MultiVD: A Transformer-based Multitask Approach for Software Vulnerability Detection

Claudio Curto[1][a], Daniela Giordano[1], Simone Palazzo[1] and Daniel Gustav Indelicato[2]
[1]*Dipartimento di Ingegneria Elettrica, Elettronica e Informatica (DIEEI), Università degli Studi di Catania, Catania, Italia*
[2]*EtnaHitech Scpa, Catania, Italia*

Keywords: Vulnerability Detection, Machine Learning, Deep Learning, Transformer.

Abstract: Research in software vulnerability detection has grown exponentially and a great number of vulnerability detection systems have been proposed. Recently, researchers have started considering machine learning and deep learning-based approaches. Various techniques, models and approaches with state of the art performance have been proposed for vulnerability detection, with some of these performing line-level localization of the vulnerabilities in the source code. However, the majority of these approaches suffers from several limitations, caused mainly by the use of synthetic data and by the inability to categorize the vulnerabilities detected. Our study propose a method to overcome these limitations, exploring the effects of different transformer-based approaches to extend the models capabilities while enhancing the vulnerability detection performance. Finally, we propose a transformer-based multitask model trained on real world data for highly reliable results in vulnerability detection, CWE categorization and line-level detection.

1 INTRODUCTION

In the last decade, research about the application of machine learning technologies in vulnerability detection in information systems has grown significantly, revolutionizing the way cybersecurity experts identify and mitigate potential threats. Leveraging the most advanced machine learning algorithm, researchers proposed various tools, methodologies and techniques (Rahman and Izurieta, 2022) to help developers to avoid and correct vulnerabilities in their software that could compromise the integrity of the software itself. One of these methodologies is the Static Application Security Testing (SAST), or static analysis. It is a testing methodology that analyzes source code to find security vulnerabilities that make an application susceptible to attack. SAST scans the application before the code is compiled and it is also known as white box testing. Given the sequential nature of programming languages, Natural Language Processing (NLP) approaches have achieved great results in source code analysis (Singh et al., 2022), leveraging LSTM (Long Short-Term Memory) architectures (Fang et al., 2018), GNN (Graph Neural Network) (Zhou et al., 2019), (Li et al., 2021) and, most recently, transformers architectures (Fu and Tantithamthavorn, 2022), (Mamede et al., 2022), (Hin et al., 2022). Specifically, transformer-based approaches emerged as state-of-the-art in vulnerability prediction tasks, both on function-level and line-level vulnerability prediction (Fu and Tantithamthavorn, 2022), (Hin et al., 2022). However, there are various open challenges in this field, as the classification of the predicted vulnerability and the model's generalization capability over different projects (Kalouptsoglou et al., 2023), (Chen et al., 2023).

To date, multiclass vulnerability classification has not been fully explored, due to the lack of adequately varied and quality datasets. In 2021 Zou et al. (Zou et al., 2021) proposed the first multiclass vulnerability classifier, namely *μVulDeePecker*. However, their system suffers of two limitations: it is unable to locate the vulnerability in the code and it is trained on a synthetic dataset. The datasets nature is a crucial aspect to consider, as a model trained on a synthetic dataset will be limited to detecting only the simple patterns present in the data, which seldom occur in real life (Chakraborty et al., 2022). In contrast, real-world datasets are derived from real-world sources and generally only contain functions that went through vulnerability-fix commits. In this case the model will be able to learn real and more

[a] https://orcid.org/0009-0006-6516-7671

Curto, C., Giordano, D., Palazzo, S. and Indelicato, D.
MultiVD: A Transformer-based Multitask Approach for Software Vulnerability Detection.
DOI: 10.5220/0012719400003767
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 21st International Conference on Security and Cryptography (SECRYPT 2024)*, pages 416-423
ISBN: 978-989-758-709-2; ISSN: 2184-7711
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

complex patterns.

