The ISC Int'l Journal of Information Security
ISeCure

November 2023, Volume 15, Number 3 (pp. 51–58)
http://www.isecure-journal.org

Presented at the ISCISC'2023 in Tehran, Iran.

Using ChatGPT as a Static Application Security Testing Tool **

Atieh Bakhshandeh [1,*], Abdalsamad Keramatfar [1], Amir Norouzi [1], and Mohammad M. Chekidehkhoun [1]

[1] *Research Center for Development of Advanced Technologies, Tehran, Iran*

ARTICLE INFO.

Keywords:
Artificial Intelligence-based Code Review, ChatGPT Model, Common Weakness Enumeration, Static Application Security Testing, Vulnerability Detection

Type:
Research Article

doi:
10.22042/isecure.2023.182082

ABSTRACT

In recent years, artificial intelligence has had a conspicuous growth in almost every aspect of life. One of the most applicable areas is security code review, in which a lot of AI-based tools and approaches have been proposed. Recently, ChatGPT has caught a huge amount of attention with its remarkable performance in following instructions and providing a detailed response. Regarding the similarities between natural language and code, in this paper, we study the feasibility of using ChatGPT for vulnerability detection in Python source code. Toward this goal, we feed an appropriate prompt along with vulnerable data to ChatGPT and compare its results on two datasets with the results of three widely used Static Application Security Testing tools (Bandit, Semgrep, and SonarQube). We implement different kinds of experiments with ChatGPT and the results indicate that ChatGPT reduces the false positive and false negative rates and has the potential to be used for Python source code vulnerability detection.

1 Introduction

Today, almost all technologies are strongly dependent on source code. Therefore, code is of increasing importance. A glance at the number of lines of used codes in some well-known tools is evidence for this claim. For instance, the number of GitHub repositories increased from 100 million in 2018 to 200 million in 2022 [1]. The increase in the amount of code will lead to more security requirements in programming. MITRE [1] and other studies indicate the growth in the number of vulnerabilities in recent years [2–8]. Specifically, software vulnerability is of great importance. A software vulnerability is a technical vulnerability that can be used for violating its security policies. Such vulnerabilities can be exploited which in turn leads to data leakage or data tampering and even denial of services.

Static source code analysis is a method for finding code vulnerabilities that is done by automatically examining the source code without having to execute the program. Static Application Security Testing (SAST) tools analyze a piece of code or a compiled version of it to identify its security problems. Cov-

* Corresponding author.
**The ISCISC'2023 program committee effort is highly acknowledged for reviewing this paper.
Email addresses: bakhshandeh@rcdat.com, keramatfar@rcdat.ir, norouzi@rcdat.ac.ir, chekidekhoon@rcdat.ir
ISSN: 2008-2045

[1] Massachusetts Institute of Technology Research and Engineering

ering a wide range of errors and high accuracy are two important features of SAST tools [9]. Most of the well-known SAST tools such as Semgrep, Bandit, and SonarQube often use rule-based techniques to find the vulnerable patterns of code. However, these tools have been shown to have some flaws, including a high rate of false positives and false negatives [4]. The more false positives a SAST tool returns, the more time and effort is required by a security expert to validate the findings of the SAST tool. Moreover, this will increase the error rate by humans, which may then lead to ignoring some vulnerabilities. On the other hand, a high rate of false negatives leads to catastrophic events.

In recent years, Machine Learning (ML) and deep learning have had remarkable advances in various areas such as natural language processing [10, 11]. Therefore, considering the high similarity between code and natural languages, the deep learning-based models are expected to be successful in code-processing tasks. Likewise, studies in this area have shown the interest of researchers in using deep learning techniques in vulnerability detection [12, 13]. Machine learning models can automatically learn the patterns of software vulnerabilities based on datasets. Furthermore, research indicates that ML models have fewer false positives compared to SAST tools [6, 14]. Results of a new research have shown the superior performance of deep learning-based models over three open-source tools in C/C++, reducing false positive and negative rate at the same time [15].

