Vulnerability - Centric Code Summarization

Arshdeep Singh
Arizona State University
asachd14@asu.edu

Brij Khajuria
Arizona State University
bkhajur1@asu.edu

Chirag Rana
Arizona State University
crana1@asu.edu

Tanuj Jain
Arizona State University
tjain22@au.edu

1 Introduction

There are numerous code repositories available online. However, not all of them have been thoroughly documented. Developers must document them so that other users can understand the meaning of a source code or snippet. There are also code blocks that may be a threat or virus; thus, it is critical to recognize and prevent such vulnerabilities. NLP approaches can help perform the tasks of code fragment documentation and vulnerability detection. Documentation of source codes fragment is essential in software engineering and cybersecurity. It is different from code summarization which is usually one line purpose of a complete source code. So, in this research-based project, we will investigate several approaches and methodologies for utilizing NLP models like GPT-3 and BERT to solve the aforementioned challenge with the help of publicly available repositories.

2 Objective

The project's goal is to describe or document a code snippet and determine whether it poses a hazard to a system. We want to use GPT-3 and pre-trained language models like BERT to perform vulnerability-centric code summarization. This model can be used to check the validity and accountability of a source code, i.e., whether the source code matches the requirements documents and helps identify backdoors and vulnerabilities.

3 Literature Survey

This section discusses previous research and some earlier work on vulnerability-centric code summary.

Software vulnerability detection should be a priority during the development process in order to make software secure and less vulnerable. Neglecting this step can lead to financial losses due to malicious activities. [1] in this paper a taxonomy of machine learning methods for software vulnerability identification is also proposed in this paper.

Additionally, there has been some study on applying deep learning algorithms to automatically find vulnerabilities in source code. VulPecker, is an automated Vulnerability detection using code similarity analysis [2]. This [3] paper highlights the ability of machine learning (ML) to immediately identify software vulnerabilities in source code.

4 Methodology

In this section, we elaborate and explain our methodology and define the steps we took to complete the project as shown in the Figure 1.

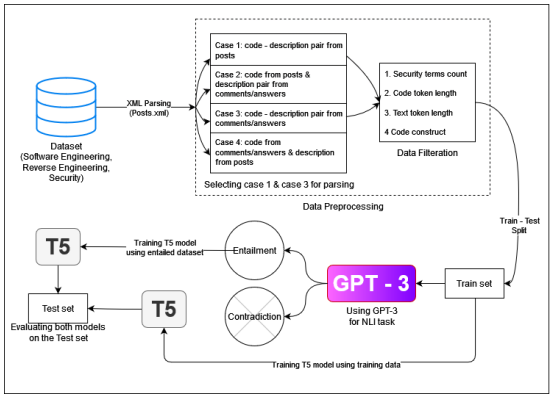

Figure 1: Architecture

The dataset that we took for the project was *StackExchange Data Dump* [4]. We shortlisted three topics related to cybersecurity for vulnerability detection. We used the latest data dumps of software engineering, reverse engineering, and security topics. There was a Posts.xml file in each dump which was parsed using an XML parser. Initially, we also had considered stack overflow data, but it was very huge (approx 98 GB), which made it difficult for us to process it with minimal resources. The final goal of the data preparation was

to get code-description pair so that we can use that data to train our model.

For data preprocessing we prepared a rule-based method to parse the Posts.xml file. We used BeautifulSoup to parse the XML file. In our initial data analysis, we found out that there were four cases of getting code-description pairs. The four cases are:

1. We take code and description pair from the posts field.
2. We take code from the posts and description from the answers/comments field.
3. We take code and description pair from the answers/comments field.
4. We take code from the answers/comments fields and description from the posts.

For our final dataset, we only consider moving forward with cases 1 and 2 as that cases are more relevant. The code-description pair extracted from the same part i.e. posts or answers/comments are more related to each other. Cases 3 and 4 gave us data that were completely irrelevant to each other.

After getting the code-description pair, we filtered the outliers and noise in our data. We filtered our on the basis of code length, description length, code construct and security terms [5] count. To check the code construct we checked if the code snippets contained parenthesis - '()[]' to verify that ¡code¿ tag really consisted of code and not some garbage data. Also, to check if the extracted data was vulnerability centric we used a words dictionary which contained around 450 security related terms and filtered the data having security terms above certain threshold. After the filtration our final data is ready for further tasks.

