2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion) | 979-8-3503-2263-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE-COMPANION58688.2023.00057

Toward More Effective Deep Learning-based Automated Software Vulnerability Prediction, Classification, and Repair

Michael Fu
Monash University
Clayton, Australia
yeh.fu@monash.edu

***Abstract*—Software vulnerabilities are prevalent in software systems and the unresolved vulnerable code may cause system failures or serious data breaches. To enhance security and prevent potential cyberattacks on software systems, it is critical to (1) early detect vulnerable code, (2) identify its vulnerability type, and (3) suggest corresponding repairs. Recently, deep learning-based approaches have been proposed to predict those tasks based on source code. In particular, software vulnerability prediction (SVP) detects vulnerable source code; software vulnerability classification (SVC) identifies vulnerability types to explain detected vulnerable programs; neural machine translation (NMT)-based automated vulnerability repair (AVR) generates patches to repair detected vulnerable programs. However, existing SVPs require much effort to inspect their coarse-grained predictions; SVCs encounter an unresolved data imbalance issue; AVRs are still inaccurate. I hypothesize that by addressing the limitations of existing SVPs, SVCs and AVRs, we can improve the accuracy and effectiveness of DL-based approaches for the aforementioned three prediction tasks. To test this hypothesis, I will propose (1) a finer-grained SVP approach that can point out vulnerabilities at the line level; (2) an SVC approach that mitigates the data imbalance issue; (3) NMT-based AVR approaches to address limitations of previous NMT-based approaches. Finally, I propose integrating these novel approaches into an open-source software security framework to promote the adoption of the DL-powered security tool in the industry.**

***Index Terms*—Cybersecurity, Software Vulnerability, Software Security**

I. Research Problem and Hypothesis

Software vulnerabilities are security flaws, glitches, or weaknesses found in software code [11] that could be exploited or triggered by attackers. Those unresolved vulnerable programs associated with critical software systems may result in extreme security or privacy risks. For instance, the recent data breaches of Optus [2] (a telecommunications company) and Medibank [30] (a private health insurer) have put millions of customers' privacy in danger. Thus, software security is an essential aspect needed to be considered during the development of software systems to prevent irreversible crises afterwards.

Recently, researchers have proposed various Deep Learning-based approaches that can learn source code patterns during training and predict whether a software program (e.g., a file or a function) is vulnerable [6], [26], [27], [34], [39]. The DL-based methods are more accurate than program analysis-based tools that rely on predefined vulnerable code patterns to capture vulnerabilities [10]. Some also proposed DL-based methods to identify the vulnerability types (i.e., vulnerability classification) of detected vulnerabilities that can provide more in-depth analysis for security analysts [1], [14], [36]. In addition, the rise of automated vulnerability repair (AVR) using DL models opens a new possibility to suggest repairs for vulnerable programs automatically [8], [9].

However, existing DL-based approaches still have limitations that can be addressed to further improve their performance and applicability to be adopted in practice. Specifically, the vulnerability prediction approaches still provide the coarse-grained level prediction that may require more effort from end users to capture the actual vulnerable code; the vulnerability classification approaches encounter the data imbalance issue that hinders the model performance; the performance of AVR models still has much room for improvement.

In addition, the shift-left testing concept (i.e. move software testing earlier in project timelines) has been proposed to encourage performing software testing at earlier stages of development instead of late phases of development. Thus, vulnerabilities could ideally be found and fixed earlier. Program analysis (PA)-based tools such as CPP Check [29] are available in an integrated development environment (IDE) like Visual Studio Code [31]. However, to date, DL-based security tools are still not available in IDEs to support developers.