Recently, ChatGPT, an AI-powered chatbot tool that uses Natural Language Processing (NLP) and machine learning algorithms to understand and respond to customer inquiries, has drawn a lot of attention. ChatGPT is vital for business professionals for several reasons. It can help save time and resources by automating tasks requiring human intervention. An important point to note is that ChatGPT has been trained with a huge amount of data till 2021 so it can be a great help in finding known patterns in thousands of packages in an automated way. The model is also trained on a large amount of code and is thus able to recognize common patterns. In this paper, we evaluate the performance of ChatGPT in identifying security vulnerabilities of Python codes and compare the results with three well-known SAST tools for Python vulnerability detection (Bandit, Semgrep, and SonarQube). The reason for choosing the Python language is that in 2022, Python was known to be the most popular programming languages along with Java, based on the Popularity of Programming Language Index (PYPL) and IEEE reports. Also, Stackscale ranked Python in third place [16]. Although Python is mainly used in the scope of machine learning and data science, its applications are not only limited to these fields, and with its famous frameworks such as Django and Flask, it is prone to vulnerabilities. The rest of the paper is organized as follows: In Section 2, we provide a brief literature review of this area. Section 3 is dedicated to the datasets we used. Section 4 provides the details of the experiments we performed with ChatGPT. In Section 5, we present the evaluation and analysis of the obtained results [2] . In Section 6 we discuss some factors that may threaten the validity of the results. Finally, Section 7 concludes the paper.

2 Related Work

In this section, we review some of the works that used different kinds of AI models for vulnerability detection. Note that we did not focus on works that proposed models for repairing the identified vulnerabilities. When it comes to artificial intelligence, the main idea is the use of supervised learning. Therefore, various machine learning models used methods of feature engineering such as the number of lines of the code, code complexity, and the number of operations and also utilized textual features[15, 17]. In general, research shows that text-based models have better performance over feature engineering and the studies also admit that machine learning models outperform the existing SAST tools.

Recently, more research has been devoted to deep learning. In this scope, researchers often used different deep learning models such as Convolutional Neural Networks (CNN), Long Short-Term Memory (LSTM), and Multilayer Perceptron (MLP) [13, 18–20]. Some of the models were based on different kinds of code property graphs and used Graph Neural Networks [13, 14], while some others only relied on tokens [20]. A new study has investigated the way deep learning models function in vulnerability detection tasks. The results of this study reveal some points: First, the results of different models are not compatible with each other. Second, fine-tuned models have shown better performance in this field. Third, usually 1000 samples of each class are enough for the training of a neural network and, finally, models usually use the same features for prediction [21]. Although studies approved the superiority of graph-based models, a new study indicates the superior performance of transformer-based models over graph-based ones [22]. In 2022, Hanif and Maffeis proposed a model named VulBERTa [23]. This model is based on RoBERTa and is used for vulnerability detection in C/C++ codes. Another recent study also has used BERT architecture and CodeBERT vectors for predicting code vulnerabilities. The results of this study prove

[2] https://github.com/abakhshandeh/ChatGPTasSAST.git

the superiority of transformer-based models over traditional deep-learning models and also graph-based models [24]. Overall, it seems that transformer-based models are effective in this area. Another recent work has evaluated ChatGPT as a large language model for detecting vulnerabilities in Java source codes and compared the results with a dummy classifier and achieved no better results than it [25]. However, there is still no academic study about comparing the results of the ChatGPT model with traditional SAST tools for Python. This paper aims to answer the question of whether the ChatGPT model is outperforming SAST tools or not.

