Identifying Vulnerable Functions from Source Code using Vulnerability Reports

Rabaya Sultana Mim, Toukir Ahammed and Kazi Sakib

Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh

Abstract

Software vulnerability represents a flaw within a software product that can be exploited to cause the system to violate its security. In the context of large and evolving software systems, developers find it challenging to identify vulnerable functions effectively when a new vulnerability is reported. Existing studies have underutilized vulnerability reports which can be a good source of contextual information in identifying vulnerable functions in source code. This study proposes an information retrieval based method called Vulnerable Functions Detector (VFDetector) for identifying vulnerable functions from source code and vulnerability reports. VFDetector ranks vulnerable functions based on the textual similarity between the vulnerability report corpora and the source code corpora. This ranking is achieved modifying conventional Vector Space Model to incorporate the size of a function which is known as the tweaked Vector Space Model (tVSM). As an initial study, the approach has been evaluated by analysing 10 vulnerability reports from six popular open-source projects. The result shows that VFDetector ranks the actual vulnerable function at first position in 40% cases. Moreover, it ranks the actual vulnerable function within rank 5 in 90% cases and within rank 7 for all analysed reports. Therefore, developers can use these results to implement successful patches on vulnerable functions more quickly .

Keywords

vulnerability identification, vulnerable function, vulnerability report, source code, vector space model

1. Introduction

A software vulnerability is a flaw, weakness, or error in a computer program or system that can be exploited by malicious attackers to compromise its integrity, availability, or confidentiality [1]. Software vulnerabilities make software systems increasingly vulnerable to attack and damage, which raises security concerns [2].

Developers need to spend a lot of time in identifying vulnerable function from large codespace when a new vulnerability is reported. Identifying vulnerable functions effectively is a perquisite of writing a patch for the reported vulnerability. This is essential for enhancing software security by addressing vulnerabilities to mitigate potential risks and threats more effectively at earliest time.

Existing studies have focused on detecting software vulnerabilities employing text-based [3, 4, 5] or graph-based [6, 7] approaches. These approaches either treat source code as plain text or apply graph analysis by representing the source code as graph. In practice, prior text-based studies treat source code as plain text and apply static program analysis or natural language processing. However, the performance of these approaches is not optimal for disregarding the source code semantics. On the other hand, graph based approaches conduct program analysis which represent the source code semantics as a graph, and then apply graph analysis methods such as Graph Neural Networks (GNN) [8] to identify vulnerabilities. Although these graph-based approaches are more efficient at identifying vulnerabilities taking into account the semantic relationship of various lines of source code but their scalability is substantially less than that of text-based approaches.

However, existing studies have underutilized vulnerability reports which can be a good source of contextual information to detect vulnerability in source code. In this context, the current study aims to verify the role of vulnerability reports in identifying vulnerable function. Vulnerability report can contain contextual information about a vulnerability which may be used to identify vulnerable functions. When a function is vulnerable against a scenario some keywords should be shared between that function and the vulnerability report. These motivate the authors to study whether vulnerable functions can be identified by analysing the source code and vulnerability report.

For this purpose, this study proposes a technique of automatic software vulnerable function identification namely VFDetector. It takes all source code files of a system as input. First, it extracts all source code functions of that system. Then static analysis is performed to extract the contents of those functions. Several text pre-processing analysis such as tokenization, stopwords removal, multiword splitting, semantic meaning extraction and lemmatization are applied on these source code along with the vulnerability report to produce code and

QuASoQ 2023: 11th International Workshop on Quantitative Approaches to Software Quality, December 04, 2023, Seoul, Korea

msse1730@iit.du.ac.bd (R. S. Mim); toukir@iit.du.ac.bd (T. Ahammed); sakib@iit.du.ac.bd (K. Sakib)

report corpora. In addition, programming language specific keywords is removed for generating code corpora. Finally, to rank the vulnerable functions, similarity scores are measured between the code corpora of the functions and report corpora by a modified version of Vector Space Model (tVSM) where larger methods get more weight while ranking.