I hypothesize that *by addressing the limitations of current DL-based approaches for software vulnerability prediction (detection), classification, and repair, we can improve the accuracy and effectiveness of DL-based approaches for the aforementioned three prediction tasks.* In my dissertation, I will test this hypothesis by focusing on the following research areas:

1) **DL-based Software Vulnerability Prediction (Section II-A):** Numerous software vulnerability predictions using DL models have been proposed. In particular, RNN-based models [26], [27] treat input programs as sequences of tokens and learn the program representation word-by-word sequentially whereas GNN-based models [6], [33], [39] treat input programs as graphs and learn the program representation based on the different graph properties of code (e.g., data flow graph). However, those approaches still predict vulnerabilities at

979-8-3503-2263-7/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE-Companion58688.2023.00057

coarse-grained levels (e.g., file or function levels) that can only predict vulnerable files or functions. I plan to propose a line-level vulnerability prediction approach. I argue that such predictions can point out which line in a function is vulnerable, which may save security analysts' effort from inspecting coarse-grained predictions.

2) **DL-based Software Vulnerability Classification (Section II-B):** DL-based software vulnerability classification (SVC) approaches have been proposed to identify vulnerability types (i.e., CWE-ID [13]) [14], [36]. The SVC task is a long-tailed learning task where the model needs to combat highly imbalanced data (the number of observations of each CWE-ID is imbalanced) during training to achieve promising performance on both common and rare CWE-IDs. Previous approaches tried mitigating this challenge using data augmentation [14] or ignoring rare CWE-IDs [36]. However, the performance did not improve and the imbalance problem remains unresolved. I plan to propose a method to mitigate the imbalanced data based on the hierarchical nature of CWE-IDs. I argue that the proposed method will produce more balanced data and the models learned from which may have better accuracy for common and rare CWE-IDs.
3) **DL-based Automated Vulnerability Repairs (Section II-C):** Recently, neural machine translation (NMT)-based methods have been proposed [8], [9], [22], [28], which treat automated vulnerability repair (AVR) as a sequence-to-sequence problem where the encoders encode the input programs and the decoders generate corresponding patches or the complete repaired version of the input. In particular, Chen *et al.* [8] propose VRepair that leverages a transformer architecture to improve from RNN-based NMTs and context matching to shorten the repair length. However, the performance of VRepair is still not accurate due to the limitations mentioned in Section II-C. In addition, context matching can not distinguish repeated patterns (code snippets with the same context) in a vulnerable program and hence does not guarantee mapping repair patches perfectly to vulnerable programs. I plan to leverage a large pre-trained transformer model and develop a perfect matching technique that can distinguish repeated patterns. I argue that by addressing the limitations of the VRepair approach, we can develop a more accurate and applicable AVR method.
4) **A DL-based Software Security Tool (Section II-D):** PA-based tools such as CPP Check [29] and Checkmarx [7] have been integrated into the software development life cycle. However, DL-based automated vulnerability prediction methods are not available for developers. I plan to integrate my proposed approaches into a framework to promote the adoption of the DL-based security tool in the industry.

II. Contributions and Results Achieved So Far

In this section, I introduce each of my four main research areas through the following flow. First, I point out the existing problems and then present my proposed solution (or solutions that will be proposed). Second, I introduce the research methods used to evaluate my proposed solutions and the achieved results (if any). Last but not least, I describe the direction of future work that can be explored to further improve the solutions and contribute to the community.

A. DL-based Software Vulnerability Prediction

DL-based approaches have been proposed [6], [26], [27], [34], [39] to learn vulnerability patterns through representation learning. DL-based approaches dynamically learn the mapping between the representation of source code and the ground truth labels (i.e., whether a given piece of code is vulnerable) while program analysis-based methods rely on manual predefined vulnerability patterns. Thus, DL-based approaches were shown to be more effective than program analysis-based methods [10]. However, previous DL-based approaches are still coarse-grained and only predict at the file or function levels.