3 Datasets Description

In this section, we provide the details about our dataset and the labels we used. Our dataset consists of 156 Python code files. These files contain 130 files of the securityEval dataset which is proposed in [26]. As the authors mentioned, these 130 files cover 75 vulnerability types that are mapped to Common Weakness Enumeration (CWE). The remaining 26 files belong to a project called PyT in which the author developed a tool for Python code vulnerability detection and used these 26 vulnerable code files for evaluating his tool [27, 28]. Since the used datasets do not provide a specific line of vulnerability, a security expert on our team rechecked the data and specified the vulnerable line. We identified the corresponding line of code of CWEs that were assigned in the labels of these files with the help of a security expert. The datasets' information and the distribution of their corresponding labels are presented in Table A.1 in Appendix A.

4 Working with ChatGPT API

In this section, we provide the details of the process of utilizing the ChatGPT model API for identifying vulnerabilities. In this study, we used the GPT-3.5-Turbo model. The GPT-3.5-Turbo model can accept a series of messages as input, unlike the previous version which only allowed a single text prompt. This capability provides some interesting features, such as the ability to store prior responses or queries with a predefined set of instructions with context. This is likely to improve the generated response. The GPT-3.5-Turbo model is a superior option compared to the GPT-3 model, as it offers better performance across all aspects while being 10 times more cost-effective per token. We did four kinds of experiments using the GPT-3.5-Turbo model.

(1) In our first experiment, we give the model the vulnerable files and ask it whether they contain any security vulnerabilities or not, without specifying the corresponding CWEs. We ask the model to just return the line number of the vulnerability if it contains any. Then, we compare these lines with ground truth labels. In effect, this experiment is a binary classification.

(2) In our second experiment, we provide the list of the corresponding CWEs and ask the model to find the vulnerabilities from the labels' list in the Python vulnerable file. In this experiment, we ask the model to respond in JSON format like [{"label": "CWE-X", "line of Code": "line no."}] so we can compare our results with those of SAST tools.

(3) In the third experiment, for each of the vulnerable files, we give the model all the labels returned from Bandit, Semgrep, and SonarQube tools for the Python code, as the classes that ChatGPT should use. We then ask the model whether each vulnerable file contains any of those vulnerabilities or not. Here, the main difference with our second experiment is that we specify the classes per vulnerable file separately. In other words, we use the model as an assistant for the SAST tools to verify the detected vulnerabilities by them. In this experiment, we use the same JSON format as the second experiment for the responses. Note that in this experiment, although we provide the labels' list beforehand for each vulnerable file, in some cases the model has returned a new CWE which is not among its input labels. This is a natural behavior seen from a language model and to address this issue in our evaluation, we consider two cases: In one case, we ignore the new labels and calculate the metrics without considering them. This policy can reduce the number of false positives of SAST tools. In another case, we consider them as well and this time the number of false negatives may decrease.

(4) In our fourth experiment, we do not provide any label list for the model and ask it to detect the vulnerabilities in the files and determine their corresponding CWEs from its own trained knowledge. Here, the format of the responses is the same JSON structure as the previous experiments.

To use the model for our experiments, we put all the vulnerable Python codes of our dataset in a directory and then we called GPT-3.5 API with an optimized prompt for each of the vulnerable Python files. The choice of prompt is the most challenging task in this process, as it has a direct effect on the results the model provides. We optimize our prompts according to [29]. Table 2 provides the list of prompts we used for each experiment.

4.1 Parameters

The parameters of the experiment are the prompt which contains the instructions the model will execute and a parameter called temperature that determines the randomness level of the model response. The temperature can take the values between 0 to 6 with 6 giving the most random output. Because we want to specify labels for a piece of code, the nature of our job is deterministic, so we set the temperature to zero to get the most stable behavior. Another important parameter is the prompt that is very influential in the results and adjusting it to get the best results is a challenging task. We used the prompts given in Table 2. We chose our prompts based on [29]. According to [29], some of the key points to create an optimized prompt are: using delimiters such as triple quotes or triple backticks to specify the piece of code, asking for structured output such as JSON, HTML, etc., specifying the steps to complete a task in a clear way, instructing the model to work out its own solutions before rushing to a conclusion. To show the sensitivity of ChatGPT to its prompts, an example is provided in Table 1 where a prompt with minor modification is given to the model, and the model responds with different answers in which the second response contains one less vulnerability compared to the first one.