In this study we explore two possible classification approaches: multiclass and multitask classification. We do this first by fine tuning the CodeBERT(Feng et al., 2020) architecture to perform a multiclass classification of the known vulnerabilities, to explore the model vulnerability pattern recognition capabilities; second, by improving the original approach with the addition of two classification heads to the model, performing simultaneously a binary classification task for vulnerability detection and a multiclass classification task for vulnerability categorization. The idea is to leverage the multitask approach to provide a more informative result from the model, while, at the same time, enhance the model performance both on the detection and classification fields. Finally, we compare the attention-based line-level detection performance between LineVul and our multitask model to show the effects of the new approach to the original task. In the last step of our study, we explore if it is possible to improve the final results of the multitask model by a different handling of the loss functions.

2 BACKGROUND AND RELATED WORKS

2.1 Software Vulnerability Assessment

A software security test consists in a validation process of a system or application with respect to certain criteria. There are several approaches to test the vulnerability of a software; these approaches can be categorized in three groups:

- Static analysis: the vulnerability assessment is performed directly on the application's source code, usually before its deployment. This form of testing can be performed using pattern recognition techniques or, in a machine learning scenario, by NLP-based techniques (LSTM, GNN, transformers).
- Dynamic analysis: the assessment is performed on the running application, monitoring its behaviour. Generally called "penetration testing", the focus of the test is to validate the application behaviour following some specific inputs given by the user.
- Hybrid analysis: a hybrid approach takes advantage of strengths and limitations of static and dynamic approaches.

Ghaffarian et al. (Ghaffarian and Shahriari, 2017) identified four classes of methodologies applied in vulnerability assessment:

- Anomaly detection approaches
- Vulnerable code pattern recognition
- Software metrics based approaches
- Miscellaneous approaches

Vulnerable code pattern recognition is the one explored in this study.

2.2 Common Weakness Enumerations (CWE)

Every year the number of possible vulnerabilities that could affect an information system grows exponentially. It has become necessary to keep track of the new vulnerabilities discovered day after day, to make it possible for developers to prevent exposures to their systems. MITRE plays a prominent role in developing and maintaining numerous resources and frameworks to improve security practices. MITRE Common Weakness Enumeration (CWE) is a community-developed list of common software security weaknesses. It represents a baseline for weaknesses identification and prevention, providing a hierarchical taxonomy of software weaknesses that can lead to one or more security vulnerabilities. According with the most recent MITRE CWE List (MITRE, 2023), the total number of registered Weakness is 934.

2.3 BERT and BERT-Based Architectures

Bidirectional Encoder Representation from Transformers (BERT) has been introduced by Devlin et al. in 2018 (Devlin et al., 2019) and has achieved remarkable results across a wide range of NLP task (e.g., question answering, sentiment analysis, text classification and more). Different variants of BERT have been released, including models for specific domains or models with improved performances like CodeBERT (Feng et al., 2020) to handle programming languages, understanding source-code data and programming related tasks. It is pre-trained with a 20 GB source code corpus with bimodal instances of NL-PL (Natural Language - Programming Language) pairs, unimodal codes (not paired with natural language texts) and natural language texts without paired codes (2.1M bimodal datapoints and 6.4M unimodal codes across the programming languages Python, Java, JavaScript, PHP, Ruby, Go). CodeBERT has been pre-trained on two tasks: Masked Language Modelling (MLM) and Replaced Token Detection (RTD).

2.4 Models for Vulnerability Detection

Zou et al. (Zou et al., 2021) presented *μVulDeePecker*, the first deep learning-based system for multiclass vulnerability detection. The underlying system architecture is constructed from Bidirectional Long-Short Time Memory (BLSTM) networks and aims to fuse different kinds of features from pieces of code (called code gadgets) and code attention to accommodate different kinds of information. For this purpose, the authors created from scratch a dataset and used it to evaluate the effectiveness of the model. The dataset contains 181,641 code gadgets, with 43,119 vulnerable units and 40 types of vulnerabilities in total. The model achieved high performance both when tested on their own test set (94.22% F1) and on real world software (94.69% F1). However, *μVulDeePecker* suffers of some limitation: first, it can detect vulnerability types, but cannot pin down the precise location of a vulnerability in the code; second, the current design and implementation focus on vulnerabilities that are related to library/API function calls.