Next, we divided the code-description pair that made up our final data set into the training and testing sets (80% Train and 20% Test). The T5 model [6] is then trained using the train set for the task of code summarization. Additionally, prompt engineering utilizing the GPT-3 [7] [8], model for the NLI job is done using this same train set (entailment or contradiction). We end up with two models for code summarization after utilizing the data from the GPT-3 NLI job to train a second model using the T5 model. The test set that was initially developed is utilized to evaluate and compare both models.

The evaluation metrics that we have chosen are two folds:

- Rouge metrics.
- Number of vulnerability keywords appearing in the summary (Using a dictionary of cybersecurity terms).

5 Experimentation

In this section we will be discussing about the experimentation that we undertook while implementing our proposed methodology.

We tweaked and experimented with GPT-3 extensively to achieve the desired results. Initially, we used few-shot learning by repeating the preceding instructions for GPT-3 to learn the output format with a few examples. GPT-3, on the other hand, produced output that favored the initial results. Then we used the sliding-window strategy to keep the last few prompts, anticipating it would learn from its previous five results. However, it had the opposite effect, as it began to return incorrect results for all subsequent rows.

Our initial results were unsatisfactory using the Natural Language Inference paradigm of Entailment, Neutral, and Contradiction. Subsequently, binary classification with Entailment and Non-Entailment was attempted with some success, but issues with answer extraction of binary values remained. To further improve the results, various combinations of prompting techniques and GPT-3 hyper-parameters were tried, and the results were extracted using fuzzy string matching of the required vocabulary. The most suitable prompt and hyper-parameters were chosen based on the final results.

Based on the experimentation, this was the final prompt and hyper-parameters that gave the most effective results:
prompt: *"Find weather the following Code and Summary are Entailment or not. Code: [X], Summary: [Y], Entailment(True/False): [Z]"*
temperature: *0.7*
max_tokens: *10*

6 Results

Rouge scores are used to assess the model's performance. One of the tasks was to determine whether GPT-3 could contribute to the model's performance after inferring the relationship of the code and text as entailment and non-entailment.

Metric/Dataset	Entire data		GPT-3 data	
Rouge-1	r	0.032	r	0.020
	p	0.35	p	0.270
	f	0.055	f	0.036
Rouge-2	r	0.003	r	0.002
	p	0.063	p	0.270
	f	0.006	f	0.004
Rouge-l	r	0.031	r	0.020
	p	0.340	p	0.046
	f	0.053	f	0.035

Table 1: Results Table

Table.1 shows the results we obtained after training the T5 model in both scenarios. It can be seen that both models performed poorly in summarizing the code to check for vulnerabilities. However, the model was able to grasp the initial context of the texts, which included describing the author's situation. For example, "I have [X] tasks to complete and am experiencing this problem." As a result, the model did not include any vulnerability keywords in the results. The T5 model gave poor results because of the following results: 1. Non-specific dataset and size. 2. Target length of the text being too large. 3. Diversity in code inputs. 4. Small segment of filtered dataset.

7 Analysis

In general, the model returns a very low rouge-score, and the output becomes over-trained on the token 'code-blk'. One reason is that the output tokens are large in size, and the data set contains a variety of code snippets. When the data is preprocessed to include the position of the code-block from the text, the model outputs "I'm a code-blk", "I have this code-blk and I'm trying to solve it", "I code-blk need help in solving" and so on. If the code-block position token is removed from the data, the model returns a similar answer but with incomplete sentences, i.e "I'm trying to be a service of the following: The problem, but I am working on my code. For example is not sure this." The main reason for this issue is that the t5 model is pretrained on a large English dataset. While using code snippets as input, the model does not receive a specific code format because it contains a variety of code with varying styles and patterns. This could be seen in the analysis, where the dataset do not include a lot of security terms, so the dataset is not specific for code-vulnerability detection.

7.1 GPT-3

Further to debug the problem we used GPT-3 to understand the issue with Language models. GPT-3 was able to understand code snippets and relation of the text considering that its is prompted in an appropriate manner for example:

- Asking whether code and text are related?
- Telling they are related but asking it in what manner they are related?
- Asking whether the code and text are completely irrelevant?

One of the reason that GPT3 was able to understand the manual examples maybe because it has already trained on it and therefore it has its context stored. When prompted on the examples from the recent dataset, the model gave very fuzzy answers and inferences. The model "Codex" is good at code generation given the task as a prompt. As we experimented the prompt with codex, the model gave repetitive answers and also repeated the input prompt.

