Measuring Software Security Using Improved CWE Base Scores

Sabrina Mamtaz Nourin[1], George Karabatis[1] and Foteini Cheirdari Argiropoulos[2]

[1]*Department of Information Systems, University of Maryland, Baltimore County (UMBC), Baltimore, Maryland, USA*
[2]*ML4Cyber, Baltimore, Maryland, USA*

Abstract

Increasing the security of a software system by decreasing the number of its vulnerabilities has been a major objective of any organization. Therefore, it is important to identify a measure that indicates the security level of the software system. This paper presents a scoring method to measure the security posture of a software system. This novel scoring method for Common Weakness Enumeration (CWE)s considers semantic information in order to increase the accuracy of the score and provides a better outlook of the security posture of a software system using full automation.

Keywords

CWE base score, software security score, context information, semantic comparison, natural language processing

1. Introduction

As software systems are quite prevalent in today's digital society, the existence of vulnerabilities in the software is often the culprit for malicious attacks, resulting in security breaches, leading to loss of information and computer assets, and inflicting disruption of operations, and financial losses. The more we examine software systems the more we discover potential vulnerabilities. Therefore, the security of software is an issue of high priority, although it is a labor intensive and complex task. The first step is to gauge the level of security in a software system and calculate it.

In this paper we propose a novel method to compute software security score using semantic knowledge and Natural Language Processing (NLP). We introduce a new scoring system for Common Weakness Enumeration (CWE), which is a community-developed list of software and hardware weakness types. Identified weaknesses in software code written in any programming language, are commonly found in CWE, which serves as a measuring stick for security tools, and as a baseline for weakness identification, mitigation, and prevention efforts [1]. There is no publicly available list of CWE scores [2]. MITRE [3] provides guidelines on how to calculate the CWE scores. But it is a time-consuming process, very user dependent and requires manual input from the user. Therefore, our objective is to provide an accurate and automated scoring method for CWEs, by implementing context similarity using NLP.

Common Vulnerabilities and Exposures (CVE) [4] provides a list that contains actual instances of the weaknesses listed in CWE. The CVE list comes with a publicly available score, along with impact and exploitability sub-scores for each vulnerability. *Impact* represents how much damage a weakness can cause if exploited, and *exploitability* measures the ease of causing damage by exploiting the weakness. Practically, the base score is the sum of impact and exploitability sub-scores [4]. CVSS (Common Vulnerability Scoring System) [5] is the official guideline to generate scores for CVE. CVSS scores are publicly available, therefore, most related research uses them to generate software security scores. However, the same CVSS score is typically assigned to the same type of vulnerabilities (all nearly similar vulnerabilities have the same score despite certain few yet substantial differences, especially when operated in different environments). Sometimes a CVSS score is not available for all possible CWEs. Based on our understanding and experimentation, using a combination of semantic information of CWE along with CVE, one can generate a more accurate software security score, because CWE provides additional information for each vulnerability and it is the basis for CVE.

The CWSS (Common Weakness Scoring System) [6] is a system that provides a mechanism for prioritizing CWEs based on a weakness scoring concept. CWSS proposes three metrics for weakness scoring – base metric, attack surface metric, and environmental metric. The base metric captures the inherent risk of the weakness, and it does not change over time and environment. Also, the base score for each CVE is generated using this base metric, and it does not change over time and system environment. The attack surface metric represents the barriers that an attacker must overcome in order to exploit the weakness. Finally, the environmental metric

PSTCI2021: 3rd International Workshop on Privacy, Security, and Trust in Computational Intelligence , November 01–05, 2021, Queensland, Australia

snourin1@umbc.edu (S. M. Nourin); georgek@umbc.edu (G. Karabatis); contact@ml4cyber.com (F. C. Argiropoulos)

CEUR Workshop Proceedings (CEUR-WS.org)

represents characteristics of the weakness that are specific to a particular environment or operational context. In this paper, we propose a solution to automatically calculate the CWE base score using NLP, without any user input. We augment the components of base metrics by utilizing semantic similarity of the CWE and CVE descriptions. By doing so, we obtain an accurate CWE score for each weakness in the system, and hence, we generate an accurate score that represents the security posture of the software.