Recently, Li *et al.* [25] proposed the IVDetect approach to address the need for fine-grained vulnerability prediction based on Recurrent Neural Networks (RNN) and Graph Convolution Networks (GCN). Li *et al.* [25] demonstrated that their IVDetect approach outperforms the state-of-the-art approaches [6], [26], [27], [34], [39] on function-level and subgraph-level vulnerability prediction. Nevertheless, IVDetect has the following three limitations: (1) the RNN architecture of IVDetect is not effective to capture long-term dependencies of source code input; (2) the training data of VDetect is limited to a project-specific dataset; (3) the subgraph-level prediction is still coarse-grained.

To address the limitations of IVDetect, I proposed a finer-grained approach named LineVul [17] (accepted at MSR 2022) which can predict both function-level and Line-level Vulnerabilities. First, LineVul leverages a transformer-based architecture that can better handle the long-term dependencies of input sequence than RNNs due to the perfect memory of its self-attention mechanism [35]. Second, the transformer architecture used by LineVul was pre-trained using a masked language modelling (MLM) technique on a large 20GB of code corpus (i.e., CodeSearchNet [21]) by Feng *et al.* [16]. This gives LineVul a good initial code representation when fine-tuned on the downstream vulnerability prediction task. Third, LineVul provides line-level vulnerability predictions which are finer-grained than subgraph-level predictions by IVDetect. The line-level predictions of LineVul are obtained through intrinsic model interpretation by summing up the self-attention weight matrices from the transformer architecture.

I performed an empirical study [17] to assess the accuracy and cost-effectiveness of the LineVul approach. This study was conducted on the large-scale Big-Vul dataset [15] (same studied dataset used by IVDetect), consisting of 188k+ C/C++

functions across 348 real-world projects and 91 different vulnerability types. The high imbalance ratio between vulnerable and non-vulnerable functions and the diverse vulnerability types ensure the dataset mimics the real-world scenario of vulnerability prediction. In the study, I asked the following research questions:

(RQ1.1) How accurate is our LineVul for function-level vulnerability predictions? LineVul achieves an F1 score of 0.91, which is 160%-379% better than the state-of-the-art baselines with a median improvement of 250%. Similarly, our LineVul achieves a Precision of 0.97 and a Recall of 0.86, which outperform the baseline approaches by 322% and 19%, respectively.

(RQ1.2) How accurate is our LineVul for line-level vulnerability localization? LineVul achieves a Top-10 Accuracy of 0.65 on line-level predictions, which is 12%-25% more accurate than the other baseline approaches.

(RQ1.3) What is the cost-effectiveness of our LineVul for line-level vulnerability localization? LineVul achieves the lowest Effort@20%Recall of 0.75, which is 29%-53% less than other baseline approaches, demonstrating the cost-effectiveness of LineVul.

Furture Work. First, the performance of LineVul may be further improved if training a model using line-level labels, which I will explore in my future work. Furthermore, Jimenez *et al.* [23] has pointed out that current vulnerability prediction approaches may encounter performance downgrade when considering the realistic evaluation scenario such as the time constraint. Thus, rigorous evaluation scenarios should be explored.

B. DL-based Software Vulnerability Classification

As mentioned in the above Section II-A, various DL-based SVP methods have been proposed to detect vulnerability in different granularity. However, those approaches can not identify what type of vulnerability is detected. This may hinder security analysts to understand models' predictions and suggest mitigation strategies since SVP models only point out where is the vulnerability and fail to explain what is the detected vulnerability. Thus, software vulnerability classification (SVC) approaches have been proposed to identify the vulnerability type (i.e., CWE-ID [13]) and explain the detected vulnerabilities [1], [14], [36].

However, the distribution of different vulnerability types is highly imbalanced (i.e., long-tailed distribution) in the real world and existing SVC methods still encounter an unresolved data imbalance issue. For instance, Das *et al.* [14] leveraged data augmentation [38] in their experiments but it did not further improve the performance; Wang *et al.* [36] only focus on the top 10 frequency CWE-IDs in their experiment to mitigate the data imbalance, which hinders the model to identify rare vulnerability types.