Fu et al. (Fu and Tantithamthavorn, 2022) introduced LineVul, a transformer-based vulnerability prediction system, achieving state-of-the-art performance. LineVul addresses the limitations of the graph-based neural networks when used in this task (Li et al., 2021) and demonstrates the potentiality of attention in vulnerability prediction at line-level. The result is a line-level vulnerability prediction system with high performance both on function-level detection and line-level detection tasks. LineVul strengths can be found in its architecture. The model can be seen as a composition of three component: CodeBERT (Feng et al., 2020), Byte Pair Encoding (BPE) Tokenizer and a single linear layer classifier. BPE is a data compression algorithm, very popular in NLP tasks for its high efficiency in building small vocabularies for text tokenization. The strength of the algorithm is its ability to handle rare or out-of-vocabulary words, using a subword tokenization approach and leading to a better generalization and coverage, especially in cases where the training data may have limited vocabulary coverage. As demonstrated in the paper, the combined use of the pretrained CodeBERT model and Byte Pair Encoding is the key for the high performances of the model: there is a 50% F1-score reduction when using a word-level tokenization and a 11% reduction when using non-pretrained weights to initialize BERT.

Mamede et al. (Mamede et al., 2022) explored different BERT-based models' performances (CodeBERT (Feng et al., 2020) and JavaBERT (De Sousa and Hasselbring, 2021)) for multi-label classification of Java vulnerabilities, training them on the Juliet synthetic dataset[1]. The studied models showed high performance when tested on synthetic data and good generalizability when tested on unknown vulnerabilities, related with the kind of vulnerabilities which the models have been trained on, leveraging their belonging SFP (Software Fault Patterns) secondary clusters. However, it is pointed out that using only synthetic data is insufficient, since the models' performance severely degrade when facing real-world scenarios. Indeed, all the studied models showed a great loss in performance when tested with real-world data, with a 50% and 58% reduction in F1-score and recall respectively for JavaBERT, indicating that the model can recognise vulnerable patterns but stumble in selecting the type of vulnerability (CWE). Generally, it has been observed that all the model suffer of an high false negative rate when tested in realistic contexts.

3 METHODOLOGY

3.1 Classification

The adopted model architecture extends the structure defined by Fu et al.(Fu and Tantithamthavorn, 2022), consisting of three components: BPE tokenizer, CodeBERT and the linear classifier. The linear classifier has been updated for the multiclass and the multitask implementation. For the multiclass approach it consists of a single linear layer with 15 output neurons. For the multitask approach the classifier is built with two classification heads, one for the multiclass classification with 15 output neurons (one for each CWE to classify) and the other for the binary classification (vulnerability target). For both tasks, we adopt Cross-entropy as loss function. When training the multitask model, a weighted loss is computed as

$$loss_W = \alpha * loss_M + \beta * loss_B$$

where $loss_M$ is the multiclass loss, $loss_B$ is the binary loss, α and β are two arbitrary parameters that assume values between 0 and 1, with 0 excluded.

3.2 Localization

For line-level vulnerability localization we use the approach by Fu et al. (Fu and Tantithamthavorn, 2022), leveraging the attention scores assigned to the tokens by the model. The idea of this approach is that high attention tokens are likely to be vulnerable tokens, so

[1]https://samate.nist.gov/SARD/test-suites/111

lines with higher attention score are the ones with higher probability to contain the vulnerability. This is done by obtaining the self-attention scores from the trained model for every sub-word token and then integrate those scores into line scores. Specifically, a whole function is split to in lists of tokens where every list represent a line (the split is done by the newline control character \n). Every token in a list will have an associated cumulative attention score, so, for each list of token scores, we summarize it into one attention line score and rank all the line scores. The ranking of the lines based on the relative attention scores will show the lines with higher probabilities to be the location of the vulnerability. This approach is applied considering the true positive outputs of the vulnerability detection task, so in our case we take into account binary predictions only.

