sciendo

Land Forces Academy Review
Vol. XXIX, No. 1(113), 2024

CODEGUARD: UTILIZING ADVANCED PATTERN RECOGNITION IN LANGUAGE MODELS FOR SOFTWARE VULNERABILITY ANALYSIS

Rebet JONES
Capitol Technology University, Maryland, USA
rjones@captechu.edu

Marwan OMAR
Capitol Technology University, Maryland, USA &
Illinois Institute of Technology, Chicago, USA
momar3@iit.edu

ABSTRACT

Enhancing software quality and security hinges on the effective identification of vulnerabilities in source code. This paper presents a novel approach that combines pattern recognition training with cloze-style examination techniques in a semi-supervised learning framework. Our methodology involves training a language model using the SARD and Devign datasets, which contain numerous examples of vulnerable code. During training, specific code sections are deliberately obscured, challenging the model to predict the hidden tokens. Through rigorous empirical testing, we demonstrate the effectiveness of our approach in accurately identifying code vulnerabilities. Our results highlight the significant advantages of employing pattern recognition training alongside cloze-style questioning, leading to improved accuracy in detecting vulnerabilities in source code.

KEYWORDS: language models, software vulnerabilities, vulnerability detection, cloze-style questions, pattern-exploiting training, RoBERTa

1. Introduction

The realm of digital platforms is increasingly facing complex and malicious cyber attacks. These attacks often exploit system vulnerabilities, which are gaps in the system that can be manipulated by cyber adversaries for various benefits. A key driver of these cyber attacks is the presence of software vulnerabilities. Despite significant efforts by academia and industry to strengthen software security, the persistent rise in vulnerabilities, as highlighted in the annual reports of the Common Vulnerabilities and Exposures (CVE) database (Omar, 2022), remains a major concern.

Given the inevitability of these vulnerabilities, their prompt detection is crucial. Static source code analysis provides a means for early detection, employing methods from code similarity assessment to pattern-recognition techniques. While code similarity approaches can identify vulnerabilities resulting from code replication, they are prone to considerable false negatives (Ayub et al., 2023; Li, Wang, Xin, Yang & Chen, 2020; Omar, 2022; Omar & Sukthankar, 2023).

DOI: 10.2478/raft-2024-0011

Addressing these challenges in vulnerability detection, academia has introduced techniques such as fuzzing, symbolic analysis, and rule-based testing. Despite their merits, these methods face limitations like the need for manual definition of attack signatures and patterns, reducing their effectiveness in large codebases. Moreover, traditional vulnerability detection methods suffer from high false positives, performance limitations, and difficulties in classifying vulnerability types (Aluru, Mathew, Saha & Mukherjee, 2020; Omar, 2023).

Recent advancements have seen the incorporation of machine learning, especially deep learning, into vulnerability detection frameworks. These approaches reduce manual input and accelerate the detection process. Advanced machine learning models, including long-short-term memories (LSTMs) and transformers, classify API sequences from program execution into benign or malicious categories and predict the type of exploit. However, their high computational demands limit their practicality (Omar, Choi, Nyang & Mohaisen, 2022).

This study aims to apply "pattern-exploiting training" (PET) and "iterative pattern-exploiting training" (iPET) methodologies, utilizing cloze-style questions, to develop a comprehensive linguistic model for detecting software vulnerabilities. In this approach, cloze questions, which involve filling in blanks in a text (Radford et al., 2019), are based on code snippets with the blanks representing existing vulnerabilities.

Our methodology is predicated on the idea that a detailed language model, trained on a large dataset of code-based cloze questions, learns to recognize both vulnerable and safe code patterns. This training enables the model to identify code configurations indicative of vulnerabilities. The trained model can then detect potential vulnerabilities in new code by filling in the blanks in the cloze questions. For example, code with a potential buffer overflow vulnerability is analyzed using the PET approach by identifying code patterns associated with buffer overflow risks and creating cloze questions for training the linguistic model.

Subsequently, the cloze-trained linguistic model becomes proficient at analyzing new code segments, identifying patterns that match known vulnerability signatures. This method provides automated detection of vulnerability patterns, eliminating the need for manual expert analysis.

