( child1, Category=='Hardware' & Feature==Word)
#child1SwSubset <- subset ( child1, Category=='Software' & Feature==Word)
#child1SwtSubset <- subset ( child1, Category=='Testing' & Feature==Word)
#child1Hwcount<-nrow(child1HwSubset)
#child1Swcount<-nrow(child1SwSubset) #child1Swtcount<-nrow(child1SwtSubset)
#child1Hwcount
#child1Swcount#child1Swtcount

op1<-0 tmpsum<-0
for(k in 1:catlength)
{
op1<-0
if( is.infinite( log2( child2SubsetsCount[k]/Child2Size ) ))
{

}else
{
op1<-( child2SubsetsCount[k]/Child2Size ) * log2( child2SubsetsCount[k]/Child2Size )
}
tmpsum<- tmpsum+ op1
}
child2Entropy<- - tmpsum

```

```
WeightedAverageEntropyofChildren <- ( (Child1Size/ DSSize) * child1Entropy) + (
(Child2Size/ DSSize) * child2Entropy )
```

```
InformationGain <- EntrophyHofD - WeightedAverageEntropyofChildren
```

```
InformationGainForAllWords <- c(c(InformationGainForAllWords), InformationGain)
}
```

```
OutputDF<-data.frame(Feature=character(0),GainValue=numeric(0))UniqueWords<-c()
```

```
UniqueWordsValue<-c()for(i in 1:DSSize)
```

```
{
```

```
if(i==1)
```

```
{
```

```
UniqueWords<-c(c(UniqueWords),AllWords[i])
```

```
val1<- EntrophyHofD - InformationGainForAllWords [i] UniqueWordsValue<-
c(c(UniqueWordsValue), val1)
```

```
OutputDF<- rbind( OutputDF,data.frame(Feature= AllWords[i] , GainValue=val1 ) )
```

```
}
```

```
if( AllWords[i] %in% UniqueWords )
```

```
{
```

```
}else
```

```
{
```

```
UniqueWords<-c(c(UniqueWords),AllWords[i])
```

```
val1<- EntrophyHofD - InformationGainForAllWords [i] UniqueWordsValue<-
c(c(UniqueWordsValue), val1)
```

```
OutputDF<- rbind( OutputDF,data.frame(Feature= AllWords[i] , GainValue =val1 ) )
```

```
}
```

```
}
```


OutputDF

```
k<-0
```

```
UniqueWordsCount<-length(UniqueWords) UniqueWordsMinusOne<-UniqueWordsCount-1
```

```
for(i in 1:UniqueWordsMinusOne)
```

```
{
```

```
k<-i+1
```

```
for(j in k:UniqueWordsCount)
```

```
{
```

```
if(UniqueWordsValue[i] < UniqueWordsValue[j])
```

```
{
```

```
tmp1<- UniqueWordsValue[i] UniqueWordsValue[i]<-UniqueWordsValue[j]
```

```
UniqueWordsValue[j]<-tmp1
```

```
tmp1<- UniqueWords[i] UniqueWords[i]<-UniqueWords[j]UniqueWords[j]<-tmp1
```

```
}
```

```
}
```

```
}
```

```
OutputDF <-data.frame(word=UniqueWords,freq=UniqueWordsValue)
```

OutputDF

```
##
```

```
##
```

```
#totalcount<- subset(totcount, select =c(1, ncol(totcount)))# barplotvalues<- (totalcount)
# barplotvalues<-data.frame(t(barplotvalues))# barplotvalues<-as.matrix(barplotvalues)
XAxisTitle <-UniqueWords barplotvalues<-UniqueWordsValue
barplot(barplotvalues,names.arg=XAxisTitle,col=c('red','blue'),beside=TRUE
,main="FEATURE WORDS WITH INFORMATION GAIN VALUE",xlab="FEATURE
WORD",ylab="GAIN VALUE")
# legend("topleft", c('Program','BugReport'),fill=c('red','blue'))#
```

#C. THE WORD VECTOR SPACE ESTABLISHMENT PAGE 6 OF 8 IN BASE PAPER

#In the application of this paper, each vulnerability description is expressed as an m-dimensional vector