Table 1. Comparison of the responses of two prompts with minor change

Prompt	Response
find all the vulnerabilities with the CWE standard in the Python code which is delimited with triple backticks. Also, give the line of the vulnerability in the code. python code:‴{Vul˙code}‴ Format your response as a list of JSON objects with "label" and "line of Code" as the keys for each element. only answer with JSON.	[{ "label": "CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')", "line of Code": "command = 'echo' + param + ' ¿¿ ' + 'menu.txt'" }, { "label": "CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')", "line of Code": "subprocess.call(command, shell=True)" }, { "label": "CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')", "line of Code": "subprocess.call('echo Menu: ¿ menu.txt', shell=True)" }]
find all the vulnerabilities with the CWE standard in the Python code. also give the line of the vulnerability in the code. python code: ‴{Vul˙code}‴ Format your response as a list of JSON objects with "label" and "line of Code" as the keys for each element. only answer with JSON.	[{ "label": "Command Injection", "line of Code": "subprocess.call(command, shell=True)" }, { "label": "Command Injection", "line of Code": "subprocess.call('echo Menu: ¿ menu.txt', shell=True)" }]

5 Results

In this section, we provide the results of our experiments. First we explain the metrics we use for evaluating our work and then we present GPT-3.5 results and compare them with three popular SAST tools for Python vulnerability detection. To be more precise, we perform the following actions: We give a dataset of 156 vulnerable python codes to Bandit, Semgrep and SonarQube SAST tools and we also query the ChatGPT model with our dataset using the appropriate prompts. We then calculate the following metrics for each of the tools' results and the model result based on our ground truth labels. Finally, we compare the results of the tools with the GPT-3.5 model.

5.1 Evaluation Metrics

In classification, we have condition positive which indicates the number of real positive cases in the data. Similarly, there is a condition negative which is the number of real negative cases in the data. Based on these conditions, there will be four parameters: true positive (TP) which is the number of positive examples labeled as such, true negative (TN) which is the number of negative examples labeled as such, false positive (FP) that is the number of negative examples labeled as positive and false negative (FN) that is the number of positive examples labeled as negative. We define precision, recall, and F-measure according to the following formulas [30].

- **Precision:** It answers the question that out of all the examples the classifier labeled positive, what proportion were correct? It is defined according to the following equation:

$$precision = \frac{TP}{TP + FP} \tag{1}$$

- **Recall:** It answers the question that out of real positive examples, what proportion did the classifier label as positive? It is defined by the following formula:

$$recall = \frac{TP}{TP + FN} \tag{2}$$

- **F-measure:** It is a measure that combines precision and recall and is defined according to the following formula:

$$F = 2 \times \frac{precision \times recall}{precision + recall} \tag{3}$$

Table 2. GPT-3.5 prompts used. Vul˙code refers to the vulnerable code, and labels1 is the list of labels of all vulnerable files and labels2 is the labels of each vulnerable file that is iterated through a loop.