We first follow the scoring guidelines from MITRE [2] to generate a security score, which is used as a baseline for comparison purposes against our method. The baseline yields a base score of a CWE by mapping CWE to CVE using their id, and taking the average of the base scores of mapped CVEs. Then, the baseline is used to generates the security score by taking the sum of base scores of all CWEs that exist in the software system, multiplied by their respective percentage of appearance. A higher security score means that the system is more secure and less vulnerable to possible attacks. However, this direct mapping between CWE and CVE may not provide an accurate score, because a large number of CWEs cannot be mapped with any CVE, while a small number of CWEs is mapped with multiple CVEs, introducing inaccuracies in the base score of CWE.

We propose a novel algorithm that leads to more accurate way to score the CWEs by mapping to CVE using context similarity. We use the available CVE scores [4] to generate a base score for each CWE. We discover similar CVEs and CWEs based on their respective description using NLP techniques, and using this similarity, we can generate the CWE base scores, and hence, an overall software security score. Software developers and managers can observe the continuous improvement of the software based on the changes of the scores of consecutive phases of fixing, testing, and debugging. In addition, the score generated for each CWE can be used to generate a ranking of weaknesses in the system, and determine which ones to address first, making a positive impact on programmers and managers. The final result of the entire process is a scored list of the actual vulnerabilities that exist in the software system, and its overall security score. In summary, the contributions of this paper are:

- Generate a more accurate base score for software weaknesses (CWEs) using context similarity of CWE and CVE.
- Calculate the security score of a software system using the generated CWE base score in a fully automated way.
- Evaluate the accuracy of the scoring mechanism through experimentation.

2. Related work

We categorize the related work in two topics: improvement of vulnerability scores, and calculating the overall security of a system. Most of these works use CVE as the basis for scoring. Very few works have considered CWE for measuring software security.

2.1. Vulnerability score improvement

A lot of work has been performed to predict and acquire a score for vulnerabilities (CVEs). In [7], the authors use logistic regression to detect the probability of having a vulnerability and binomial distribution to detect the severity of vulnerabilities, on vulnerability datasets of Adobe flash player and Firefox. The authors used mean time to vulnerability (temporal factor), local risk rate, mean risk rate, and overall risk value as important factors to predict the severity of a vulnerability. In [8], the authors used attack surface entity point connection with vulnerability location to predict vulnerability exploitation potential. All of these works used the CVSS (Common Vulnerability Scoring System) [5] score. In [9], temporal and environmental metrics were used to obtain an improved CVSS (Common Vulnerability Scoring System) score. In [10], the authors used temporal factor with Pareto and Weibull distribution [11] on discover, disclosure, exploit and patch date to add a temporal metric. Subsequently, [12] and [13] were inspired from [10] for temporal metrics. In [14], the authors also used the temporal factor to calculate software integrity level. These temporal factors consider the time passed since the discovery of that vulnerability. The temporal factor is adjusted as newer versions of CVE come in, sometimes the base scores change, some other times a resolved CVE is even removed from the list. So, for the time being, we can ignore the temporal metrics. In [13], the authors used software context information to prioritize CVSS based vulnerabilities, where the context information works as equivalent to temporal and environmental metrics. Our goal is to make a better use of context information by incorporating it with CWE to generate a better CWE score. Recently, information extraction techniques are being used to predict vulnerability score from the vulnerability description. In [15], the authors used word embedding (using skip-gram [16]) and one-layer shallow Convolutional Neural Network (CNN) [17] to predict vulnerability severity using the CVE database. They used probabilistic distribution to predict the severity level of the vulnerabilities. In [18], the authors also used word embedding (using word2vec [19]) with CNN to create sentence embedding of CVE, CWE and Common Attack Pattern Enumeration and Classification (CAPEC) [20], and knowledge graph to predict security entity relationship. The vectors generated by word embedding do not provide the entire context information. Hence, there is room for mistakes, therefore, the semantic process does not produce the correct result. To resolve this, we are using sentence embedding. It does a better, due to comparison between sentences, because it considers the context of sentence. In [21], the authors used PageRank [22] algorithm on CVSS to calculate the CWE score. They tried to improve it by using the child-parent relationship between different CWE. This can be useful to identify the possibility of a weakness being present in the system. But as we certainly know which CWEs exist in the system from our prior work [23], we can directly calculate the CWE score without considering its possible children.