In experiments, as an initial study ten Common Vulnerabilities and Exposures (CVE) reports are chosen randomly from six open source GitHub repositories. Based on the commit link available in reports we crawled the corresponding projects before the vulnerability was patched. The result analysis shows that VFDetector ranks the vulnerable functions at the first position in 40% cases, whereas it ranks the actual vulnerable function within top 5 in 90% cases and within top 7 in 100% cases.

It is evident from the results that VFDetector performs promisingly in detecting vulnerable functions against a vulnerability report in a large scale software systems. It is also observed that in Top 5 and Top 7 ranking, the functions which ranks above the actual vulnerable function are the related functions of that vulnerability which acquires higher similarity. It guides a developer to patch those related functions too in order to mitigate that vulnerability from the system.

The remainder of this paper is structured as follows: Section 2 gives an overview of previous studies on vulnerability detection at file level or function level. Section 3 describes our methodology for detecting vulnerable functions in a project. Section 4 reports our experimental findings and the analysis thereof. Section 5 demonstrates the threats to validity of our work. Section 6 motivates future research directions and concludes this paper.

2. Related Work

In recent years, the research community has directed significant attention toward the issue of vulnerability detection, primarily due to the complex challenges it presents. The existing body of literature has introduced numerous methodologies in response to these challenges. These methods can be classified into three distinct categories based on the degree of automation: manual, semi-automatic, and fully automatic techniques.

Manual techniques rely on human experts to create vulnerability patterns. However, all patterns can not be generated manually, which leads to reduced detection efficiency in practical scenarios. In contrast, semi-automatic techniques involve human experts in the extraction of specific features like API symbols [9] and function calls [10], which are then fed into traditional machine learning models for vulnerability detection. Full-automatic techniques utilize Deep Learning (DL) to automatically extract features and construct vulnerability patterns without manual expert intervention. Recently, DL based techniques [11, 12, 13] has gained extensive use in detecting source code vulnerabilities due to its ability to automatically extract features from source code. DL based methods can be categorized into text-based and graph-based methods.

Text based methods: The text-based approach in vulnerability detection treats a program's source code as text and employs natural language processing techniques to identify vulnerabilities. Russell et al. [3] introduced the TokenCNN model, which utilizes lexical analysis to acquire source code tokens and employs a Convolutional Neural Network (CNN) to detect vulnerabilities.

Li et al. [4] proposed Vuldeepecker, a method that collects code gadgets by slicing programs and transforms them into vector representations, training a Bidirectional Long Short Term Memory (BLSTM) model for vulnerability recognition.

Zhou et al. [5] introduced μVulDeePecker, which enhances Vuldeepecker by incorporating code attention with control dependence to detect multi-class vulnerabilities. However, the performance of these text-based approaches is limited because they rely solely on static source code analysis and do not account for the program semantics.

Graph based methods: To address the limitations of text-based methods, researchers have turned to dynamic program analysis to convert a program's source code semantics into a graph representation facilitating vulnerability detection through graph analysis. Zhou et al. [6] introduced Devign which employs a graph neural network for vulnerability identification. This approach includes a convolutional module that efficiently extracts critical features for graph-level classification from the learned node representations. By pooling the nodes, a comprehensive representation for graph-level classification is achieved.

Cheng et al. [7] introduced a different approach named Deepwukong which divides the program dependency graph into various subgraphs after distilling the program semantics based on program points of interest. These subgraphs are then utilized to train a vulnerability detector through a graph neural network. While these graph-based techniques prove more effective in identifying vulnerabilities but it is important to note that their scalability is worse than text based methods due to large number of graph nodes in complex program.