Experiment No.	Prompt
*Experiment*1	You will be provided with a Python code delimited by triple backticks. If it contains any security vulnerability, identify the lines of vulnerable code and only write the line in quotation If the code does not contain a vulnerability, then simply write None. python code: '''{Vul˙code}'''
*Experiment*2	Which of the following vulnerabilities from the list of vulnerabilities exist in the Python code which is delimited with triple backticks. also give the line of the vulnerability in the code. python code:'''{Vul˙code}''' list of vulnerabilities: {", ".join(labels1)} Format your response as a list of JSON objects with "label" and "line of Code" as the keys for each element. only answer with JSON.
*Experiment*3	Which of the following vulnerabilities from the list of vulnerabilities exist in the Python code which is delimited with triple backticks. also give the line of the vulnerability in the code. Python code: '''{Vul˙code}''' list of vulnerabilities: {", ".join((labels2)} Format your response as a list of JSON objects with "label" and "line of Code" as the keys for each element. only answer with JSON.
*Experiment*4	Your task is to determine whether the following Python code which is delimited with triple backticks, is vulnerable or not? identify the following items: - CWE of its vulnerabilities. - lines of vulnerable code. Format your response as a list of JSON objects with "label" and "line of Code" as the keys for each vulnerability. If the information isn't present, use "unknown" as the value. Make your response as short as possible and only answer with JSON. python code:'''{Vul˙code}'''

5.2 Analyzing Results

In this section, we present the results based on the mentioned metrics in the previous section. The results for *Experiment*1 in which we did not ask the model to return the CWEs, are provided in Table 3. The precision for the model in this experiment is not better than the other three tools. Furthermore, the low recall suggests that using this model for only detecting vulnerable lines of a code does not give any better results than SAST tools since low recall leads to a high false negative rate. Likewise, the results of *Experiment*2, which are presented in Table 4, indicate that using the GPT-3.5 model with all the classes given as labels does not provide superior results in comparison with the SAST tools. Note that

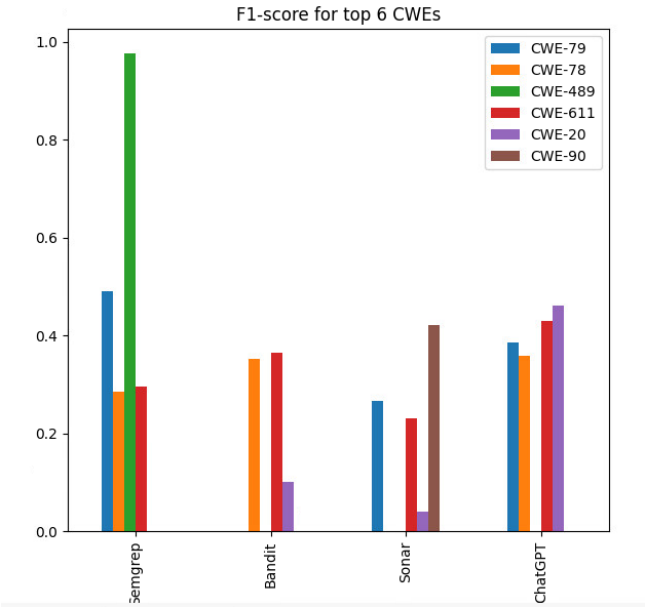


Figure 1. F1-score of top 6 CWE classes in *Experiment*3 (case 2)

in this experiment, the order of the given labels to the GPT-3.5 model has a high impact on the generated results from the model. This is because when the order of the labels is changed, the prompt is modified and as we mentioned before, the prompt has a great effect on the model's results. Therefore, we gave the labels in a random order. Here, we reach the same conclusion as [25] in which the authors concluded that the capabilities of the ChatGPT model for detecting vulnerabilities in code are limited.

The results of *Experiment*3 in which we provided the classes per vulnerable file, are given in Table 5. Here the figures indicate that case 1, in which we do not accept the new labels returned from the model, has produced better results than case 2. These results are significantly better than those of SAST tools in this experiment. The F1-score for the top 6 CWEs in terms of frequency is illustrated in Figure 1 for this experiment. This behavior shows that using ChatGPT as an assistant along with SAST tools can be a good idea. Moreover, if we do not provide any labels for the model and ask it to return the CWEs of the vulnerable codes from its own knowledge, as we did in *Experiment*4, we obtain the results in Table 6 which are comparable to the SAST tools. By and large, our experiments show that using the ChatGPT model as an assistant for SAST tools can provide hopeful results.