```

#(m is the number of feature words in the feature word set)m<-UniqueWordsCount
str1<-"

```

```

for(j in 1:m)
{
df[600,'Description']
}

```

```
#df$Vulnerability.Type.s.#UniqueWordsCount
```

```
df['VectorEncoded']<- "dfrows<-nrow(df)
```

```

for(i in 1:dfrows)
{
vs<- df[i,4] #Description]cnninputvector<-"
for(j in 1:UniqueWordsCount )
{

if( grepl(UniqueWords[j] ,vs))
{
cnninputvector <- paste(cnninputvector,'1',sep=")
}else
{
cnninputvector <- paste(cnninputvector,'0',sep=")
}
}
df[i,'VectorEncoded']<-cnninputvector
}
df$VectorEncoded length(df$VectorEncoded[1])#Project Ends Here
#=====

if(FALSE)
{
#GradDescent
#https://www.kaggle.com/bryanb2003/gradient-descent-machine-learning-in-r GradD <- function(x,
y, alpha = 0.006, epsilon = 10^-10){
iter <- 0
i<- 0
x<- cbind(rep(1,nrow(x)), x)
theta <- matrix(c(1,1),ncol(x),1)

```

```

cost <- (1/(2*nrow(x))) * t(x %>% theta - y) %>% (x %>% theta - y)
delta <- 1
while(delta > epsilon){ i <- i + 1
  theta <- theta - (alpha / nrow(x)) * (t(x) %>% (x %>% theta - y))
  cval <- (1/(2*nrow(x))) * t(x %>% theta - y) %>% (x %>% theta - y)
  cost <- append(cost, cval)
  delta <- abs(cost[i+1] - cost[i])
  if((cost[i+1] - cost[i]) > 0){
    print("The cost is increasing. Try reducing alpha.")
    return()
  }
  iter <- append(iter, i)
}
print(sprintf("Completed in %i iterations.", i))
return(theta)
}

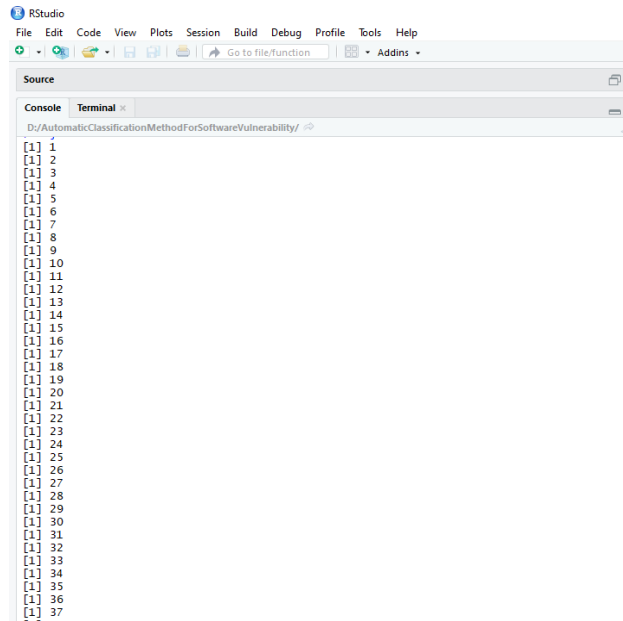
```

```

data("iris")
library(gradDescent)
x <- as.matrix(iris[,c(2:4)])
y <- as.matrix(iris[,1])
stheta <- GradD(scale(x), y, alpha = 0.006, epsilon = 10^-10)
stheta
data("cars")
x <- as.matrix(cars$speed)
y <- as.matrix(cars$dist)
theta <- GradD(x, y, alpha = 0.006, epsilon = 10^-10)
theta
}