2.2. Security score

As the vulnerability score gained from CVSS only provides information about individual CVEs, and not the total security posture of the software system, we need an approach to calculate the software security score accurately. There are only few existing works that concentrate on calculating the security score of the entire software. In [24], the authors proposed a method to assess the potential risk of cyber-attacks utilizing network-wide compliance reports. They consider vulnerability distribution, dependency between the vulnerabilities, and network configuration. But they do not consider the important impact factor. In [25], the authors propose that security metrics are proportional to the weighted total of base score and the weight of the vulnerabilities. In [26], the authors used aspect-oriented Stochastic Petri nets for threat modeling, where the authors followed the security metrics calculation proposed in [25]. They consider threat categorization of STRIDE for creating a way for threat mitigation. CVSS is used in [26], [24], [25] as the basis of vulnerability scoring. Our goal is to improve the security score compared to the above methods by using CWE along with CVE for scoring, and by using environmental metrics.

3. Approach

In order to calculate the security score of a software system, we first need to identify the number of weaknesses (CWEs) that exist in the system, and the base scores of those CWEs. Based on our previous work [23], we obtain a list of existing CWEs in a system by scanning the software with a scanning tool and we identify the true weaknesses using our machine learning methodology. Using the CWE base score, we derive the overall security score of a software system. First, we create a baseline method to calculate the software security score based on one of the existing and most commonly used methods [2]; then, we create and implement our own method and compare the results of both methods.

3.1. Baseline

For the baseline, we generate the CWE base score by mapping the CWE ID with its corresponding CVE ID, using the CVE list provided by MITRE [4]. We have followed the scoring suggestions from MITRE [2], which are:

Base Score for a CWE: Every CVE can be mapped to a CWE (multiple CVEs can be mapped with the same CWE), but not every CWE can be mapped to a CVE. Therefore, some CWE ($\mathrm{CWE_X}$) can be mapped with one or more CVEs, and some CWE cannot be mapped with any CVE. The CVE list provided by MITRE [4] includes the following information for each CVE: a CWE ID, a CVSS base score, and impact and exploitability sub-scores. For each CWE, we take the average of the CVSS base scores of its mapped CVEs. Then we normalize this average score by dividing it by the range of CVSS base scores (difference of the maximum and minimum CVSS base scores among all the mapped CVEs) for that particular CWE. The result is the base score *Sc(CWE_X)* of a CWE.

$$Sc(CWE_X) = \frac{avg(CVSS) - min(CVSS)}{max(CVSS) - min(CVSS)} \quad (1)$$

Equation 1 is the popularly used method of calculating CWE base scores, and it is presented in [2]. As several CWEs can be mapped to a CVE, the unmapped CWEs were ignored in [2]. In order to consider unmapped CWEs we need to assign a base score for each one of them. One possible solution is to use the severity level generated by the scanning tools. When the source code is passed through the scanning tools, it generates a list of possible CWEs in the system, along with their severity level. The different severity levels are - informational, low, medium, high, and critical. We map these severity levels to numeric values in the range of possible CWE scores (0 to 10). Severity levels that are informational, low, medium, high, and critical are mapped to 1,3,5,7 and 9 respectively.

Weight of a CWE: We also calculate the weight *(wt(CWE_X))* for each CWE representing the percentage of times a CWE appears within the software system.

$$wt(CWE_X) = \frac{\#\ of\ times\ CWE_X\ occurs}{total\ \#\ of\ weaknesses} \quad (2)$$

In [2], weight is represented as frequency, and it is calculated as the number of CVEs that can be mapped to a CWE within the National Vulnerabilities Database (NVD) [27]. The NVD is the U.S. government repository of vulnerability management data. This mapping with the entire database of CVE was performed to rank the entire CWE list. As every software system has a small number of repeated CWEs, we do not need to consider the entire list of CWE to generate that system's security score. So, we modified the weight calculation as follows: instead of counting the number of times a CWE maps with any CVE in the entire NVD database, we count the number of times a particular CWE appears in the system.