6 Threats to Validity

In this Section, we discuss some factors in our experiments that could affect the correctness of the results. Our biggest challenge was the choice of the prompts of

	Precision	Recall	F1
Semgrep	0.6694	**0.1504**	**0.2457**
Bandit	0.7450	0.1447	0.2424
SonarQube	**0.9104**	0.1161	0.2060
GPT-3.5	0.7413	0.0819	0.1475

Table 3. Results of *Experiment*1 (binary classification)

	Precision	Recall	F1
Semgrep	**0.4682**	**0.1123**	**0.1812**
Bandit	0.3168	0.0609	0.1022
SonarQube	0.3283	0.0419	0.0743
GPT-3.5	0.1659	0.0761	0.1044

Table 4. Results of *Experiment*2 (selecting from the list)

	Precision	Recall	F1
Semgrep	0.4682	0.1123	0.1812
Bandit	0.3168	0.0609	0.1022
SonarQube	0.3283	0.0419	0.0743
Experiment3,GPT-3.5-Case 1	**0.7807**	**0.2781**	**0.4101**
Experiment3,GPT-3.5-Case 2	0.333	0.1542	0.2109

Table 5. Results of *Experiment*3 (SAST assistant)

	Precision	Recall	F1
Semgrep	**0.4682**	0.1123	**0.1812**
Bandit	0.3168	0.0609	0.1022
SonarQube	0.3283	0.0419	0.0743
GPT-3.5	0.3350	**0.1238**	0.1808

Table 6. Results of *Experiment*4 (free classification)

ChatGPT. There are some metrics for measuring the effectiveness of a prompt for LLMs. In [31] naturalness and expressiveness are mentioned as two important factors for a prompt. Here, we tried to choose the most efficient prompts in terms of these metrics and also based on what was explained in Section 4.1 [29]. However, it is possible that a more careful selection of the prompt can affect the results. Another factor that may also affect the results is the size of the dataset and its accessibility on the Internet. Furthermore, the distribution of the CWEs of the dataset is of great importance. To overcome this threat, we chose three different datasets for better generalization of the vulnerabilities they cover, but there may be still a few coverage of the vulnerabilities. Moreover, we only compare this model with three SAST tools for Python language. Perhaps, further SAST tools affect the results. Finally, we only tested the GPT-3.5 model of ChatGPT, and the new billable version (GPT-4) may performs better than this version.

7 Conclusion

In this paper, we did four types of experiments with the ChatGPT model to detect the security vulnerabilities of Python codes. We compared this model with Bandit, Semgrep, and SonarQube which are popular SAST tools for Python codes. We concluded that using the GPT-3.5 model for vulnerability detection of codes in some special manners gives promising results. Specifically, if we use it as a SAST tool assistant, it will produce results that can help to improve the returned results of SAST tools. Overall, we believe this model has the potential to be used in vulnerability detection tasks regarding the factors that may affect the correctness of the results that we described in 6. However, we admit that this study is not general in all aspects and provides primary steps toward this path. In future studies, the behavior of the latest model of ChatGPT (GPT-4) which is more powerful than the GPT-3.5 model, can be examined in vulnerability detection of codes with the hope of obtaining better results. Moreover, the Temperature parameter of the model can be set to values other than zero and innovative rules can be passed to decide for the most efficient obtained results. Another suggestion is to use one-shot learning in future works. Moreover, it should be considered that there is a security caution about using ChatGPT as a SAST tool because it is required to upload the source code on the OpenAI servers.

A Appendix

The distribution of labels of our dataset is provided in Table A.1.

References

[1] Wikipedia. https://en.wikipedia.org/wiki/GitHub, 2023. Accessed: 2023-03-27.

[2] cvedetails. https://www.cvedetails.com/browse-by-date.php, 2023. Accessed: 2015-08-23.