To address the imbalance issue of the SVC task, I will propose a method based on the hierarchical nature of vulnerability types, which can reduce the imbalance ratio of a dataset and learn better models. I will empirically evaluate the proposed approach on a widely-adopted benchmark dataset and compare it with (1) other SVC models and (2) long-tailed learning techniques that focus on mitigating the imbalance training process. I will ask the following research questions to assess the proposed method:

(RQ2.1) What is the accuracy of the proposed approach for classifying software vulnerabilities (i.e., CWE-IDs)?

(RQ2.2) Does the proposed approach outperform long-tailed learning methods for imbalanced data?

(RQ2.3) What are the contributions of the components of the proposed approach?

Furture Work. Another mainstream of SVC methods is to leverage multi-task learning that could improve the model by learning with other related tasks (e.g., CVSS [12] severity score classification). However, the existing methods rely on loss summation to update the model for different tasks, which does not guarantee an optimal solution for all of the tasks [3], [19], [24]. To address this limitation, I will propose a multi-objective learning-based approach that can find out the optimal solution for each task when updating the model.

C. DL-based Automated Vulnerability Repairs

As mentioned in the above Section II-A and II-B, researchers proposed various DL-based approaches to help under-resourced security analysts predict vulnerability [6], [26], [27], [34], [39], localize the location of vulnerabilities down to the line level [17], [20], and classify the vulnerability types [1], [14], [36] to obtain the potential impact of the vulnerability and the corresponding mitigation strategy based on its type. However, security analysts still have to spend a huge amount of effort manually fixing or repairing vulnerabilities after the detection [5], [32].

Recently, Chen *et al.* [8] proposed VRepair, an automated vulnerability repair (AVR) approach based on neural machine translation (NMT) using an encoder-decoder transformer. VRepair was proposed to shorten the repair length (i.e., length of decoder outputs) using a context-matching method and address various challenges of prior work in the AVR problem (e.g., SequenceR [9], an RNN-based NMT model). However, VRepair has the following three limitations: (1) VRepair was pre-trained on a small bug-fix corpus which may not generate optimal vector representations of input source code; (2) VRepair relied on word-level tokenization with copy mechanism to handle out-of-vocabulary(OOV) and not overload the vocab size, which still has limited ability to generate new tokens that never appeared before; (3) VRepair leveraged the vanilla transformer with an absolute positional encoding which limits the ability of its self-attention to learn the relative position information of code tokens within input sequences.

To address the limitations of VRepair, I proposed a T5-based Vulnerability Repair approach named VulRepair [18] (accepted at FSE 2022). First, VulRepair employs a CodeT5 [37]

model which was pre-trained with multiple learning objectives on large code bases consisting of 8.35 million functions to generate more meaningful representations of source code. Second, VulRepair employs BPE subword tokenization to handle the OOV issue. Third, the T5 architecture used by VulRepair considers the relative position information of each embedded code token in self-attention.

I performed an empirical study [18] to assess the accuracy of the VulRepair approach and its component. This study is conducted on CVEFixes [4] and Big-Vul dataset [15] which consist of 8,482 vulnerable functions and their corresponding patches provided by Chen *et al.* [8]. Moreover, the vulnerabilities were collected from 1,754 open-source software projects with diverse 180+ vulnerability types spanning from 1999 to 2021. In the study, I asked the following research questions:

(RQ3.1) What is the accuracy of our VulRepair for generating software vulnerability repairs? VulRepair achieves a Percentage of Perfect Prediction (%PP) of 44%, which is 21% more accurate than VRepair [8] and 13% more accurate than another baseline method, CodeBERT [16].

(RQ3.2) What is the benefit of using a pre-training component for vulnerability repairs? Regardless of the model architectures being pre-trained, the pre-training corpus including both programming language and natural language improves the %PP by 30%-38% for the performance of vulnerability repair approaches. The result highlights the substantial benefits of using the pre-training component in the AVR task.