Weakness Score of a CWE: After generating the CWSS base score for each CWE in the system, we take their weighted average to determine the security score [25] [26]. To get the base score of each CWE, we multiply their mapped base score with their weight. The sum of the base scores of all existing CWE in a system is the weakness score of a system.

$$Weakness = \sum(wt(CWE_X) \times Sc(CWE_X)) \quad (3)$$

Security Score: In this method, the CWE base score is generated from CVE base scores. As the CWE base score is the weighted average of base scores of the mapped CVEs, the maximum value of CWE base scores must also be 10. Since the highest value of base scores is 10, and the weights are less than 1 (weight in this case is defined as the percentage of times a CWE appears in a system), this score identifies how weak the overall system is, and ranges from 1-10. We multiply the base score by 10 to generate a more conventional score out of 100. Therefore, the security score of the system, based on the weakness score as follows:

$$Security\ score = 100 - Weakness \quad (4)$$

In the baseline, we map a CVE to a CWE, to get the CWE base score. Though this method is widely used, it does not give us an accurate CWE score, due to the fact that many CWEs map to the same CVE. Therefore, these CWEs will have similar base score. In real life, CWEs that map to the same CVE sometimes may not operate in a similar way, and hence may not cause similar security breach. This difference leads to an error in the base score calculation. In order to have a more precise base score, we make use of the contextual information to map the CWEs to CVEs; consequently, we minimize this error. Also, considering the severity level as the base scores for unmapped CWEs generates an error in the security score, because the severity level is generic, and can vary from one scanning tool to another. However, if we use context similarity, we can generate a more accurate score for any non-mapped CWEs. Therefore, for a more accurate estimation of the security posture of the system, we propose to create a general method to generate CWE base score using context similarity, impact, and exploitability sub-scores for each weakness.

3.2. Proposed Approach

Figure 1 depicts an overview of the proposed scoring system. We start by providing the source code to the static analysis scanning tools, which scan the code and generate a list of potential CWEs. Based on our prior research, we observe that many of these potential CWEs are in fact false positives. So, we designed and implemented the VINCI (Vulnerability IdeNtifiCatIon) tool, which identifies and labels the false positive vulnerabilities using the output report of the scanning tools [23]. VINCI uses author information along with other criteria to accurately identify false positive CWEs, allowing only true positive CWEs to be considered for security scoring. Consequently, using VINCI, we get an actual list of weaknesses (CWE) in a system. Then, we calculate an accurate base score for the CWEs, and using this base score, generate the software security score.

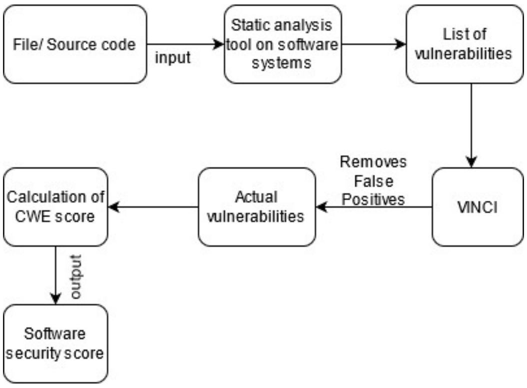

Figure 1: Overview of the proposed system for measuring the security score of a software application

Measuring similarity between a CWE and a CVE using NLP: At this point we need to identify how similar a given CWE is to a CVE. Hence, we need to perform a contextual comparison between the descriptions of CWE and CVE to find out which vulnerability (CVE) is contextually similar to a weakness (CWE) by using their descriptions. This is not only important but also critical since we can get a better idea whether a CVE and a CWE work in the similar way and cause security breach in the same way, or not. To do this, we create a universal vector of all the CVE and CWE descriptions using sentence embedding. Sentence embedding is an NLP process that embeds each sentence into an n-dimensional vector space, by inheriting the features from underlying word embeddings. So, each sentence vector is created based on the words in that sentence, and the context of those words. After representing two sentences as vectors (for example sentences like 'I drive car' and 'I can drive'), we can calculate their cosine similarity.