3.3 Dataset

For a fair comparison with LineVul we use the same benchmark dataset, provided by Fan et al. (Fan et al., 2020), namely BigVul. BigVul is a large C/C++ vulnerability dataset, collected from open-source GitHub projects. BigVul dataset is the only vulnerability dataset that provides line-level ground-truth, necessary to our study to compare line-level prediction performance between the two models. All the functions in the dataset are assigned a CWE ID describing a vulnerability type, even the not vulnerable functions. This is explained analyzing the way the methods are gathered. The data collection procedure is structured in three steps.

1. First, Fan et al. perform a scraping procedure of the CVE database, collecting all the vulnerabilities information until the 2019.
2. From all the entries collected, are collected the ones with a github reference link pointing to a code repository to retrieve the vulnerable projects. For every project, the commit history is extracted.
3. Each retrieved commit from the history is considered as a mini version of its project, and every mini version is mapped to the relative CVE information retrieved in the first step. For each of the commits explicitly considered as relevant, they extracted the code changes that fixed the vulnerability; in this way, it was possible to build the vulnerable code. All the other not relevant commits were considered as not vulnerable.

So, different functions are gathered from the same reference link, reporting both vulnerable and not vulnerable samples, all sharing the same CVE information previously gathered. These functions, even if not vulnerable, may share useful information to recognize some hidden pattern typical of that CWE. From this perspective, we want to explore the results of a classification-based approach considering both vulnerable and not vulnerable samples for CWE classification.

To avoid possible inconsistencies between the data, we obtained the dataset from the LineVul GitHub repository[2]. The dataset in the repository is available both in its split version (train, validation and test split) and unsplit. We obtained the unsplit one, that came with a total of 4.841.688 samples. However all these samples can't be used as they present inconsistent entries e.g. None/NaN values, missing CVE labels or CWEs as pieces of string or code instead the ID. We performed a cleaning procedure removing all the samples where the CWE ID is not in the format "CWE-number", reducing their number to 146,625 functions, with 7,117 vulnerable functions. Another important aspect of the dataset is its high imbalance among all the different CWE IDs and between vulnerable and not vulnerable functions. This is strictly related to the fact that some vulnerabilities may occur less frequently than others. To get more reliable results, we restricted the multiclass classification to the top-15 CWE classes in terms of distribution in the dataset. The selected classes are reported in Figure 1. In the end, the final unbalanced dataset used is composed by 126,313 functions, with 7,089 vulnerable ones. Considering the high imbalance between vulnerable and not vulnerable samples, we test the model performance by training it with two variations of the dataset: unbalanced and balanced, the latter obtained by undersampling the not vulnerable occurrences. The resulting balanced dataset is composed by 14,178 functions equally distributed between vulnerable and not vulnerable. For all tests, the datasets are split into train, validation, and test sets with 80/10/10 ratio, with validation and test sets stratified to keep the same CWE ID class distribution as the train set.

3.4 Evaluation Metrics

Accuracy, precision, recall and F1-score are computed for training, validation, and testing phases. We take note of the best F1-score on the validation set to save the model checkpoint for testing. For the line-level detection, Fu et al. adopt two metrics: Top-10 accuracy and Initial False Alarm (IFA). Top-10 accuracy measures, after producing for each function a top 10 ranking of the lines based on their cumulative at-

[2]https://github.com/awsm-research/LineVul/tree/main/data

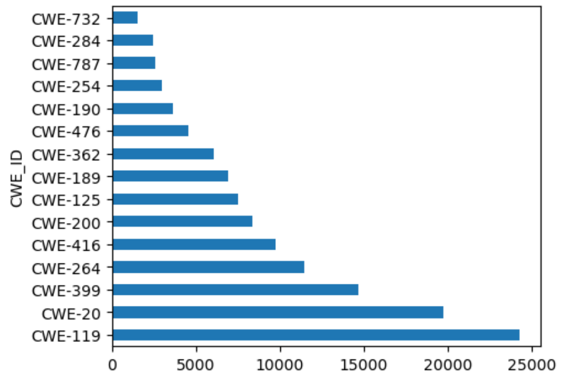


Figure 1: Top 15 CWEs in the dataset.