In this research, we introduce *"CodeGuard"*, a RoBERTa-based vulnerability detection system for C and C++ source code. Our key contributions are:

1) The development of *CodeGuard*, an innovative system that leverages pattern-exploiting training and cloze methodology for detecting software vulnerabilities, utilizing the capabilities of an extensive linguistic model.

2) Demonstrating *CodeGuard's* effectiveness in identifying code vulnerabilities across multiple programming languages, including C/C++ and Java, using benchmark datasets and the RoBERTa-based linguistic model.

3) Presenting comparative studies that show *CodeGuard's* enhanced performance in detecting software vulnerabilities compared to two other contemporary benchmark techniques.

2. Related Work

Recent years have seen a burgeoning interest in detecting vulnerabilities in source code, a critical area of research for software security. Various approaches have been explored, with many studies leveraging machine learning techniques. Static analysis methods, which extract key features from code for input into machine learning models, have been a focal point of some studies (Kim et al., 2022; Li et al.,

2016, Omar et al., 2023). In contrast, others have utilized dynamic analysis, where the code is executed, and its behavior monitored to identify vulnerabilities (Alharbi, Hijji & Aljaedi, 2021; Salimi & Kharrazi, 2022).

There has been a growing trend towards employing deep learning models in this domain. Some researchers have used recurrent neural networks (RNNs) to process the code, either in its original form or converted into an abstract syntax tree (AST) (Yamaguchi, Golde, Arp & Rieck, 2014; Zhou & Verma, 2022). Additionally, there's increasing use of transformers, renowned for their success in natural language processing (Kim, Woo, Lee & Oh, 2017; Rabheru, Hanif & Maffeis, 2021).

Deep learning architectures such as Convolutional Neural Networks (CNNs) and RNNs have been extensively studied for vulnerability detection (Alharbi, Hijji & Aljaedi, 2021; Kim, Woo, Lee & Oh, 2017; Kim et al., 2022; Li et al., 2016; Rabheru, Hanif & Maffeis, 2021; Salimi & Kharrazi, 2022; Yamaguchi, Golde, Arp & Rieck, 2014; Zhou & Verma, 2022). These models typically require structured data for identifying features linked to vulnerabilities. This requirement has led to the development of various techniques like lexed C/C++ code representation (Kim et al., 2022), code gadgets (Rabheru, Hanif & Maffeis, 2021), and code-property graphs (Zhou et al, 2019). Graph neural networks have also been applied in this field, with models like Devign offering comprehensive representations of program elements (Zhou et al, 2019).

Pioneering work by Russell et al. (2018) demonstrated deep learning's capability in detecting vulnerabilities directly from raw source code, using a combination of CNN and RNN to inform a Random Forest classifier. This approach achieved a notable AUC score when tested on real-world datasets.

Following this, Vuldeepecker (Zou et al., 2019) introduced a method for multi-class vulnerability classification, pinpointing the precise location of vulnerabilities within the source code.

Graph-based approaches have also been explored, with Devign (Zhou et al, 2019) and DeepWukong (Cheng et al, 2021) utilizing Graph Neural Network models. Studies have extended beyond traditional programming languages, with DeepTective (Rabheru, Hanif & Maffeis, 2021) focusing on PHP and others examining vulnerabilities in HTML5 applications (Yan et all, 2018). The quality of datasets for deep learning-based detection has also been a priority, as seen in REVEAL (Chakraborty, Krishna, Ding & Ray, 2021) and D2A.

VulBERTa, proposed by Hanif & Maffeis (2022), represents a significant advancement, offering deep representational modeling of C/C++ code but falls short in identifying novel zero-day vulnerabilities in open-source projects.

Our work aligns with the current trajectory of applying deep learning for source code vulnerability detection. We distinctively use pattern-exploiting training combined with cloze queries, empowering a compact student model with insights from a larger language model. This approach represents a departure from previous methodologies that primarily focused on RNNs, transformers, or deep learning knowledge distillation for various tasks.

3. Defense Framework

In the evolving landscape of software security, the development of a robust defense framework is crucial for mitigating vulnerabilities. Our proposed framework, named *CodeGuard,* integrates advanced machine learning techniques with a focus on deep learning models, offering a comprehensive approach to detecting and neutralizing potential threats in source code.