$$similarity(A, B) = cos\theta = \frac{(A.B)}{|A||B|} \quad (5)$$

In equation 5, A and B represent the sentence vectors of the two sentences. If similarity(A,B) increases, it means that the angular difference between these two sentences decreases, meaning the sentences are similar. Using this process, we perform a semantic search between a CWE and a CVE. Semantic search is the task of finding similar sentences from a corpus to a given sentence, by using the context of the sentence. Therefore, we conduct a semantic search to find the closest CVEs to any given CWE based on similarity, which is represented by the sentence vector. This is represented in lines 1 to 3 in algorithm 1. We use Python NLP libraries to create sentence vectors, such as the *sentence-transformers* [28] library. As there is not enough CVE/CWE description data to train the model, and the vulnerability/weakness descriptions are well formatted, we decided to use a pre-trained model for sentence embedding. This model is trained on well formatted text, which means it is good at creating a vector for grammatically and syntactically correct texts, and it works well for our purpose. We have manually checked each CWE and its similar CVEs to confirm the accuracy of the model.

Using the generated sentence vectors for CVE and CWE description in the universal vector space, we calculate how much two sentences differ from each other. By subtracting their vectors using distance modules from *Scipy* [29], we get a score that represents how different these two sentences are, i.e. we obtain the distance between them. This distance score is always less than or equal to 1, (1 signifies that the sentences are completely different, and 0 signifies that both sentences are the same). By subtracting the distance score from 1, we get the similarity score between two sentences. Higher similarity score means that two sentences are more similar. When we compare the sentence vector of a CWE against the universal sentence vector of the CVEs, this similarity score helps us find the most similar CVEs for that CWE. This is equivalent to the probability of a CWE being related to a particular CVE. For each weakness in CWE list, we consider the 5 most similar CVEs. The CVE list [4] comes with base score of a CVE, along with its impact and exploitability sub-score. For each CWE we take the weighted average of the impact and exploitability subscore of the 5 most similar vulnerabilities, where weight is the similarity score between the CWE and the CVE. The impact score is represented as,

$$impact = \frac{\sum_{i=1}^{5}(impact_i \times similarity_i)}{5} \tag{6}$$

And the exploitability score is represented as,

$$exploitability = \frac{\sum_{i=1}^{5}(exploit_i \times similarity_i)}{5} \tag{7}$$

Therefore, the base score is given by Equation 8 below.

$$base\_score = impact + exploitability \tag{8}$$

The entire process to calculate the base score is shown in Algorithm 1.

Algorithm 1

Input: the list of the CWE and CVE descriptions
Output: CWE base scores

```
1: Create universal sentence vectors (vect_x) for all CWE
   and CVE descriptions using sentence embeddings
2: for all  CWE do
3:     Find the most similar CVEs by comparing the sentence vectors.
           $similarity(x, y)\frac{(vect_x.vect_y)}{|vect_x||vect_y|}$
4:     Calculate the average impact and exploitability
       sub-scores as in equations 6 and 7
5:     Calculate CWE base score using impact and exploitability sub-scores as in equation 8
6: end for
```

An advantage of this process is that, unlike the baseline, every CWE gets scored, depending on its similarity with a CVE. Even if a CWE does not properly map to any CVEs, it gets assigned an appropriate score. The sub-scores of the mapped CVEs are multiplied by the similarity score, which give us the weighted sub-scores that help us generate accurate base score for a CWE with respect to the CVE. We can also generate CWE base scores directly from CVE base scores (without generating impact and exploitability sub-scores separately, and then adding them) using the same method. But impact and exploitability of two CWEs can sometimes vary largely, even if they have the same base score. So, it is better to derive impact and exploitability score first, and then add them to generate the base score.