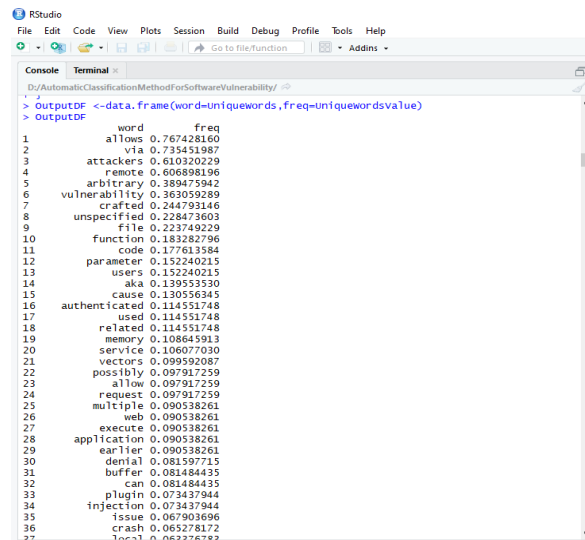
```

B. SCREEN SHOTS



The screenshot shows the RStudio interface with the console pane active. The console displays a list of 37 items, each preceded by a line number in square brackets. The items are:

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
[1] 11
[1] 12
[1] 13
[1] 14
[1] 15
[1] 16
[1] 17
[1] 18
[1] 19
[1] 20
[1] 21
[1] 22
[1] 23
[1] 24
[1] 25
[1] 26
[1] 27
[1] 28
[1] 29
[1] 30
[1] 31
[1] 32
[1] 33
[1] 34
[1] 35
[1] 36
[1] 37
```



The screenshot shows the RStudio interface with the console pane active. The console displays the output of a data frame creation command. The output is a data frame with two columns: 'word' and 'freq'. The data is as follows:

```
> outputDF <- data.frame(word=uniquewords, freq=uniquewordsvalue)
> outputDF
  word      freq
1  allows 0.767428160
2    via 0.735451987
3 attackers 0.610320229
4   remote 0.606898196
5 arbitrary 0.389475942
6 vulnerability 0.363059289
7   crafted 0.244793146
8 unspecified 0.228473603
9    file 0.223749229
10 function 0.183282796
11   code 0.177613584
12 parameter 0.152240215
13   users 0.152240215
14   aka 0.139553530
15   cause 0.130556345
16 authenticated 0.114551748
17   used 0.114551748
18   related 0.114551748
19   memory 0.108645913
20 service 0.106077030
21 vectors 0.099592087
22 possibly 0.097917259
23   allow 0.097917259
24 request 0.097917259
25 multiple 0.090538261
26   web 0.090538261
27 execute 0.090538261
28 application 0.090538261
29   earlier 0.090538261
30 denial 0.081597715
31 buffer 0.081484435
32   can 0.081484435
33 plugin 0.073437944
34 injection 0.073437944
35 issue 0.067903696
36 crash 0.065278172
37 1sec-1 0.063276782
```

RStudio

File Edit Code View Plots Session Build Debug Profile Tools Help

Source

Console Terminal

```
D:/AutomaticClassificationMethodForSoftwareVulnerability/
36 create 0.009270274
37 local 0.063376783
38 vulnerabilities 0.063376783
39 discovered 0.058849870
40 impact 0.057118401
41 demonstrated 0.054322956
42 note 0.054322956
43 php 0.049796043
44 sql 0.045269130
45 attacks 0.045269130
46 access 0.045269130
47 execution 0.040742217
48 information 0.040742217
49 data 0.040742217
50 http 0.040742217
51 input 0.040742217
52 stack 0.036215304
53 files 0.036215304
54 server 0.036215304
55 read 0.031688391
56 attacker 0.031688391
57 conduct 0.031688391
58 write 0.031688391
59 heapbased 0.031688391
60 cve 0.031688391
61 overread 0.027161478
62 bug 0.027161478
63 security 0.027161478
64 directory 0.022634565
65 user 0.022634565
66 inject 0.022634565
67 url 0.022634565
68 crosssite 0.022634565
69 properly 0.022634565
70 library 0.022634565
71 linux 0.022634565
72 overflow 0.022634565
73 kernel 0.022634565
```