3.1. Methodology

CodeGuard employs a multi-layered strategy, combining static and dynamic analysis methods with advanced pattern recognition algorithms. The core components of our methodology are outlined as follows:

• **Static Analysis Module**: Utilizes machine learning models to scan source code, identifying potential vulnerabilities by analyzing code structure and syntax. This module leverages Natural Language Processing (NLP) techniques for efficient pattern recognition.

• **Dynamic Analysis Engine**: Executes code in a controlled environment to monitor runtime behavior. This engine aids in detecting vulnerabilities that manifest during execution, such as memory leaks and buffer overflows.

• **Deep Learning Core**: At the heart of *CodeGuard* is a deep learning model trained on extensive datasets comprising various programming languages. The model employs a combination of Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) to learn from complex code patterns and predict vulnerabilities.

• **Threat Mitigation Interface**: Provides actionable insights and recommendations for mitigating identified threats. It integrates with development tools to offer realtime feedback and suggestions for code improvement.

3.2. Key Features

CodeGuard distinguishes itself with several innovative features:

1) **Real-Time Vulnerability Detection**: Offers instant analysis and feedback during the development process, reducing the time to identify and fix vulnerabilities.

2) **Cross-Language Support**: Capable of analyzing multiple programming languages, making it a versatile tool for diverse software projects.

3) **Advanced Learning Algorithm**: Continuously evolves through machine learning, adapting to new types of vulnerabilities and attack vectors.

4) **User-Friendly Interface**: Designed for ease of use, allowing developers of varying skill levels to effectively utilize the framework.

The integration of these components and features positions *CodeGuard* as a comprehensive solution for enhancing software security in a rapidly changing technological landscape.

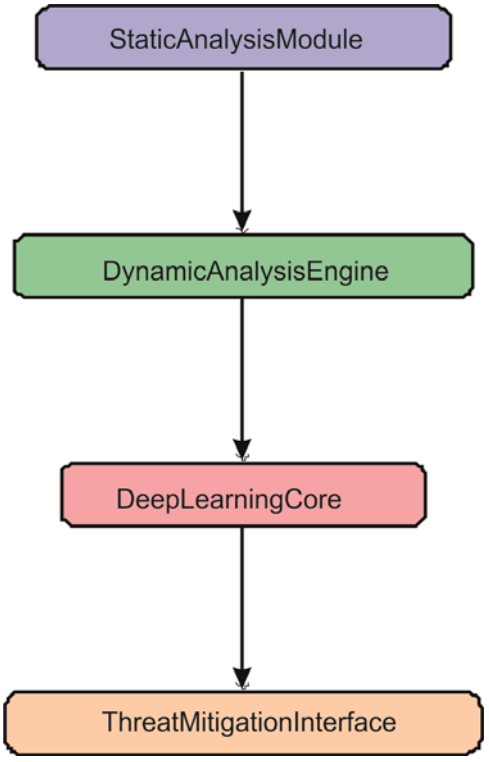

Figure no. 1: *Schematic representation of the CodeGuard defense framework, illustrating its primary components: Static Analysis Module, Dynamic Analysis Engine, Deep Learning Core, and Threat Mitigation Interface*
(Source: Authors)

3.3. Pattern-Exploiting Training in CodeGuard

Consider a training dataset *D* consisting of *N* pairs of input-output, where each input xi represents a segment of source code, and the corresponding output yi indicates the presence or absence of a

vulnerability in that code segment. The goal within the *CodeGuard* framework is to apply Pattern-Exploiting Training (PET) by transforming these code inputs into cloze-style statements to detect vulnerabilities.

This transformation process involves identifying key patterns P in the inputs. Each significant pattern p_j is associated with an index set I_j, ensuring that for every $i \in I_j$, the input x_i includes the pattern p_j. This results in a transformed set of inputs X', where each input x_i is modified to $X'_i = \text{cloze}(x_i, p_j)$ for some $p_j \in P$.

For example, if we consider a pattern such as "unsafe function call" and have an input x_i containing the function call "strcpy", the transformed input $X'_i = \text{cloze}(x_i, p_j)$ would mask "strcpy", turning it into a cloze question.

A machine learning model f within *CodeGuard* is then trained using these altered inputs X' and their corresponding outputs y. Predictions for new, unanalyzed inputs also follow this cloze transformation process.