Exploring the existing literature, it is evident that text-based methods lacks in incorporating program semantics while graph-based methods achieve high accuracy considering source code semantics but have scalability issues in complex scenarios. Moreover, due to the underutilization of contextual information like vulnerability reports with source code existing methods fails to detect complicated vulnerabilities in real-world projects. Because whenever

a new vulnerability is reported in a system it is hard to detect in which function the vulnerability exist as the system consist of huge volume of functions. Before using vulnerable reports as a source of contextual information in existing methods, it is important to verify whether vulnerable functions can be identified effectively using these reports. Moreover, identifying vulnerable functions using vulnerability reports can play an effective role to minimize the search space in existing methods.

3. Methodology

This study proposes an approach which detects vulnerable functions from huge volume of files of a large software system using vulnerability reports. The overall process of this approach consist of three distinct steps and those are Source Code Corpora Generation, Vulnerability Report Corpora Generation, Ranking Vulnerable Functions. Each of these steps encompasses a series of tasks as illustrated in Figure 1. At first, all files and their corresponding functions are extracted from a particular version of a software system. Then these source code is processed to create code corpora. Similarly vulnerability report is processed to produce report corpora. Finally, similarity between the report and code corpora is measured using tweaked Vector Space Model (tVSM) to rank the vulnerable source code functions.

3.1. Dataset

We used the benchmark dataset Big-Vul[1] developed by Fan et al. [14]. This dataset comprises reliable and comprehensive code vulnerabilities which are directly linked to the publicly accessible CVE database. Notably, the creation of this dataset involved a significant investment of manual resources to ensure its high quality. Furthermore, this dataset is noteworthy for its substantial scale, being one of the most extensive vulnerability datasets available. It is derived from a collection of 348 open-source Github projects, encompassing a time span from 2002 to 2019, and covers 91 distinct Common Weakness Enumeration (CWE) categories. This comprehensive dataset comprises approximately 188,600 C/C++ functions, with 5.6% of them identified as vulnerable (equivalent to 10,500 vulnerable functions). This dataset provides granular ground-truth information at the function level, specifying which functions within a codebase are susceptible to vulnerabilities.

3.2. Source Code Corpora Generation

Source code corpora consist of source code terms used to assess similarity with vulnerability report corpora. Therefore, the precision of code corpora generation directly impacts the precision of matching, consequently enhancing the accuracy of vulnerability localization. In this step all the folders are extracted from a system with their corresponding C/C++ files. From each of these files all functions are extracted automatically in individual C files which ensures function level analysis. For Example: CVE-2014-2038 of Linux version 3.13.5 consist of 15,675 files which has total 229,682 functions.

This stage generates a vector of lexical tokens by doing lexical analysis on every source code file. There are unnecessary tokens in source code which do not contain any vulnerability related information. These tokens are discarded from source code such as programming language specific keywords (e.g., int, if, float, switch, case, struct), stop words (e.g., all, and, an, the). Many words in the source code may include multiple words. For example, the term "addRequest" consists of the keywords "add" and "Request". Mutiwords are separated using multi word identifier. Furthermore, statements are divided according to certain syntax-specific separators like ‘ , ’, ‘=’, ‘(’, ‘)’, ‘{’, ‘}’, ‘/’, and so on. WordNet[2] is used to derive each word’s semantic meanings because a term might have more than one synonym. In specific cases, developers and Quality Assurance (QA) personnel may employ different terminology, even though they are referring to the same scenario with equivalent meanings. For example, the term ‘finalize’ may have multiple synonyms such as ‘conclude’ or ‘complete.’ When describing a situation, if a developer uses ‘finalize’ but QA opts for ‘conclude’, it’s challenging for the system to identify these variances without considering the semantic meanings of these words. Therefore, the extraction of semantic meaning is crucial in achieving accurate rankings for vulnerable functions.

The final stage of code corpora generation incorporates WordNet lemmatization, a technique that normalizes words to their base or dictionary form. WordNet lemmatization utilizes the comprehensive WordNet lexical database, organizing words into synonymous sets called synsets. This method identifies word lemmas based on the word’s part of speech and context within WordNet, offering a more context-aware approach to lemmatization. As a result, it considers a word’s meaning and contextual usage, allowing for precise reduction of words. For instance, it transforms "running" to "run" and "better" to "good" based on their meanings and parts of speech, unlike standard lemmatization that typically relies on suffix removal.