Finally, when we have all the base weakness scores for all CWEs, we can calculate the total weakness score of the system by taking the average of their base scores using the following equation:

$$weakness\ score = \frac{\sum(base\ score_i)}{total\ number\ of\ CWEs} \tag{9}$$

where, *n* represents the total number of CWEs in a system.

Like the baseline, this weakness score always ranges from 1 to 10. We multiply the resulting score by 10 to get a more conventional score out of 100. It is more desirable for a higher score to reflect a more secure system, Therefore, the security score of the system, based on the weakness score is as follows:

$$security\ score = 100 - 10 \times weakness\ score \tag{10}$$

This calculation process is represented in lines 4 to 5 in Algorithm 1.

4. Experimental Evaluation

In order to evaluate our technique, compare it against the most commonly used ones, and to measure the improvement, we performed several experiments and recorded the results using various actual and synthetic datasets.

4.1. Datasets

We applied our method on two real-world open source software systems (WhatsApp [30] and Atom [31]) and a synthetic dataset (SARD [32]). After running static analysis tools on these software systems, we obtain a list of possible weaknesses. However, this list contains false positives, which are removed by running the VINCI tool [23]. The result contains only actual CWEs that exist in the system (true positives). WhatsApp had 490 actual CWEs, Atom had 160 CWEs and SARD had 30 CWEs. In every system, one CWE might be repeated multiple times. There are also a lot of CWEs that cannot be directly mapped with any CVE, especially in the case of SARD and Atom. The unmapped CWEs add significant effect on the security score. The base scores of CWEs are calculated as the methods discussed in section 3.2.

4.2. Experiments and Results

In our experiments we derive the security posture of a software system using the security score described in our approach. We used a 64 bit Dell laptop with Windows 10 Pro operating system, Intel(R) Core(TM) i7- 1065G7 CPU (2 GHz) with 8GB RAM and 500 GB hard disk. Figure 2 shows a comparison of the final security score baseline method versus two versions of our approach. The first version calculates weakness scores acquired directly by averaging the base score of similar CVEs, without calculating the impact and exploitability sub-scores separately. The second version calculates the impact and exploitability sub-scores separately, and then adds them to obtain the base score for CWE. Different methods of calculating security score for different systems are represented in the X-axis, and the security score for each case is represented in the Y-axis. The security score ranges from 1 to 100, where higher scores identify better security.

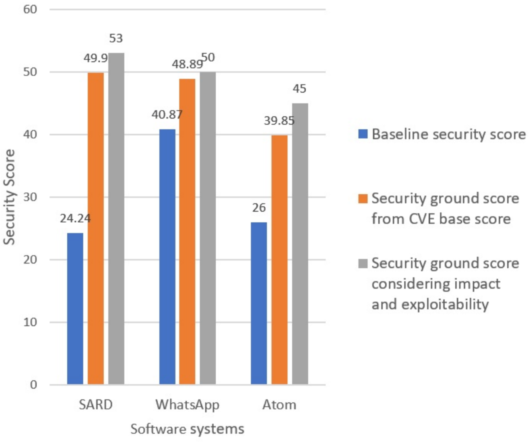

Figure 2: Security score using different methods

We observe a large difference of security score between the baseline and our proposed methods. The final security score generated by our proposed method is significantly higher than the baseline score. This is due to the fact that each CWE in the baseline is mapped to one or more CVEs, and sometimes there is no equivalent CVE for a given CWE. For the non-mapped CWEs, the baseline considers the severity level as the base score, which is often not accurate. This can sometimes increase and sometimes decrease the actual security score, injecting more inaccuracy in the final score. Also, if the number of non-mapped CWEs is very large, it can greatly affect the actual security score. The more non-mapped CWE a system contains, the more its final security score varies from the baseline score. So, even though the baseline approach is the most commonly used method, it provides inaccurate results for most of the cases, especially in the SARD dataset. As it is synthetic, it may not reflect real word scenarios, even lesser number of distinct CWEs. Among our datasets, SARD has the least number of unique CWEs, and a higher number of CWEs that are non-mapped to a CVE. Therefore, the baseline security score for SARD is very low, resulting in a large difference from the security score generated using NLP. We also experimented on Atom version 1.33.0, which contains a large number of unmapped CWEs. We observe that there is a big difference between the baseline score and the final security score for Atom too, with a low baseline score and much higher security score. The base scores achieved by our process utilize context and are available for all CWEs. For the two methods that we implemented, we observe that, the one calculating impact and exploitability sub-score gives slightly better result than directly calculating the base score. This happens because even if two CVEs have the same base score, their impact on the system can be different, and their exploitability can vary too. So, the CWE base score can be different sometimes if we calculate impact and exploitability sub-scores of CVE first and add them later, as opposed to calculating it directly from CVE base scores. The difference of these scores occurs for some CWEs, but in some other cases, impact and exploitability sub-scores do not vary much when they have the same base score. So, the overall security score, which consists of all CWEs, might not always be very different for these two methods.