tention score, the percentage of these functions where at least one actual vulnerable line appear in the ranking. The idea behind this metric is that security analysts may ignore line-level recommendations if they do not appear in the top-10 ranking. Thus, a high top-10 accuracy value if preferred. It is computed by iterating all the flaw line indices and verifying if the current index is in the top-10 ranking. In this way we determine if it is correctly localized and count it accordingly. Initial False Alarm (IFA) measures the number of lines predicted incorrectly as vulnerable that security analysts need to inspect before finding the vulnerable one, for a given function: a low IFA value is preferred, as it indicates that security experts will spend less time inspecting false alarms raised by the system. Considering the top-10 line scores ranking previously computed, IFA values are obtained by looking the position of all the flaw lines in the ranking and then considering the minimum (we consider the first vulnerable line that appears in the ranking). The idea is that if a flaw line is, for example, in the 5th position in the ranking, the security analyst will have to inspect 5 clean lines before finding the flaw one.

4 EXPERIMENTS SETTING

For the model implementation, the pre-trained CodeBERT tokenizer and CodeBERT are downloaded from the HuggingFace repository. For the multiclass approach the linear classifier consists of a single linear layer with 15 output neurons; for the multitask approach the classifier is implemented to return a couple of outputs, one for multiclass classification and the other for binary classification. The training is performed on a NVIDIA RTX A6000 GPU. For the hyperparameter aspect, we leave the CodeBERT default settings unchanged and choose $2 * 10^{-5}$ for the learning rate, with AdamW as the optimizer. The model is trained for 10 epochs with cross-entropy as loss function for all the classifications. The multitask final loss is first computed as an average of the two, while in a second experiment we explore a different approach with a weighted loss.

Table 1: Multiclass classification results.

%	Accuracy	Precision	Recall	F1
Multiclass (only vulnerable)	53.31	55.68	56.43	55.39
Multiclass (balanced)	70.87	68.06	69.18	68.15
Multiclass (unbalanced)	79.61	77.92	81.81	79.67
Multitask (balanced)	71.86	69.85	72.37	70.79
Multitask (unbalanced)	79.03	77.39	81.40	79.19

5 RESEARCH QUESTIONS

During the study we examine the following research questions:

1. Which of the explored approaches has the best performance?
2. What are the effects of the multitask approach in vulnerability localization?
3. Does a weighted loss increase the model performances instead of an average loss?

RQ1. Which of the Explored Approaches Has the Best Performance?

During the training of the model, we take note of the test metrics for each task (multiclass, binary and multitask) and for two versions of the dataset, i.e. unbalanced and balanced, plus an additional test with only vulnerable sample in the case of the multiclass classification. Finally, we compare the results with the ones obtained with the multitask approach. These results are reported in Table 1 for all versions of the datasets. The results show how the multiclass-only classification model has the better performance when trained on all the available data, with ∼34% higher F1-score. In the same way, multitask classifier shows ∼10% higher performance, with a 79.19% F1 score in the unbalanced data scenario. From these results we may assume that the amount of available data plays a key role in the vulnerability classification task and that a greater amount of data is required to achieve better results.

For the binary task, we compare the multitask model with LineVul, retrained on both version of the dataset. The results are shown in table 2. It is noticeable, in the unbalanced scenario, that the proposed model preserves the LineVul performance, with the

Table 2: Comparison between LineVul and multitask binary classification.