[1] https://github.com/ZeoVan/MSR_20_Code_vulnerability_CSV_Dataset

[2] https://wordnet.princeton.edu/

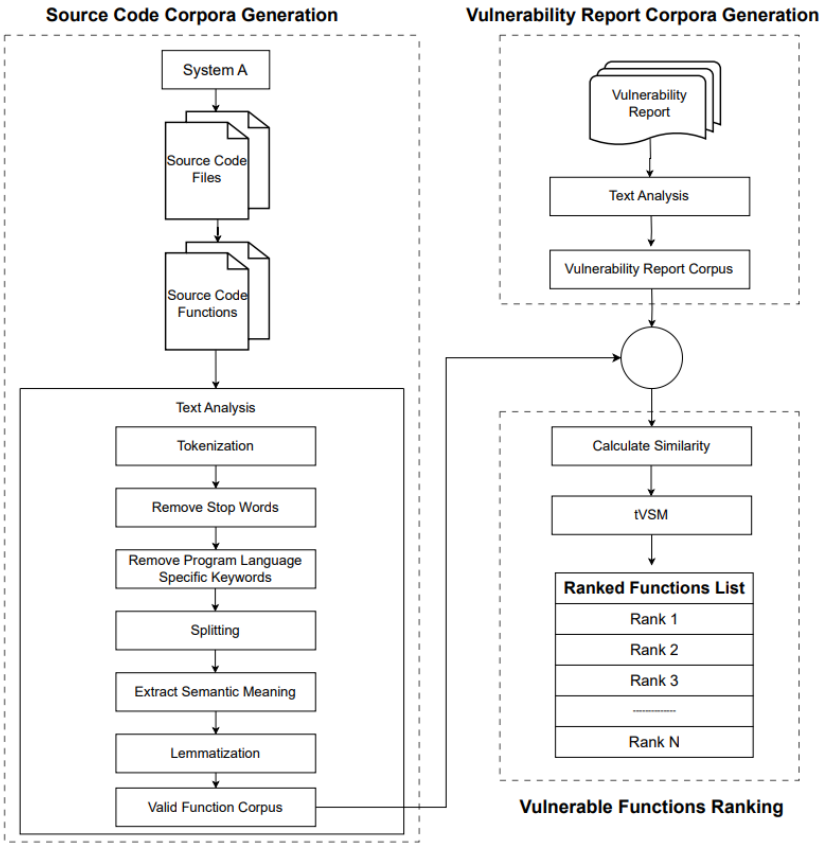

Figure 1: Overview of VFDetector

3.3. Vulnerability Report Corpora

A software vulnerability report contains information like description about the vulnerability, severity rating, vulnerability identifier (CVE-ID), reference to additional sources which gives valuable insights about a software vulnerability issue. However, these reports can also include irrelevant terms such as stop words and words in various tenses (present, past, or future). To refine vulnerability reports, pre-processing is necessary. In the initial stage of vulnerability report corpora creation, stop words are eliminated. We apply WordNet Lemmatizer, similar to what's used for source code corpora generation, to generate refined report corpora containing only relevant terms.