5. Conclusion

It is very important to estimate how secure a software is, in order to determine how much effort and attention should be allocated in terms of having a more secure software system. The more CWEs a system has, and the more severe those CWEs are, the more vulnerable the system is to attacks. Using the CWEs that exist in a system, we can generate a score that identifies the security posture of a software system, therefore providing idea about the vulnerability of the software system against cyber attacks.

In this paper, we have designed and implemented a method to provide a more accurate software security score based on the existing CWE information. This is accomplished by scoring each CWE in the software as accurately as possible, utilizing contextual information. We have used semantics and NLP to map CWEs to CVEs, in order to generate a base score for CWE. In order to guarantee the correctness of the semantic mapping process, we manually checked the correctness of sentence similarity. In the future, we plan to reduce the need of manual checking by generating and collecting sufficient data for supervised learning, which will make the process even more automated.

In addition, we have evaluated our methods through experimentations, where we have compared the results with the most generally used baseline, which is currently the state-of-art [2]. We have tested our proposed method and the baseline method on different real world and synthetic software systems. In every case, our proposed method has produced significantly more accurate and logical scores compared to the popularly used baseline method.

The resulting procedure can help system managers and developers get an accurate estimation about the security posture of the system, leading towards more secure software (which are less prone to cyber-attacks), and therefore increasing the quality of software. This method for CWE scoring can also be used to prioritize weaknesses that exist in a system, and determine which ones to address first. In a whole, our method can reduce the workload relevant to ensuring software security, and help the organizations create more secure software systems using less time and resources.

Acknowledgments

This research has been partially supported by the State of Maryland through TEDCO Maryland Innovation Initiative (MII) grant # 0719-003.

This research has been partially funded by a grant from the the Office of Innovation and Entrepreneurship of the US Economic Development Administration by the Bureau of US Department of Commerce.

References

[1] MITRE, Common weakness enumeration (cwe), https://cwe.mitre.org/, 2006. Accessed: 2021-06-28.

[2] MITRE, 2019 cwe top 25 most dangerous software errors, 1.https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html, 2019. Accessed: 2021-05-17.

[3] T. M. Corporation, The mitre corporation, https://www.mitre.org/, 1997. Accessed: 2021-08-12.

[4] NVD, Common vulnerabilities and exposures (cve), https://cve.mitre.org/, 1999. Accessed: 2021-06-28.

[5] NVD, Common vulnerability scoring system, https://www.first.org/cvss/, 1999. Accessed: 2021-06-29.

[6] MITRE, Common weakness scoring system (cwss), https://cwe.mitre.org/cwss/cwss_v1.0.1.html, 2006. Accessed: 2020-07-07.

[7] X. Zhu, C. Cao, J. Zhang, Vulnerability severity prediction and risk metric modeling for software, Applied Intelligence 47 (2017) 828–836.

[8] A. A. Younis, Y. K. Malaiya, Using software structure to predict vulnerability exploitation potential, in: 2014 IEEE Eighth International Conference on Software Security and Reliability-Companion, IEEE, 2014, pp. 13–18.