[3] Kumar V, Anjum M, Agarwal V, and Kapur PK. A hybrid approach for evaluation and prioritization of software vulnerabilities. *Predictive Analytics in System Reliability. Cham: Springer International Publishing*, - -:39–51, 2023.

[4] Sharma A. Zhou Y. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

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

[6] Perl H, Dechand S, Smith M, Arp D, Yamaguchi, Rieck K, and et al. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

Table A.1. Details of datasets

Vulnerability	Occurrence in [26]	Occurrence in [27]
CWE-15	15	0
CWE-1004	1	0
CWE-614	1	0
CWE-489	22	0
CWE-20	0	18
CWE-22	3	11
CWE-78	25	5
CWE-79	26	11
CWE-80	0	3
CWE-89	2	6
CWE-90	0	15
CWE-94	0	7
CWE-95	0	2
CWE-99	0	2
CWE-113	0	5
CWE-116	0	7
CWE-117	0	6
CWE-1204	0	3
CWE-193	0	4
CWE-200	0	5
CWE-209	0	4
CWE-215	0	4
CWE-250	0	3
CWE-252	0	2
CWE-259	0	2
CWE-269	0	4
CWE-283	0	2
CWE-284	0	3
CWE-285	0	5
CWE-295	1	8
CWE-297	0	10
CWE-306	0	4
CWE-312	0	3
CWE-319	0	2
CWE-321	0	1
CWE-326	0	4
CWE-327	0	7
CWE-329	0	4
CWE-330	0	1
CWE-331	0	1
CWE-339	0	4
CWE-347	0	3
CWE-352	0	1
CWE-367	0	3
CWE-377	0	3
CWE-379	0	4
CWE-384	0	4
CWE-385	0	6
CWE-400	0	3
CWE-406	0	9
CWE-414	0	7
CWE-425	0	4
CWE-434	0	9
CWE-454	0	6
CWE-462	0	5
CWE-477	0	2
CWE-488	0	4
CWE-502	0	15
CWE-521	0	5
CWE-522	0	12
CWE-595	0	4
CWE-601	0	14
CWE-605	0	4
CWE-611	0	22
CWE-641	0	3
CWE-643	0	8
CWE-703	0	13
CWE-730	0	10
CWE-732	0	4
CWE-759	0	4
CWE-760	0	2
CWE-776	0	3
CWE-798	0	4
CWE-827	0	3
CWE-835	0	5
CWE-841	0	10
CWE-918	0	8
CWE-941	0	8
CWE-943	0	7

[7] Jabeen G, Rahim S, Afzal W, Khan D, Khan A, Hussain Z, and et al. Machine learning techniques for software vulnerability prediction: a comparative study. *Applied Intelligence*, 52, 2022.

[8] Maffeis S. Hanif H. Vulberta: Simplified source code pre-training for vulnerability detection. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2022.

[9] Berabi B, He J, Raychev V, and Vechev M. Tfix: Learning to fix coding errors with a text-to-text transformer. In *Proceedings of the 38th International Conference on Machine Learning; Proceedings of Machine Learning Research: PMLR*, pages 78–91, 2021.

[10] Lorenz Hüther, Bernhard J. Berger, Stefan Edelkamp, Sebastian Eken, Lara Luhrmann, and et al Hendrik Rothe. Machine learning in the context of static application security testing - ml-sast. *Federal Office for Information Security (BSI)*, 2021.

[11] Abdalsamad Keramatfar, Mohadeseh Rafiee, and Hossein Amirkhani. Graph neural networks: a bibliometrics overview. *Machine Learning with Applications*, 10:100401, 2022.

[12] Chakraborty S, Krishna R, and Ding Yand Ray B. Deep learning based vulnerability detection: Are we there yet? In *IEEE Transactions on Software Engineering*, pages 3280–96. IEEE, 2022.

[13] Fu Michael and et al. 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*, pages 935–947, 2022.