3.4. Ranking Vulnerable Functions

In this step, relevant vulnerable functions are ranked based on the textual similarity between the query (report corpus) and each of the function in the code corpus. Vulnerable functions are ranked by applying tVSM. We employ tVSM, which modifies the Vector Space Model (VSM) by emphasising large-scale functions. In traditional VSM, the cosine similarity is used to measure the ranking score between the associated vector representations of a report corpus (r) and function (f), according to Equation 1.

$$Similarity(r, f) = cos(r, f) = \frac{\vec{V_r} \cdot \vec{V_f}}{|\vec{V_r}| \cdot |\vec{V_f}|} \quad (1)$$

Here, $\vec{V_r}$ and $\vec{V_f}$ are the term vectors for the vulnerability report (r) corpus and function (f) corpus respectively. Throughout the years, numerous adaptations of the tf(t,d) function have been introduced with the aim of enhancing the VSM model's effectiveness. These encompass logarithmic, augmented, and Boolean modifications of the traditional VSM [15]. It has been noted that the logarithmic version can yield improved performance, as indicated by prior studies [16, 17, 18]. From that point of view, tVSM modified Equation 1 and uses the logarithm of term frequency (tf) and iff(inverse function frequency) to give more importance on rare terms in the functions.

Thus tf and iff are calculated using Equation 2 and 3 respectively.

$$tf(t, f) = 1 + logf_{tf} \tag{2}$$

$$iff = log(\frac{\#functions}{n_t}) \tag{3}$$

Here, f_{tf} represents the frequency of a term t appearing in a function f, $\#functions$ denotes the total count of functions within the search space, n_t signifies the overall number of functions that include the term t. Thus in equation 4 each term weight is calculated as follows:

$$\begin{aligned} weight_{t \in f} &= (tf)_{tf} \times (iff)_t \\ &= (1 + logf_{tf}) \times log(\frac{\#functions}{n_t}) \end{aligned} \tag{4}$$

The VSM score is calculated using equation 5.

$$cos(r, f) = \sum_{t \in r \cap f} (1 + \log f_{tr}) \times (1 + \log f_{tf}) \times iff^2 \times \frac{1}{\sqrt{\sum (1 + \log f_{tr}) \times iff^2}} \times \frac{1}{\sqrt{\sum (1 + \log f_{tf}) \times iff^2}} \tag{5}$$

Traditional VSM tends to give preference to smaller functions when ranking them, which can be problematic for large functions because they may receive lower similarity scores. Past research [19, 20, 21] has indicated that larger source code files are more likely to contain vulnerabilities. Therefore, in the context of vulnerability localization, it's crucial to prioritize larger functions in our ranking. To address this issue, we introduce a function denoted as 'x' (as shown in Equation 6) within the tVSM model, aiming to account for the function's length.

$$x(terms) = 1 - e^{-Normalize(\#terms)} \tag{6}$$

Equation 6 represents a logistic function, specifically an inverse logit function, designed to ensure that larger functions receive higher rankings. We employ Equation 6 to calculate the length value for each source code function based on the number of terms contained within the function. Here we apply the normalized value of '#terms' as the argument for the exponential function e^{-x}. The normalization process is defined in Equation 7.

Let z denote a set of data, with z_{max} and z_{min} representing the maximum and minimum values of z term, respectively. The normalized value for z term is determined as:

$$Normalize(z) = \frac{z - z_{min}}{z - z_{max}} \tag{7}$$

Considering the above analysis, tVSM score is calculated by multiplying the weight of each function, denoted as x(terms), with the cosine similarity score represented by cos(r, f), as described in Equation 8:

$$tVSM(r, f) = x(terms) \times cos(r, f) \tag{8}$$

Once the tVSM score for each function has been computed, a list of vulnerable functions is arranged in descending order of scores. The function with the highest similarity score is positioned at the top of the ranked list.

4. Experiment and Result Analysis

This section provides information on the practical implementation, the criteria used for evaluation and experimental result analysis of this study.

4.1. Implementation

The proposed method is implemented in python (version 3.11.5). The experiment was conducted on an Windows server equipped with an Intel(R) Core(TM) i5-10300H CPU processor @3.0GHz and having 64GB of RAM. The implementation involves various python libraries and NLTK (Natural Language Toolkit) libraries for text processing and feature extraction. It takes function files as input and provides ranking of suspicious vulnerable functions as output.