RStudio

File Edit Code View Plots Session Build Debug Profile Tools Help

Source

Console Terminal

```
D:/AutomaticClassificationMethodForSoftwareVulnerability/
+ outputDF<- rbind( outputDF,data.frame(Feature= Allwords[i] , Gainvalue =val1 )
+ )
+ }
+ }
> outputDF
  Feature Gainvalue
1 xss 0.018107652
2 service 0.106077030
3 via 0.735451987
4 denial 0.081597715
5 web 0.090538261
6 attackers 0.610320229
7 remote 0.606898196
8 cause 0.130556345
9 scripting 0.018107652
10 crosssite 0.022634565
11 allows 0.767428160
12 inject 0.022634565
13 arbitrary 0.389475942
14 script 0.022634565
15 html 0.013580739
16 parameter 0.152240215
17 vulnerability 0.363039289
18 crafted 0.244793146
19 function 0.183282796
20 unspecified 0.228473603
21 buffer 0.081484435
22 files 0.036215304
23 issue 0.067903696
24 sql 0.045269130
25 information 0.040742217
26 code 0.177613584
27 application 0.090538261
28 crash 0.065278172
29 obtain 0.018107652
30 possibly 0.097917259
31 file 0.223749229
```



```

RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function Addins

Source

Console Terminal
D:/AutomaticClassificationMethodForSoftwareVulnerability/
[461] 0.4620000000 0.4630000000 0.4640000000 0.4650000000 0.4660000000
[466] 0.4670000000 0.4680000000 0.4690000000 0.4700000000 0.4710000000
[471] 0.4720000000 0.4730000000 0.4740000000 0.4750000000 0.4760000000
[476] 0.4770000000 0.4780000000 0.4790000000 0.4800000000 0.4810000000
[481] 0.4820000000 0.4830000000 0.4840000000 0.4850000000 0.4860000000
[486] 0.4870000000 0.4880000000 0.4890000000 0.4900000000 0.4910000000
[491] 0.4920000000 0.4930000000 0.4940000000 0.4950000000 0.4960000000
[496] 0.4970000000 0.4980000000 0.4990000000 0.5000000000 0.5010000000
[501] 0.5020000000 0.5030000000 0.5040000000 0.5050000000 0.5060000000
[506] 0.5070000000 0.5080000000 0.5090000000 0.5100000000 0.5110000000
[511] 0.5120000000 0.5130000000 0.5140000000 0.5150000000 0.5160000000
[516] 0.5170000000 0.5180000000 0.5190000000 0.5200000000 0.5210000000
[521] 0.5220000000 0.5230000000 0.5240000000 0.5250000000 0.5260000000
[526] 0.5270000000 0.5280000000 0.5290000000 0.5300000000 0.5310000000
[531] 0.5320000000 0.5330000000 0.5340000000 0.5350000000 0.5360000000
[536] 0.5370000000 0.5380000000 0.5390000000 0.5400000000 0.5410000000
[541] 0.5420000000 0.5430000000 0.5440000000 0.5450000000 0.5460000000
[546] 0.5470000000 0.5480000000 0.5490000000 0.5500000000 0.5510000000
[551] 0.5520000000 0.5530000000 0.5540000000 0.5550000000 0.5560000000
[556] 0.5570000000 0.5580000000 0.5590000000 0.5600000000 0.5610000000
[561] 0.5620000000 0.5630000000 0.5640000000 0.5650000000 0.5660000000
[566] 0.5670000000 0.5680000000 0.5690000000 0.5700000000 0.5710000000
[571] 0.5720000000 0.5730000000 0.5740000000 0.5750000000 0.5760000000
[576] 0.5770000000 0.5780000000 0.5790000000 0.5800000000 0.5810000000
[581] 0.5820000000 0.5830000000 0.5840000000 0.5850000000 0.5860000000
[586] 0.5870000000 0.5880000000 0.5890000000 0.5900000000 0.5910000000
[591] 0.5920000000 0.5930000000 0.5940000000 0.5950000000 0.5960000000
[596] 0.5970000000 0.5980000000 0.5990000000 0.6000000000 0.6010000000

slot "error":
[1] 0.002742832

slot "biasweight":
[1] 0.5634711

slot "output":
[1] 0.9461584