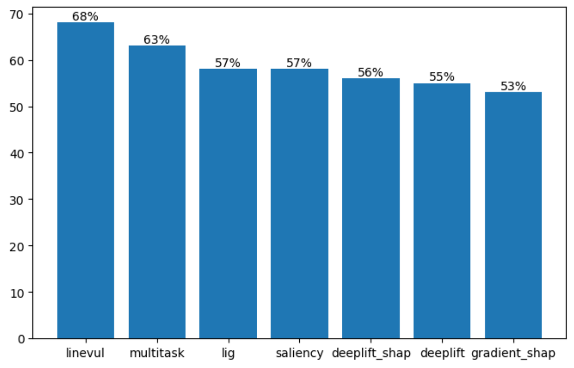
%	Accuracy	Precision	Recall	F1
LineVul (balanced)	97.81	99.39	96.01	97.67
Multitask (balanced)	97.88	97.9	97.94	97.88
LineVul (unbalanced)	99.07	96.89	87.2	91.79
Multitask (unbalanced)	99.03	97.31	93.87	95.51

exception of a higher recall value. In this case, recall indicates the rate of function correctly classified as vulnerable with respect of all vulnerable functions. So, a higher recall value may indicate that the model has gained a higher coverage of the vulnerable class. The models trained on balanced data show the same overall performance.

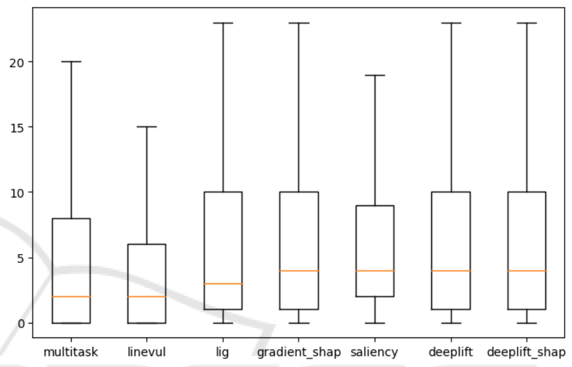
RQ2. What Are the Effects of the Multitask Approach in Vulnerability Localization?

For this RQ we leverage the attention-based approach proposed by Fu et al., computing the attention scores assigned to every line of code and evaluating them to identify the ones that represent the causes of vulnerability. We evaluate the model performance in line-level localization using the model agnostic techniques introduced by Fu et al.: Layer Integrated Gradient (LIG) (Sundararajan et al., 2017), Saliency (Simonyan et al., 2014), DeepLift (Ancona et al., 2018), (Shrikumar et al., 2017), DeepLiftSHAP (Lundberg and Lee, 2017), GradientSHAP (Lundberg and Lee, 2017). This choice is justified by the logic behind the mechanism adopted for the line-level detection task, that leverages model explainability concepts to identify the most relevant features in the prediction of the functions vulnerability status (vulnerable or not vulnerable). The benefit from this approach is dual: it shows the high transformers performance in a line level task and the attention mechanism potential as a model explainability tool.

Replicating the tests, the model registered a 63% top10 Accuracy and an IFA value of 5.22 with median 2. Compared with LineVul our model has slightly worse performance. This is explained by the task the models are trained for. LineVul is trained exclusively for vulnerability detection, so it assigns the higher attention score to the tokens which better help it detecting the vulnerability. Our model is trained instead for vulnerability detection and vulnerability classification, so it needs to assign the attention scores considering the tokens that help it classify the vulnerability. Another aspect that affect the line-level results is the true positive rate. Considering that only the vulnerable functions have a flaw-line and flaw-line-



(a) Top-10 accuracy



(b) Initial False Alarm (IFA)

Figure 2: Top-10 accuracy and IFA distributions computed with attention mechanism and model agnostic techniques compared.

index in the dataset, the line-level evaluation is performed only on the model's true positive outputs. Recall values in RQ1 indicate that our model achieved a higher true positive rate than LineVul, consisting in a higher number of samples evaluated. Therefore, this could be another possible cause of lower performance. However, the results are still better than the ones of the model agnostic explainability tools, as shown in Figures 2a and 2b.

RQ3. Does a Weighted Loss Increase the Model Performances Instead of an Average Loss?