Table 1
List of Analyzed Open Source Projects

#	Project Name	CVE ID	Source Code
1	Chrome	CVE-2011-3916	github.com/chromium/chromium/tree/f1a59e0513d63758588298e98500cac82ddccb67
2	Radare2	CVE-2017-16359	github.com/radareorg/radare2/tree/1f5050868eedabcbf2eda510a05c93577e1c2cd5
3	Linux	CVE-2013-6763	github.com/torvalds/linux/tree/f9ec2e6f7991e748e75e324ed05ca2a7ec360ebb
4	Linux	CVE-2013-2094	github.com/torvalds/linux/tree/41ef2d5678d83af030125550329b6ae8b74618fa
5	Linux	CVE-2014-2038	github.com/torvalds/linux/tree/a9ab5e840669b19aca2974e2c771a77df2876434
6	ImageMagick	CVE-2017-15033	github.com/ImageMagick/ImageMagick/tree/c29d15c70d0eda9d7ffe26a0ccc181f4f0a07ca5
7	Tcpdump	CVE-2017-13000	github.com/the-tcpdump-group/tcpdump/tree/a7e5f58f402e6919ec444a57946bade7dfd6b184
8	Tcpdump	CVE-2018-14470	github.com/the-tcpdump-group/tcpdump/tree/aa3e54f594385ce7e1e319b0c84999e51192578b
9	FFmpeg	CVE-2016-10190	github.com/FFmpeg/FFmpeg/tree/51020adcecf4004c1586a708d96acc6cbddd050a
10	FFmpeg	CVE-2019-11339	github.com/FFmpeg/FFmpeg/tree/3f086a2f665f9906e0f6197cddbfacc2f4b093a1

Table 2
Summary of Tested Projects

CVE ID	Total Commits	Total Files	Total Functions	Vulnerable Functions	VFDetector Ranking
CVE-2017-13000	4,466	180	638	`extract_header_length()`	1
CVE-2018-14470	4,548	185	640	`babel_print_V2()`	1
CVE-2017-15033	12,558	586	4,694	`ReadYUVImage()`	1
CVE-2017-16359	16,362	965	9,197	`store_versioninfo_gnu_verdef()`	1
CVE-2011-3916	93,104	4,929	15,042	`WebGLObject()`	2
CVE-2019-11339	93,322	2,572	16,724	`mpeg4_decode_studio_block()`	2
CVE-2016-10190	82,768	2,286	14,713	`http_buf_read()`	3
CVE-2014-2038	413,259	15,674	229,682	`nfs_can_extend_write()`	4
CVE-2013-2094	362,534	18,358	257,550	`perf_swevent_init()`	5
CVE-2013-6763	401,141	19,260	273,898	`uio_mmap_physical()`	7

4.2. Evaluation

To conduct this research we used the extensive Big-Vul dataset which contains large scale vulnerability reports of C/C++ code from open source GitHub projects. Other C/C++ datasets can also be used. Based on the highest number of vulnerabilities reported, we choosed top six well known projects from this dataset which are Chrome, Linux, Radare2, ImageMagick, Tcpdump and FFmpeg as shown in Table 1. As the selected projects are open-source in nature and are hosted on GitHub, serving as the primary platform for storing code and managing version control. It allows us to extract all essential commits for our analysis. Additional information about the repositories can be found in Table 2. As an initial study, VFDetector was evaluated using ten vulnerability reports from these six open-source projects which are chosen randomly from the dataset. Table 1 lists the analysed project name, CVE ID of report, and the source code link.