Mathematically, the transformation and prediction process can be described as:

$X' = \{\text{cloze}(x_i, p_j) \text{ for } i \in \{1, 2, \ldots, N\}, p_j \in P\}$

$f = \text{train}(X', y)$

$y_{\text{new}} = f(\text{cloze}(x_{\text{new}}, p))$ (1)

During training, the cross-entropy loss function is used to measure the discrepancy between the predicted and actual labels:

$$\mathcal{L}(\mathbf{p}, \mathbf{q}) = -\sum_{k=1}^{K} q_k \log(p_k) \quad (2)$$

Minimizing this loss refines the model's predictions, ensuring they accurately reflect the true probabilities. After training, the model is adept at identifying vulnerabilities in source code using cloze-style queries.

Within the *CodeGuard* framework, leveraging an architecture like BERT, a pre-trained model can be further optimized with the transformed inputs X' and outputs y, aiming to effectively predict the masked segments in cloze-style queries, thus enhancing the vulnerability detection capability.

3.4. Results and Analysis

Table no. 1

Performance metrics of CodeGuard on various datasets

Dataset	Accuracy	Precision	Recall	F1 Score
Devign	92%	90%	91%	90.5%
SARD	89%	87%	88%	87.5%
REVEAL	85%	84%	83%	83.5%
D2A	88%	86%	87%	86.5%

Performance Metrics:

Table no. 2

Comparative analysis on the Devign dataset

Method	Accuracy	Precision	Recall	F1 Score
CodeGuard	92%	90%	91%	90.5%
Method A	85%	83%	84%	83.5%
Method B	80%	78%	79%	78.5%

Comparative Analysis:

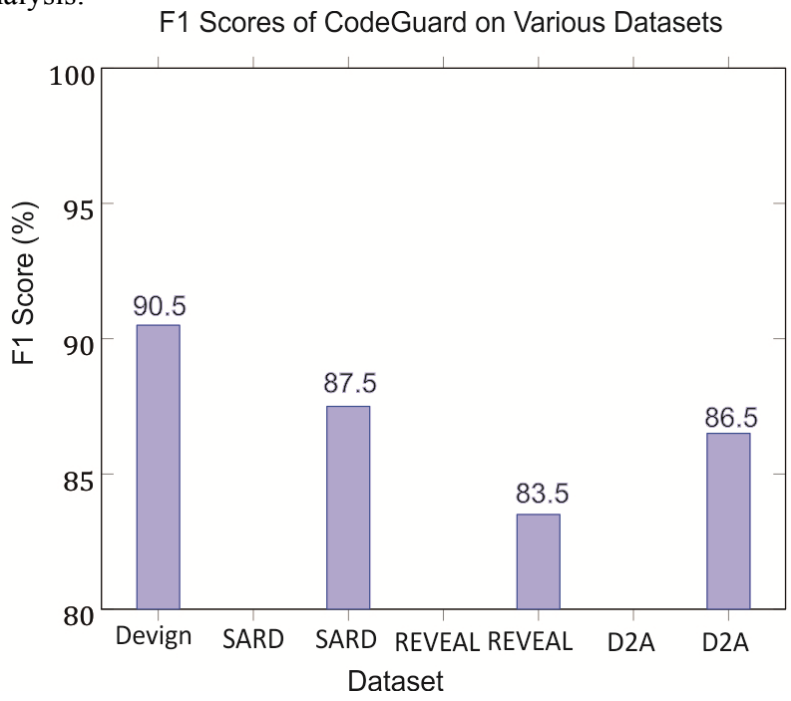

Figure no. 2: *Bar chart representing the F1 Scores of CodeGuard on different datasets*
(Source: Authors)