(RQ3.3) What is the benefit of using BPE tokenization for vulnerability repairs? Regardless of the model architectures, the BPE subword tokenization improves the %PP by 9%-14% for the performance of vulnerability repair approaches. The result highlights the substantial benefits of using BPE tokenization in the AVR task.

(RQ3.4) What are the contributions of the components of our VulRepair? The pre-training component of VulRepair is the most important component to achieving satisfying performance. Besides, without a proper design of the model architecture and tokenizer, the performance of VulRepair can be drastically decreased. This finding highlights that designing an NMT-based automated vulnerability repair approach is still challenging, requiring a deep understanding of modern Transformer architectures to achieve satisfying results.

Furture Work. Despite the advancement of VulRepair, there still exist two aspects of NMT-based AVR methods that are worth to be explored and improving. First, existing NMT models consider identically vulnerable and non-vulnerable parts of code and only implicitly learn to localize the vulnerability to generate corresponding repairs. Second, the context-matching repair technique proposed by Chen *et al.* [8] may fail due to the vulnerable code having repeated patterns that are used to match the repair. To address the two limitations, I will explore (1) a method that can focus more on vulnerable code when generating the corresponding repairs and (2) a method that can achieve perfect matching for the context-matching repair.

D. A DL-based Software Security Tool

As introduced in the above three sections, I will focus on improving DL-based methods to predict, classify, and repair vulnerable source code in my thesis, which helps automate the software security analysis process. In particular, Section II-A focuses on DL methods for the early vulnerability detection stage; Section II-B focuses on DL methods for identifying types of detected vulnerabilities to provide more explanations; Section II-C focuses on DL methods for suggesting repairs of detected vulnerabilities.

The program analysis-based method such as Checkmarx [7] has been integrated into software development workflows to achieve security testing during development. However, DL-based approaches have not been integrated into any IDEs to help detect security issues during software development.

To bridge the gap between DL-based vulnerability approaches and developers, I will propose a real-time software security tool that integrates my proposed methods to predict, classify, and repair vulnerable source code during software development. This tool will be developed as an extension of Visual Studio Code [31] to support developers to capture potential security issues during the early stage of the development life cycle.

I will conduct a qualitative user study to obtain software practitioners' perceptions of the proposed tool. I will answer the following research question to assess the usefulness of the proposed tool based on an empirical survey of practitioners:

(RQ4.1) How do software practitioners perceive the usefulness of the proposed DL-based security tool?

Furture Work. I plan to keep developing the tool into a framework where end users can use their models to generate predictions for each task. Moreover, this tool can also be integrated into the CI/CD pipelines to capture potential vulnerabilities after committing the code.

III. Timeline

I just completed the first year of my 3-year PhD program. In my remaining years, I will keep improving on my four research areas following the direction mentioned before.

IV. Conclusion

In my PhD program, I will address the limitations of the existing approaches and the unresolved challenges of DL-based software vulnerability prediction, classification, and repair. In particular, I will propose (1) finer-grained SVP methods that could save security analysts' effort in inspecting coarse-grained predictions; (2) SVC methods that mitigate the imbalanced data issue; (3) AVR methods that are more accurate and have perfect-matched repair patches. Last but not least, I will integrate these novel approaches into an open-source software security framework to promote the adoption of the DL-powered security tool in the industry.

REFERENCES

[1] E. Aghaei, W. Shadid, and E. Al-Shaer, "Threatzoom: Hierarchical neural network for cves to cwes classification," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2020, pp. 23–41.

[2] ASIC, "Guidance for consumers impacted by the optus data breach," https://asic.gov.au/about-asic/news-centre/news-items/guidance-for-consumers-impacted-by-the-optus-data-breach/, 2022.

[3] I. Babalau, D. Corlatescu, O. Grigorescu, C. Sandescu, and M. Dascalu, "Severity prediction of software vulnerabilities based on their text description," in *2021 23rd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 2021, pp. 171–177.