To measure the effectiveness of the proposed vulnerability detection method, we use the Top N Rank metric. This metric signifies the count of vulnerable functions ranked in the top N (where N can be 1, 5, or 7) in the obtained results. When assessing a reported vulnerability, if the top N query results include at least one function that corresponds to the location where the vulnerability needs to be addressed, we determine that the vulnerable function is detected successfully. Table 2 includes ten vulnerability reports from six open source projects with their number of commits, total files, total functions, actual vulnerable functions name and finally VFDetector ranking in Top N ranked functions in output. The results of Table 2 shows that among the ten CVE reports VFDetector ranks the actual vulnerable function at the 1st position for four (40%) reports which are CVE ID #13000, #14470, #15033, #16359. For five reports (50%) with CVE ID #3916, #2094, #2038, #10190 and #11339 it ranks the vulnerable function in Top 5 rank. It indicates that nine (90%) reports are ranked in Top 5. For one report CVE-2013-6763 of Linux Kernel version 3.12.1 it ranks the vulnerable function in Top 7 rank i.e., in 7^{th} position out of total 273,898 functions. Upon manual inspection, we observed that the six functions preceding the vulnerable function exhibit a higher similarity score compared to the actual vulnerable function. The reason behind this can be the inter-connectedness of these six functions with the vulnerable function through function calls. It is also noticeable that projects with less number of functions ranks the vulnerable function in 1^{st} position and with large number of functions the ranking decreases slightly. The reason behind this is larger projects might contain more associated functions which are needed to be fixed in order to address a particular vulnerability.

In summary, the experimental results show that VFDetector can detect vulnerable functions from a huge volume of functions and can also suggest developers with the related functions having highest similarity scores which might need to be patched to address the reported vulnerability. Moreover, to the best of our knowledge we are the first to incorporate vulnerability reports in software vulnerability detection from the concept that vulnerability report's description contain conceptual information about a reported vulnerability. Based on the promising results in this initial evaluation, the future work can be analyzing more vulnerable reports from diverse projects to make the approach comparable and generalizable.

5. Threats to Validity

In this section, we discussed the potential threats which may affect the validity of this study.

Threats to external validity: The generalizability of the acquired results poses a threat to external validity. The dataset that we used in our research was gathered from open-source. Open-source projects may contain data that differs from those created by software companies with sound management practices. Seven Apache projects are examined in this study. More projects from other systems are needed to be evaluated for the generalisation. However, to overcome this threat large-scale

diversified projects with long change history is to be chosen.

Threats to internal validity: One limitation of our approach is its reliance on sound programming practices when naming variables, methods, and classes. If a developer uses non-meaningful names, it could have an adverse impact on the effectiveness of vulnerability detection.ay not fully represent the characteristics of the whole program. Additionally, our model is evaluated with C/C++ functions and it may encounter challenges in detecting vulnerabilities in other programming languages.

Threats to construct validity: We used the WordNet database and lemmatizer of NLTK library as essential components in text pre-processing to extract word semantics and reduce words to their base forms. Since these resources are well known for their usefulness in NLP, we relied on their accuracy. Moreover, vulnerability reports offer essential information that developers rely on to address and patch vulnerable functions. A bad vulnerability report delays the fixing process. It's worth noting that our approach is dependent on the quality of these reports. If a vulnerability report lacks sufficient information or contains misleading details, it can have a detrimental impact on the performance of VFDetector.

6. Conclusion

Once a new vulnerability is reported, developers need to know which files and particular which function should be modified to fix the issues. This can be especially challenging in large software projects, where examining numerous source code files can be time-consuming and costly.

In this paper, a software vulnerability detection technique has been proposed named as VFDetector for detecting relevant vulnerable functions based on vulnerability reports. Since detecting vulnerabilities from vulnerability report is an information retrieval process, we apply static analysis on both source code and vulnerability reports to create code and report corpora. Finally, VFDetector leverages a tweaked Vector Space Model (tVSM) to rank the source code functions based on the similarity. Our evaluation conducted on six real-world open source projects show that VFDetector ranks vulnerable functions at the 1^{st} position in most cases.

In future, VFDetector can be applied to industrial projects to access the generalization of the results in practice. Besides, dynamic analysis can be incorporated in this approach to improve detection performance. Moreover, minimizing the search space in a function and pinpointing statement-level vulnerabilities is also a potential future scope.

References

[1] J. Han, D. Gao, R. H. Deng, On the effectiveness of software diversity: A systematic study on real-world vulnerabilities, in: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2009, pp. 127–146.