[9] J. A. Wang, F. Zhang, M. Xia, Temporal metrics for software vulnerabilities, in: Proceedings of the 4th annual workshop on Cyber security and information intelligence research: developing strategies to meet the cyber security and information intelligence challenges ahead, 2008, pp. 1–3.

[10] S. Frei, M. May, U. Fiedler, B. Plattner, Large-scale vulnerability analysis, in: Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense, 2006, pp. 131–138.

[11] A. Alzaatreh, F. Famoye, C. Lee, Weibull-pareto distribution and its applications, Communications in Statistics-Theory and Methods 42 (2013) 1673–1691.

[12] R. Wang, L. Gao, Q. Sun, D. Sun, An improved cvss-based vulnerability scoring mechanism, in: 2011 Third International Conference on Multimedia Information Networking and Security, IEEE, 2011, pp. 352–355.

[13] C. Fruhwirth, T. Mannisto, Improving cvss-based vulnerability prioritization and response with context information, in: 2009 3rd International Symposium on Empirical Software Engineering and Measurement, IEEE, 2009, pp. 535–544.

[14] T. Fujiwara, J. M. Estevez, Y. Satoh, S. Yamada, A calculation method for software safety integrity level, in: Proceedings of the 1st Workshop on Critical Au-

tomotive applications: Robustness & Safety, 2010, pp. 31–34.

[15] Z. Han, X. Li, Z. Xing, H. Liu, Z. Feng, Learning to predict severity of software vulnerability using only vulnerability description, in: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2017, pp. 125–136.

[16] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, in: Advances in neural information processing systems, 2013, pp. 3111–3119.

[17] K. O'Shea, R. Nash, An introduction to convolutional neural networks, arXiv preprint arXiv:1511.08458 (2015).

[18] H. Xiao, Z. Xing, X. Li, H. Guo, Embedding and predicting software security entity relationships: A knowledge graph based approach, in: International Conference on Neural Information Processing, Springer, 2019, pp. 50–63.

[19] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, arXiv preprint arXiv:1301.3781 (2013).

[20] MITRE, Capec, https://capec.mitre.org/, 2007. Accessed: 2021-08-10.

[21] Y. Du, Y. Lu, A weakness relevance evaluation method based on pagerank, in: 2019 IEEE Fourth International Conference on Data Science in Cyberspace (DSC), IEEE, 2019, pp. 422–427.

[22] A. N. Langville, C. D. Meyer, Google's PageRank and beyond, Princeton university press, 2011.

[23] F. Cheirdari, G. Karabatis, Analyzing false positive source code vulnerabilities using static analysis tools, in: 2018 IEEE International Conference on Big Data (Big Data), IEEE, 2018, pp. 4782–4788.

[24] M. N. Alsaleh, E. Al-Shaer, Enterprise risk assessment based on compliance reports and vulnerability scoring systems, in: Proceedings of the 2014 Workshop on Cyber Security Analytics, Intelligence and Automation, 2014, pp. 25–28.

[25] J. A. Wang, H. Wang, M. Guo, M. Xia, Security metrics for software systems, in: Proceedings of the 47th Annual Southeast Regional Conference, 2009, pp. 1–6.

[26] N. H. Sherief, A. A. Abdel-Hamid, K. M. Mahar, Threat-driven modeling framework for secure software using aspect-oriented stochastic petri nets, in: 2010 The 7th International Conference on Informatics and Systems (INFOS), IEEE, 2010, pp. 1–8.

[27] NIST, National vulnerability database, https://nvd.nist.gov/vuln, 1999. Accessed: 2021-06-21.

[28] N. Reimers, I. Gurevych, Sentence-bert: Sentence embeddings using siamese bert-networks, in: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, 2019. URL: http://arxiv.org/abs/1908.10084.

[29] E. Jones, T. Oliphant, P. Peterson, et al., Scipy: Open source scientific tools for python (2001).

[30] WhatsApp Inc. (Facebook, Inc.), Whatsapp, 2009. URL: https://whatsapp.com.

[31] MIT, Atom, https://atom.io/, 2014. Accessed: 2021-07-16.

[32] P. Black, Sard: Thousands of reference programs for software assurance (2017). URL: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=923127.