[4] G. Bhandari, A. Naseer, and L. Moonen, "Cvefixes: automated collection of vulnerabilities and their fixes from open-source software," in *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2021, pp. 30–39.

[5] T. Britton, L. Jeng, G. Carver, and P. Cheak, "Reversible debugging software "quantify the time and cost saved using reversible debuggers"," 2013.

[6] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering (TSE)*, 2021.

[7] Checkmarx, "Checkmarx," https://checkmarx.com/, 2006.

[8] Z. Chen, S. Kommrusch, and M. Monperrus, "Neural transfer learning for repairing security vulnerabilities in c code," *IEEE Transactions on Software Engineering (TSE)*, 2021.

[9] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering (TSE)*, vol. 47, no. 9, pp. 1943–1959, 2019.

[10] R. Croft, D. Newlands, Z. Chen, and M. A. Babar, "An empirical study of rule-based and learning-based approaches for static application security testing," in *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2021, pp. 1–12.

[11] CSRC, "Definition of software vulnerability," https://csrc.nist.gov/glossary/term/software_vulnerability, 2022.

[12] CVSS, "Common vulnerability scoring system (cvss)," https://nvd.nist.gov/vuln-metrics/cvss, 2003.

[13] CWE, "Common weakness enumeration (cwe)," https://cwe.mitre.org/index.html, 2006.

[14] S. S. Das, E. Serra, M. Halappanavar, A. Pothen, and E. Al-Shaer, "V2w-bert: A framework for effective hierarchical multiclass classification of software vulnerabilities," in *2021 IEEE 8th International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 2021, pp. 1–12.

[15] J. Fan, Y. Li, S. Wang, and 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 (MSR)*, 2020, pp. 508–512.

[16] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

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

[18] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "Vulrepair: a t5-based automated software vulnerability repair," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022, pp. 935–947.

[19] X. Gong, Z. Xing, X. Li, Z. Feng, and Z. Han, "Joint prediction of multiple vulnerability characteristics through multi-task learning," in *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2019, pp. 31–40.

[20] D. Hin, A. Kan, H. Chen, and M. A. Babar, "Linevd: Statement-level vulnerability detection using graph neural networks," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE, 2022.

[21] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[22] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.

[23] M. Jimenez, R. Rwemalika, M. Papadakis, F. Sarro, Y. Le Traon, and M. Harman, "The importance of accounting for real-world labelling when predicting software vulnerabilities," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 695–705.

[24] T. H. M. Le, D. Hin, R. Croft, and M. A. Babar, "Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 717–729.

[25] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021*. Association for Computing Machinery, Inc, 2021, pp. 292–303.

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

[27] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv e-prints*, pp. arXiv–1801, 2018.

[28] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 101–114.

[29] D. Marjamäki, "Cppcheck," https://cppcheck.sourceforge.io/, 2007.

[30] Medibank, "Cyber event updates and support - medibank," https://www.medibank.com.au/health-insurance/info/cyber-security/, 2022.

[31] Microsoft, "Visual studio code," https://code.visualstudio.com/, 2022.

[32] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, "Understanding the reproducibility of crowd-reported security vulnerabilities," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 919–936.

[33] V.-A. Nguyen, D. Q. Nguyen, V. Nguyen, T. Le, Q. H. Tran, and D. Phung, "Regvd: Revisiting graph neural networks for vulnerability detection," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 178–182.

[34] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and 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.

[35] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in Neural Information Processing Systems (NIPS)*, vol. 30, 2017.

[36] X. Wang, S. Wang, K. Sun, A. Batcheller, and S. Jajodia, "A machine learning approach to classify security patches into vulnerability types," in *2020 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2020, pp. 1–9.

[37] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *in Proceedings of the International Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2021.

[38] J. Wei and K. Zou, "Eda: Easy data augmentation techniques for boosting performance on text classification tasks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 6382–6388.

[39] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems (NIPS)*, 2019, pp. 10 197–10 207.