[2] H. Alves, B. Fonseca, N. Antunes, Software metrics and security vulnerabilities: dataset and exploratory study, in: 2016 12th European Dependable Computing Conference (EDCC), IEEE, 2016, pp. 37–44.

[3] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, M. McConley, Automated vulnerability detection in source code using deep representation learning, in: 2018 17th IEEE international conference on machine learning and applications (ICMLA), IEEE, 2018, pp. 757–762.

[4] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, Y. Zhong, Vuldeepecker: A deep learning-based system for vulnerability detection, in: Proceedings of the 25th Annual Network and Distributed System Security Symposium, 2018.

[5] D. Zou, S. Wang, S. Xu, Z. Li, H. Jin, μ vuldeepecker: A deep learning-based system for multiclass vulnerability detection, IEEE Transactions on Dependable and Secure Computing 18 (2019) 2224–2236.

[6] Y. Zhou, S. Liu, J. Siow, X. Du, Y. Liu, Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks, Advances in neural information processing systems 32 (2019).

[7] X. Cheng, H. Wang, J. Hua, G. Xu, Y. Sui, Deepwukong: Statically detecting software vulnerabilities using deep graph neural network, ACM Transactions on Software Engineering and Methodology (TOSEM) 30 (2021) 1–33.

[8] F. Yamaguchi, N. Golde, D. Arp, K. Rieck, Modeling and discovering vulnerabilities with code property graphs, in: 2014 IEEE Symposium on Security and Privacy, IEEE, 2014, pp. 590–604.

[9] F. Yamaguchi, M. Lottmann, K. Rieck, Generalized vulnerability extrapolation using abstract syntax trees, in: Proceedings of the 28th annual computer security applications conference, 2012, pp. 359–368.

[10] S. Neuhaus, T. Zimmermann, C. Holler, A. Zeller, Predicting vulnerable software components, in: Proceedings of the 14th ACM conference on Computer and communications security, 2007, pp. 529–540.

[11] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, Sysevr: A framework for using deep learning to detect software vulnerabilities, IEEE Transactions on Dependable and Secure Computing 19 (2021) 2244–2258.

[12] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, H. Jin, Vulcnn: An image-inspired scalable vulnerability detection system, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 2365–2376.
[13] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Y. Zhang, Z. Chen, D. Li, Vuldeelocator: A deep learning-based system for detecting and locating software vulnerabilities, IEEE Transactions on Dependable and Secure Computing (2021).
[14] J. Fan, Y. Li, S. Wang, T. N. Nguyen, A c/c++ code vulnerability dataset with code changes and cve summaries, in: Proceedings of the 17th International Conference on Mining Software Repositories, 2020, pp. 508–512.
[15] H. Schütze, C. D. Manning, P. Raghavan, Introduction to information retrieval, volume 39, Cambridge University Press Cambridge, 2008.
[16] W. B. Croft, D. Metzler, T. Strohman, Search engines: Information retrieval in practice, volume 520, Addison-Wesley Reading, 2010.
[17] S. Rahman, K. Sakib, An appropriate method ranking approach for localizing bugs using minimized search space., in: ENASE, 2016, pp. 303–309.
[18] S. Rahman, M. M. Rahman, K. Sakib, A statement level bug localization technique using statement dependency graph., in: ENASE, 2017, pp. 171–178.
[19] N. E. Fenton, N. Ohlsson, Quantitative analysis of faults and failures in a complex software system, IEEE Transactions on Software engineering 26 (2000) 797–814.
[20] T. J. Ostrand, E. J. Weyuker, R. M. Bell, Predicting the location and number of faults in large software systems, IEEE Transactions on Software Engineering 31 (2005) 340–355.
[21] H. Zhang, An investigation of the relationships between lines of code and defects, in: 2009 IEEE international conference on software maintenance, IEEE, 2009, pp. 274–283.