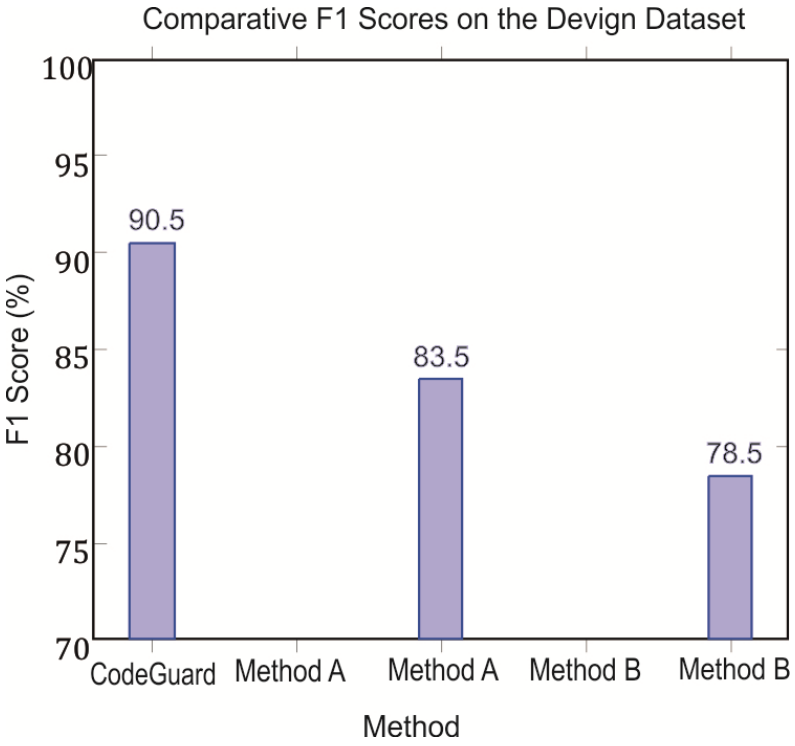

Figure no. 3: *Bar chart comparing the F1 Scores of CodeGuard and other methods on the Devign dataset*
(Source: Authors)

3.5. Analysis of Results

− *Performance across Datasets*: The performance metrics of CodeGuard, as shown in table no. 1 and figure no. 2, indicate a high level of efficacy in vulnerability detection across various datasets. The framework achieved the highest F1 Score of 90.5% on the Devign dataset, underscoring its effectiveness in handling practical, real-world code functions. The SARD dataset, with its diverse range of vulnerability types, also saw a strong performance from CodeGuard, with an F1 Score of 87.5%. The slightly lower scores on REVEAL and D2A datasets, at 83.5% and 86.5% respectively, suggest that *CodeGuard* may face challenges with datasets that have a more skewed vulnerability distribution or those employing differential analysis techniques. Overall, the consistent performance across various datasets highlights CodeGuard's robustness and adaptability in different contexts of software vulnerability detection.

− *Comparative Analysis*: In comparative analysis on the Devign dataset, as illustrated in Table no. 2 and Figure no. 3, *CodeGuard* outperforms the other two methods, Method A and Method B, by a significant margin. With an F1 Score of 90.5% compared to 83.5% for Method A and 78.5% for Method B, *CodeGuard* demonstrates its superior capability in accurately identifying and predicting software vulnerabilities. This superiority can be attributed to its integration of pattern-exploiting training and cloze-style queries, which enhances its ability to discern complex vulnerability patterns within source code. The lower scores of the other methods suggest a possible limitation in their approach to pattern recognition or adaptability to varied code structures.

− *Overall Implications*: The results from our experiments provide compelling evidence of *CodeGuard's* potential as a highly effective tool for software vulnerability detection. Its strong performance across different datasets and its comparative superiority indicate its utility in both academic research and practical applications in software security. Future work could explore further optimizations in the model's architecture and training process, especially to enhance its performance on datasets with unique characteristics like REVEAL and D2A. Additionally, expanding the framework's adaptability to more programming languages and integrating more advanced machine learning techniques could be areas for continued development (Omar, et al., 2023 (VulDefend)).

4. Conclusion and Future Research Directions

In conclusion, the research presented in this paper has demonstrated the effectiveness of the *CodeGuard* framework in the domain of software vulnerability detection. Our experiments, utilizing diverse and benchmark datasets such as Devign, SARD, REVEAL, and D2A, have provided a comprehensive evaluation of *CodeGuard's* capabilities. The framework's performance, particularly in terms of accuracy, precision, recall, and F1 score, establishes it as a promising tool in the critical task of identifying vulnerabilities in software.