[14] et al. Zhou, Yaqin. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systemsy*, 32, 2019.

[15] et al. Ding, Yangruibo. Velvet: a novel ensemble learning approach to automatically locate vulnerable statements. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 959–970. IEEE, 2022.

[16] spectrum. https://spectrum.ieee.org/top-programming-languages-2022, 2022. Accessed:2023-06-23.

[17] et al. Lomio, Francesco. Just-in-time software vulnerability detection: Are we there yet? *Journal of Systems and Software*, - -:111283, 2022.

[18] Rebecca Russell and et al. Kim. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and ap-*

plications (ICMLA), pages 757–762. IEEE, 2018.

[19] Zhen Li and et al. Zou. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[20] Laura Wartschinski and et al. Nollers. Vudenc: Vulnerability detection with deep learning on a natural codebase for python. *Information and Software Technology*, 144:106809, 2022.

[21] et al. Steenhoek, Benjamin. An empirical study of deep learning models for vulnerability detection. *arXiv preprint arXiv:2212.08109*, 2022.

[22] et al. Chen, Yizheng. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. *arXiv preprint arXiv:2304.00409*, 2023.

[23] Hazim Hanif and Sergio Maffeis. Vulberta: Simplified source code pre-training for vulnerability detection. In *International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2022.

[24] Michael Fu and Chakkrit Tantithamthavorn. Linevul: a transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 608–620, 2022.

[25] Pavel Zadorozhny Cheshkov, Anton and Rodion Levichev. Evaluation of chatgpt model for vulnerability detection. *arXiv preprint arXiv:2304.07232*, 2023.

[26] Mohammed Latif Siddiq and Joanna CS Santos. Securityeval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*, pages 29–33, 2022.

[27] Bruno Thalmann Stefan Micheelsen. Pyt: A static analysis tool for detecting security vulnerabilities in python web applications, 2016.

[28] python-security. https://github.com/python-security/pyt/tree/master/examples, 2018. Accessed: 2023-06-15.

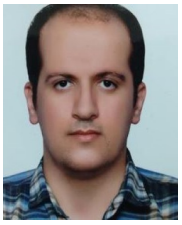
[29] Isa Fulford Andrew Ng. Chatgpt prompt engineering for developers. https://www.deeplearning.ai/short-courses/chatgpt-prompt-engineering-for-developers, April 2023. Accessed: 2023-04-27.

[30] Atieh Bakhshandeh and Zahra Eskandari. An efficient user identification approach based on netflow analysis. In *2018 15th International ISC (Iranian Society of Cryptology) Conference on Information Security and Cryptology (ISCISC)*, pages 1–5. IEEE, 2018.

[31] Catherine Tony, Markus Mutas, Nicolás E Díaz Ferreyra, and Riccardo Scandariato. Llmseceval: A dataset of natural language prompts for security evaluations. *arXiv preprint arXiv:2303.09384*, 2023.

Atieh Bakhshandeh has been a cyber security researcher at RCDAT since 2014. She has a master's degree in Computer Science and her interested research areas include Data Analysis for Security Threat Detection and Penetration Testing.

Abdalsamad Keramatfar received his Ph.D. in Information Technology Engineering in 2021 with a focus on Natural Language Processing and Deep Learning. He worked as a Data Scientist for 6 years at SID and is currently working as an AI researcher at RCDAT.



Amir Norouzi is a Data Scientist with a master's degree in Bioelectric Engineering from Amirkabir University of Technology. He is experienced in Machine Learning, Data Engineering, Deep Learning, and Data Analysis. He has been working in AI since 2017 and is currently working as a researcher in RCDAT.

Mohammad M. Chekidehkhoun graduated from Telecommunications Engineering in 2016 with a focus on Identifying Threats in the Mobile Network. He has been working as a Security Specialist at RCDAT for 12 years, focusing on Penetration Testing and Sode Security Reviews.