```

```

RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function Addins

Source

Console Terminal
D:/AutomaticClassificationMethodForSoftwareVulnerability/

Output Layer 1 : 0.02656613
Output Layer 2 : 0.02683825
Output Layer 3 : 0.02711315
Output Layer 4 : 0.02739087
Output Layer 5 : 0.02767144
Output Layer 6 : 0.02795488
Output Layer 7 : 0.02824122
Output Layer 8 : 0.0285305
Output Layer 9 : 0.02882274
Output Layer 10 : 0.02911797
Output Layer 11 : 0.02941622
Output Layer 12 : 0.02971754
Output Layer 13 : 0.03002193
Output Layer 14 : 0.03032945
Output Layer 15 : 0.03064011
Output Layer 16 : 0.03095396
Output Layer 17 : 0.03127102
Output Layer 18 : 0.03159133
Output Layer 19 : 0.03191493
Output Layer 20 : 0.03224183
Output Layer 21 : 0.03257209
Output Layer 22 : 0.03290572
Output Layer 23 : 0.03324278
Output Layer 24 : 0.03358328
Output Layer 25 : 0.03392728
Output Layer 26 : 0.0342748
Output Layer 27 : 0.03462587
Output Layer 28 : 0.03498055
Output Layer 29 : 0.03533885
Output Layer 30 : 0.03570083
Output Layer 31 : 0.03606652
Output Layer 32 : 0.03643595
> cat(retResult)# calling convenience
0.02656613 0.02683825 0.02711315 0.02739087 0.02767144 0.02795488 0.02824122 0.0285305
0.02882274 0.02911797 0.02941622 0.02971754 0.03002193 0.03032945 0.03064011 0.03095396
0.03127102 0.03159133 0.03191493 0.03224183 0.03257209 0.03290572 0.03324278 0.03358328
0.03392728 0.0342748 0.03462587 0.03498055 0.03533885 0.03570083 0.03606652 0.03643595

```



```

RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function Addins

Source
Console Terminal
D:/AutomaticClassificationMethodForSoftwareVulnerability/
> train()
[1] 1
[11]
An object of class "Neuron"
Slot "inputs":
[1] 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0
.5883027
[10] 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0
.5883027
[19] 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0
.5883027
[28] 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0
.5883027
[37] 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0
.5883027
[46] 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0
.5883027
[55] 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0
.5883027
[64] 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0
.5883027
[73] 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0
.5883027
[82] 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0
.5883027
[91] 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0
.5883027
[100] 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0
.5883027
[109] 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0.5883027 0
.5883027
[118] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0
.0000000
[127] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0
.0000000
[136] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0
.0000000
[145] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0
.0000000

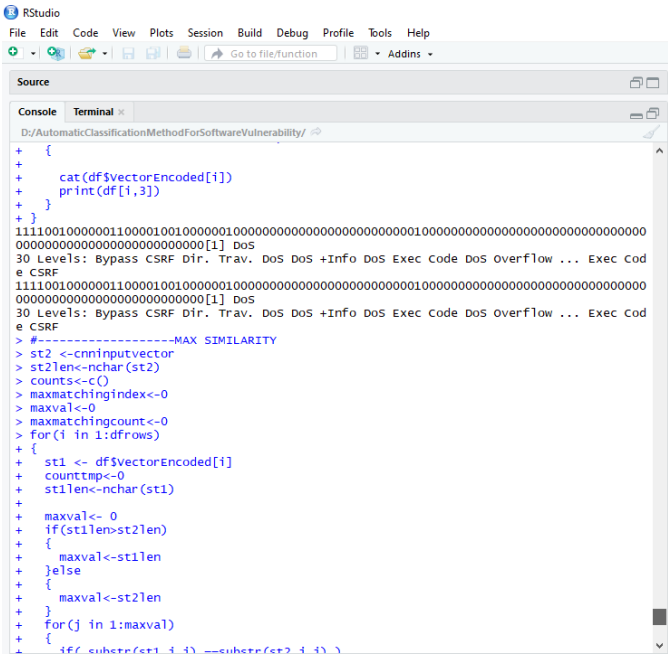
```