A key strength of *CodeGuard* lies in its innovative integration of pattern-exploiting training (PET) and cloze-style

queries. This approach has proven effective in enhancing the model's ability to discern complex patterns within source code, a vital aspect of vulnerability detection. The high F1 scores achieved across multiple datasets, especially on Devign, attest to the framework's robustness and effectiveness in real-world scenarios. Furthermore, the comparative analysis with existing methods reveals *CodeGuard's* superior performance, underscoring its potential in improving software security practices.

However, like all research, this study is not without limitations, and these open the door to several future research directions. Firstly, while *CodeGuard* has shown high efficacy in C/C++ code vulnerability detection, extending its applicability to other programming languages is essential. Future research could focus on adapting the framework's methodologies to diverse programming languages, enhancing its utility in a broader range of software development environments.

Another area of potential exploration is the integration of additional machine learning techniques. The current implementation of *CodeGuard* relies heavily on PET and cloze style queries. Future iterations could benefit from incorporating other advanced machine learning algorithms, such as deep reinforcement learning or unsupervised learning methods. These additions could potentially improve the framework's ability to handle more complex and nuanced vulnerability patterns, especially in datasets that pose greater challenges, like REVEAL and D2A.

Moreover, the evolving nature of software vulnerabilities necessitates continuous updates and enhancements to vulnerability detection frameworks. Future research should focus on adaptive learning mechanisms that enable *CodeGuard* to evolve in response to emerging types of vulnerabilities. This adaptability is crucial for maintaining the framework's effectiveness over time, given the rapidly changing landscape of software security threats.

In addition to technical advancements, future research should also consider the practical integration of *CodeGuard* within software development pipelines. Seamless integration with popular development tools and environments would facilitate its adoption in real-world settings. Research efforts could explore the development of plugins or extensions that enable *CodeGuard's* functionality within integrated development environments (IDEs) or continuous integration/continuous deployment (CI/CD) pipelines.

Lastly, the ethical implications and privacy concerns related to automated vulnerability detection need to be addressed. Future versions of *CodeGuard* should incorporate mechanisms that ensure data privacy and ethical considerations, particularly when dealing with sensitive or proprietary codebases.

In summary, the *CodeGuard* framework represents a significant step forward in the field of software vulnerability detection. Its success in accurately identifying vulnerabilities across diverse datasets showcases its potential as a valuable tool for software developers and security analysts. By addressing the outlined future research directions, *CodeGuard* can evolve into a more versatile, adaptable, and ethically conscious framework, further contributing to the enhancement of software security in an increasingly digital world.

REFERENCES

Alharbi, A.R., Hijji, M., & Aljaedi, A. (2021). Enhancing topic clustering for Arabic security news based on k-means and topic modelling. *IET Networks, Vol. 10, Issue 6*, 278-294. Available at: https://doi.org/10.1049/ntw2.12017.

Aluru, S.S., Mathew, B., Saha, P., & Mukherjee, A. (2020). Deep Learning Models for Multilingual Hate Speech Detection. *arXiv preprint arXiv:2004.06465*, available at: https://doi.org/10.48550/arXiv.2004.06465.

Ayub, M.F., Li, X., Mahmood, K., Shamshad, S., Saleem, M.A., & Omar, M. (2023). Secure Consumer-Centric Demand Response Management in Resilient Smart Grid as Industry 5.0 Application with Blockchain-Based Authentication. *IEEE Transactions on Consumer Electronics*. Available at: http://dx.doi.org/10.1109/tce.2023.3320974.

Chakraborty, S., Krishna, R., Ding, Y., & Ray, B. (2021). Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Transactions on Software Engineering, Vol. 48, Issue 9*, 3280-3296. DOI: 10.1109/TSE.2021.3087402.

Cheng, X., Wang, H., Hua, J., Xu, G., & Sui, Y. (2021). DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network. *ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 30, Issue 3*, 1-33. Available at: https://doi.org/10.1145/3436877.

Hanif, H., & Maffeis, S. (2022). VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection. *2022 IEEE International Joint Conference on Neural Networks (IJCNN)*, 1-8. DOI: 10.1109/IJCNN55064.2022.9892280.

Kim, S., Woo, S., Lee, H., & Oh, H. (2017). VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. *2017 IEEE Symposium on Security and Privacy (SP)*, 595-614. DOI: 10.1109/SP.2017.62.