All the previous evaluations have been done computing the average loss between multiclass and binary classification ones. With this RQ we want to study a different approach to compute the loss of the model. In this case, we run multiple experiments evaluating how a weighted loss may have an impact on model performance. Particularly, from the previous experiments it has been observed that the multiclass task is the one with the higher loss values when compared

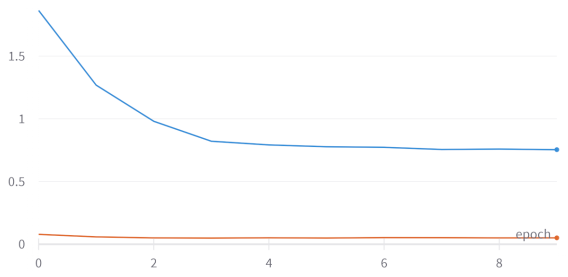


Figure 3: Binary loss (orange) and multiclass loss (blue) computed during evaluation with unbalanced data.

with the binary one, as shown in Figure 3.

The binary loss assumes values between 0.07 and 0.05, while the multiclass one has a value of 1.86 at epoch 0 and a value of 0.75 at epoch 9. From this observation, we decided to perform new tests with both the datasets leveraging different weight values for the multitask loss, keeping the value of 1.0 for the binary one. The obtained results, however, do not show significant performance improvements. Registered F1-scores presents slightly different values when compared with the ones obtained with the previous strategy. Further tests are necessary to explore new potentially ways to handle the high loss gap between our tasks.

6 RESULTS AND DISCUSSION

From the performed studies we obtained the following results:

1. The multitask approach outperforms the multiclass-only one on the balanced dataset, while the ones with unbalanced data share almost the same performance. Both apporaches show the best results when trained with an higher number of samples, even if highly unbalanced;
2. The binary tasks show better performance when trained on the balanced dataset. LineVul and our model share similar results in vulnerability detection at function-level; in the unbalanced data scenario, the higher recall value indicates that the multitask approach enhanced the true positive rate relatively to the false negative rate, that results in a lower number of vulnerable function predicted as not-vulnerable.
3. Our model has higher line-level localization performance when compared with the other benchmark techniques, but slightly worse than LineVul.
4. A weighted approach to loss functions handling has near zero impact on the model's performance.

As pointed out by this study, a critical aspect of research in vulnerability classification is related in particular to the available data and some compromise needed to be done:

1. The amount of registered vulnerable functions is very limited, hence it is very difficult to get a big and balanced dataset.
2. Some vulnerabilities occur less frequently than others and there are few registered functions representative of these categories. For this reason it is difficult to learn a representative pattern for a lot of CWEs.

Another aspect to considered is the method adopted for line-level localization. The approach proposed by Fu et al. leverages a key aspect of the transformer-based architectures, achieving state-of-the-art performance. However, from our results, we see that this is tied with the task we are training the model to. Training the model for vulnerability detection may make it assign the attention scores to the tokens that better help it to identify a vulnerability; but in our case the model needs to recognize the type of vulnerability too. Indeed, our results in line-level localization, even if better than the benchmark explainability methods, are lower than LineVul ones. As future work, we are interested in continuing the study of new ways to identify the vulnerable lines in the code for a more informative implementation of the vulnerability detector.

7 CONCLUSIONS

In this study we explored two possible transformer-based approaches for software vulnerability detection, extending the previous work by Fu et al. We performed two evaluations for both approaches to demonstrate the impact of the data used for the training on the models. We showed how using a more populated but unbalanced dataset produces the better results in vulnerability categorization, while a balanced one produce better results in vulnerability detection. Furthermore, comparing our results with Fu et al., we assessed how the multitask approach is at the same time more informative for a security analyst and equally accurate on vulnerability detection at function-level. However, the subset of classes used is limited to vulnerability distribution in the BigVul dataset. We need to explore different grouping approaches for the vulnerability and more various dataset to get a wider coverage of the known flaws in source codes. Our objective for future work is to extend the capability of our model to cover a wider range of vulnerabilities or groups of vulnerabilities. Moreover, on line-level detection our model performed slightly worse than LineVul, but still better than the other benchmark techniques considered. Our goal is to explore new approaches to upgrade the model in this task.