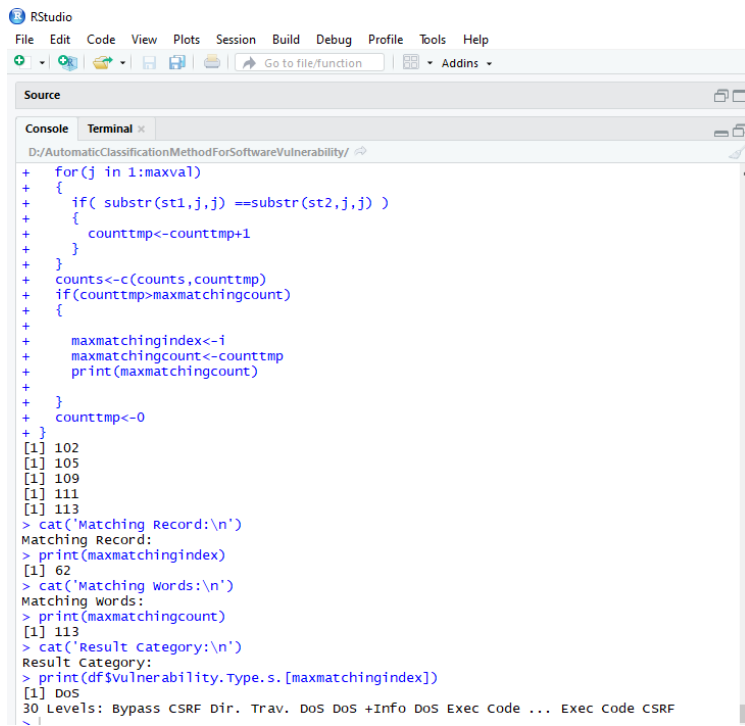
```

RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function Addins

Source
Console Terminal
D:/AutomaticClassificationMethodForSoftwareVulnerability/
> for(i in 1:10)
+ {
+   Mat[i,1]<-paste('User',i,sep='')
+   Mat[i,2]<- as.integer( runif(1) * inputlayercount)
+ }
> cat('User\tsongtypeinterested:\tsongcode(p4,g5,c5,w5,l5)\tsongcharacteristic\n')
User SongTypeInterested: SongCode(p4,g5,c5,w5,l5) songcharacteristic
> cat('-----\n')
> Matcount<-nrow(Mat)
> for(i in 1:Matcount)
+ {
+   cat(Mat[i,1],'\t',Mat[i,2],'\t',MatSongTypes[i,2],'\t', Matchcharacteristics[as.in
teger( Mat[i,2]),2],'\n')
+ }
User1 952 000100001000010000100001 0.03002193
User2 2105 0001000010000100001000010 0.03462587
User3 21 0001000010000100001000100 0.02656613
User4 1735 000100001000010000101000 0.0324278
User5 264 00010000100001000010000 0.02739087
User6 2354 000100001000010001000001 0.03606652
User7 1260 0001000010000100010000010 0.03127102
User8 988 000100001000010001000100 0.03002193
User9 1583 000100001000010001001000 0.03257209
User10 2211 000100001000010001010000 0.03533885
> MatMusicfeaturecharacteristics<-Matchcharacteristics
> #-----
>
>
> #Matchcharacteristics[20,2]
>
> #BELOW LINES ARE NOT NECESSARY INSTEAD OF COMMENT - IF CONDITION IS WRITTEN SO THAT
CODE WILL NOT WORK
> if(FALSE)
+ {
+   #DUMP FUNCTION LOGIC STARTS
+   for(i in 1:nrow)