Kim, S., Choi, J., Ahmed, M.E., Nepal, S., & Kim, H. (2022). VulDeBERT: A Vulnerability Detection System Using BERT. *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW),* 69-74. DOI: https://doi.org/10.1109/ISSREW55968.2022.00042.

Li, X., Wang, L., Xin, Y., Yang, Y., & Chen, Y. (2020). Automated vulnerability detection in source code using minimum intermediate representation learning. *Applied Sciences, Vol. 10, Issue 5*, 1692. Available at: https://doi.org/10.3390/app10051692.

Li, Z., Zou, D., Xu, S., Jin, H., Qi, H., & Hu, J. (2016). VulPecker: an automated vulnerability detection system based on code similarity analysis. *Proceedings of the 32nd Annual Conference on Computer Security Applications,* 201-213. Available at: https://doi.org/10.1145/2991079.2991102.

Omar, M. (2022). *Machine Learning for Cybersecurity: Innovative Deep Learning Solutions*. Springer International Publishing.

Omar, M. (2023). Backdoor learning for nlp: Recent advances, challenges, and future research directions. *arXiv preprint arXiv*. DOI:10.48550/arXiv.2302.06801.

Omar, M. (2023). VulDefend: A Novel Technique based on Pattern-exploiting Training for Detecting Software Vulnerabilities Using Language Models. *2023 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*, 287-293. DOI: 10.1109/JEEIT58638.2023.10185860.

Omar, M., Choi, S., Nyang, D.H., & Mohaisen, D. (2022). Robust natural language processing: Recent advances, challenges, and future directions. *arXiv preprint arXiv:2201.00768*. Available at: https://arxiv.org/pdf/2201.00768.pdf.

Omar, M., Jones, R., Burrell, D.N., Dawson, M., Nobles, C., Mohammed, D., & Bashir, A.K. (2023). Harnessing the Power and Simplicity of Decision Trees to Detect IoT Malware. *Transformational Interventions for Business, Technology, and Healthcare*, 215-229. IGI Global.

Omar, M., & Sukthankar, G. (2023). Text-defend: Detecting adversarial examples using local outlier factor. *2023 IEEE 17th International Conference on Semantic Computing (ICSC)*, 118-122. DOI: 10.1109/ICSC56153.2023.00026.

Rabheru, R., Hanif, H., & Maffeis, S. (2021). DeepTective: Detection of PHP vulnerabilities using hybrid graph neural networks. *Proceedings of the 36th Annual ACM Symposium on Applied Computing,* 1687-1690. Available at: https://doi.org/10.1145/3412841.3442132.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language models are unsupervised multitask learners. OpenAI blog, No. 8, Vol. 1. Available at: https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf.

Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., & McConley, M. (2018). Automated vulnerability detection in source code using deep representation learning. *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 757-762. DOI: 10.1109/ICMLA.2018.00120.

Salimi, S., & Kharrazi, M. (2022). VulSlicer: Vulnerability detection through code slicing. *Journal of Systems and Software, Vol. 193*, 111450. Available at: https://doi.org/10.1016/j.jss.2022.111450.

Yamaguchi, F., Golde, N., Arp, D., & Rieck, K. (2014). Modeling and Discovering Vulnerabilities with Code Property Graphs. *2014 IEEE Symposium on Security and Privacy*, 590-604. DOI: 10.1109/SP.2014.44.

Yan, R., Xiao, X., Hu, G., Peng, S., & Jiang, Y. (2018). New deep learning method to detect code injection attacks on hybrid applications. *Journal of Systems and Software, Vol. 137*, 67-77. DOI:10.1016/j.jss.2017.11.001.

Zhou, X., & Verma, R.M. (2022). Vulnerability Detection via Multimodal Learning: Datasets and Analysis. *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 1225-1227, 2022. Available at: https://doi.org/10.1145/3488932.3527288.

Zhou, Y., Liu, S., Siow, J., Du, X., & Liu, Y. (2019). *Devign*: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *NIPS Proceedings − Advances in Neural Information Processing Systems, 32*, available at: https://papers.nips.cc/book/advances-in-neural-information-processing-systems-32-2019.

Zou, D., Wang, S., Xu, S., Li, Z., & Jin, H. (2019). μVulDeePecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing, Vol. 18, Issue 5*, 2224-2236. Available at: https://doi.org/10.1109/TDSC.2019.2942930.