REFERENCES

Ancona, M., Ceolini, E., Öztireli, C., and Gross, M. (2018). Towards better understanding of gradient-based attribution methods for deep neural networks. *ICLR*.

Chakraborty, S., Krishna, R., Ding, Y., and Ray, B. (2022). Deep learning based vulnerability detection : Are we there yet ? *IEEE Transactions on Software Engineering*, 48(9):3280–3296.

Chen, Y., Ding, Z., Alowain, L., Chen, X., and Wagner, D. (2023). DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 654–668. ACM.

De Sousa, N. T. and Hasselbring, W. (2021). Javabert : Training a transformer-based model for the java programming language. In *2021 36th IEEE / ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 90–95.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). Bert : Pre-training of deep bidirectional transformers for language understanding.

Fan, J., Li, Y., Wang, S., and Nguyen, T. N. (2020). A c / c ++ code vulnerability dataset with code changes and cve summaries. In *IEEE/ACM 17th International Conference on Mining Software Repositories (MSR)*, pages 508–512. ACM.

Fang, Y., Li, Y., Liu, L., and Huang, C. (2018). Deepxss : Cross site scripting detection based on deep learning. In *ACM International Conference Proceeding Series*, pages 47–51. Association for Computing Machinery.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. (2020). Codebert : A pre-trained model for programming and natural languages. *Findings of EMNLP*.

Fu, M. and Tantithamthavorn, C. (2022). Linevul: A transformer-based line-level vulnerability prediction. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE.

Ghaffarian, S. M. and Shahriari, H. R. (2017). Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, 50(4).

Hin, D., Kan, A., Chen, H., and Babar, M. A. (2022). LineVD: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 596–607. ACM.

Kalouptsoglou, I., Siavvas, M., Ampatzoglou, A., Kehagias, D., and Chatzigeorgiou, A. (2023). Software vulnerability prediction: A systematic mapping study. *Information and Software Technology*, 164:107303.

Li, Y., Wang, S., and Nguyen, T. N. (2021). Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC / FSE, pages 292–303. Association for Computing Machinery.

Lundberg, S. M. and Lee, S.-I. (2017). A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Mamede, C., Pinconschi, E., Abreu, R., and Campos, J. (2022). Exploring transformers for multi-label classification of java vulnerabilities. In IEEE, editor, *2022 IEEE 22nd International Conference on Software Quality , Reliability and Security (QRS)*, pages 43–52.

MITRE (2023). Cwe list version 4.13. https://cwe.mitre.org/data/index.html. Accessed 15 November 2023.

Rahman, K. and Izurieta, C. (2022). A mapping study of security vulnerability detection approaches for web applications. In *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 491–494.

Shrikumar, A., Greenside, P., and Kundaje, A. (2017). Learning important features through propagating activation differences. In *Proceedings of the 34th International Conference on Machine Learning*, pages 3145–3153. PMLR.

Simonyan, K., Vedaldi, A., and Zisserman, A. (2014). Deep inside convolutional networks : Visualising image classification models and saliency maps.

Singh, K., Grover, S. S., and Kumar, R. K. (2022). Cyber security vulnerability detection using natural language processing. In *2022 IEEE World AI IoT Congress (AIIoT)*, pages 174–178.

Sundararajan, M., Taly, A., and Yan, Q. (2017). Axiomatic attribution for deep networks. In *Proceedings of the 34th International Conference on Machine Learning*, pages 3319–3328. PMLR.

Zhou, Y., Liu, S., Siow, J., Du, X., and Liu, Y. (2019). Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Zou, D., Wang, S., Xu, S., Li, Z., and Jin, H. (2021). μvuldeepecker : A deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing*, 18(5):2224–2236.