```





The screenshot shows the RStudio environment. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for saving, running, and other functions. The main window is divided into two panes: 'Source' and 'Console'. The 'Source' pane shows an R script with the following code:

```
+ for(j in 1:maxval)
+ {
+   if( substr(st1,j,j) ==substr(st2,j,j) )
+   {
+     counttmp<-counttmp+1
+   }
+ }
+ counts<-c(counts,counttmp)
+ if(counttmp>maxmatchingcount)
+ {
+   maxmatchingindex<-j
+   maxmatchingcount<-counttmp
+   print(maxmatchingcount)
+ }
+ counttmp<-0
+ }
```

The 'Console' pane shows the output of the script:

```
[1] 102
[1] 105
[1] 109
[1] 111
[1] 113
> cat('Matching Record:\n')
Matching Record:
> print(maxmatchingindex)
[1] 62
> cat('Matching words:\n')
Matching words:
> print(maxmatchingcount)
[1] 113
> cat('Result category:\n')
Result category:
> print(df$vulnerability.type.s.[maxmatchingindex])
[1] DoS
30 Levels: Bypass CSRF Dir. Trav. DoS DoS +Info DoS Exec Code ... Exec Code CSRF
~ |
```

REFERENCES

- [1] S. A. I. B. S. Arachchi and I. Perera, (May2018) “Continuous integration and continuous delivery pipeline automation for agile software project management,” in Proc. Moratuwa Eng. Res. Conf.(MERCon), pp. 156–161.
- [2] B. Fitzgerald, and K.-J. Stol, —(2017)Continuous Software Engineering: A Roadmap and Agenda,||Journal of Systems and Software, vol.123,.
- [3] M. Fowler.(21/10/2015) "Continuous Integration. Available at: <http://martinfowler.com/articles/continuousIntegration.html>;
- [4] G. Huang, Y. Li, Q. Wang, J. Ren, Y. Cheng, and X. Zhao, (2019)“Automatic classification method for software vulnerability based on deep neural network,” IEEE Access, vol. 7, pp. 28291– 28298,.
- [5] J. Humble, and D. Farley,(2010) Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment automation, 1st edition ed.: Addison-Wesley Professional,.
- [6] M. Leppanen, S. Makinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mantyla, and T. Mannisto, (2015)—The Highways and Country Roads to Continuous Deployment, IEEE Software, vol. 32, no. 2, pp. 64-72,.
- [7] A. Phillips, M. Sens, A. de Jonge, and M. van Holsteijn, (2015)The IT Manager’s Guide to Continuous Delivery: Delivering business value in hours, not months: XebiaLabs.
- M. Qasaimeh, A. Shamlawi, and T. Khairallah, (2015)“Black box evaluation of Web application scanners: Standards mapping approach,” J. Theor. Appl. Inf. Technol., vol. 96, no. 14, pp. 4584–4596.
- [8] Trevor Hastie, Robert Tibshirani and Jerome Friedman, The Elements of Statistical Learning: Data Mining Inference and Prediction Second Edition[M], Springer, pp. 10-1, 2009.
- [9] F. Yamaguchi, M. Lottmann, and K. Rieck, (2012)“Generalized vulnerability extrapolation using abstract syntax trees,” in Proc. 28th Annu. Comput. Secur. Appl. Conf. (ACSAC), , pp. 359–368.
- [10] M. Zolanvari, M. A. Teixeira, L. Gupta, K. M. Khan, and R. Jain, (Aug. 2019 “Machine learning-based network vulnerability analysis of industrial Internet of Things,” IEEE Internet Things J., vol. 6,no. 4, pp. 6822–6834.