7.2 T5 model

T5 model was able to understand the starting context of the dataset. We have implemented the model by considering 8 batches and 20 epochs while training and we had run variations on the input token length of 512 and target token length for 256 and 128. Most text started with the contextual meaning having a problem or explaining the situation with the code snippets in their post. This was grasped appropriately by the model but was not able to learn from the meaning in the center of the text due to diversity in the meaning.Problem resolved but caused Overtraining of the token "code-blk".Previously done summarization tasks involves small length target sentence, while reducing the target string could lose on the vulnerability texts, and longer texts affects the performance of the model as shown in the results.

7.3 Dataset

The dataset initially contained 32,046 datapoints, which were further reduced to 2,812 by the data filtering process described in the methodology section (As shown in Figure 2). We tried training the model on the entire dataset before filtering, but the results were the same, with only the initial context of the text providing incomplete answers. The

majority of the dataset was filtered based on security keywords after a thorough search. The code was made up of various code languages and bash scripts, resulting in a diverse set of tokens. Because stack-dumps were mostly made up of questions that people had while not working on their code, the output was also similar in meaning.

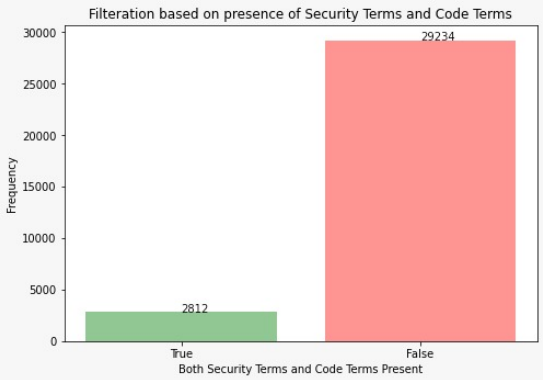

Figure 2: Training Dataset

8 Challenges

Here are few challenges that we faced while working on the project.

8.1 Data Extraction

The text in the data set was not properly structured. We had to divide the text in four different cases as mentioned in section 4. One of the topic that we selected for our data was Stack Overflow which we had to discard because it was almost 98 GB in size which we were not able to process. The extracted data had very less number of vulnerability centric keywords in them, which resulted in very less amount of data with vulnerability centric summaries of code snippets. These were the challenges related to data extraction part.

8.2 GPT-3

While using GPT-3 we had few challenges. We faced issues with demo account of GPT-3 that we were limited by 60 requests per minute, which extended the time taken for the output generation using prompts for our data set. In the extracted code and summaries of the data set, there were large number of rows with token length of more than 2048, which is limited by GPT-3 capability. This prevented us to analyze all the data set, as we had to drop these rows. The output generated by GPT-3 was in natural language and not according to our requirements, so we had to filter and extract the required information of Entailment or not. This was challenging task because GPT-3 is unreliable to produce consistent responses. Also, GPT-3 is not trained on this kind of data to find the Inference on such long code-summary pairs. It classified many such pairs to be entailing, but were not.

8.3 T5

GPU was a must to train the model and we were having several CUDA Out of Memory error in between the training even after resolving and clearing cache based on the torch documentation. We also faced CUDA out of memory error in the 'run_summarization.py' script provided by the hugging face community in an inbuilt function of trainer which is hard to resolve. Google Colab has a certain time limit for GPU usage so we used our offline GPU for training. Tokenizer splits the code, larger integral values and hashes to inappropriate tokens based on its vocabulary knowledge which caused inarticulate tokens passed to the model.

9 Future Work

We can experiment with other types of filtering out vulnerability-centric code, as well as mix and match by combining two approaches to see if those get better results in the future. Also we can use different machine learning techniques to extract relevant summary, as currently we assuming that the answer provided in the post is complete explanation for the given code but it is not the case in most of the cases. Also we can use a classifier to filter out non-relevant codes which are natural text and not part of the code.

10 Conclusion

This project has concluded that GPT-3 is a useful tool for natural language inference tasks, but its performance is limited when it comes to larger datasets. The T5 model that was trained on the dataset also showed signs of overfitting due to the presence of the 'code-blk' token. Additionally, the dataset did not have many vulnerability-centric keywords, which likely prevented the model from providing reliable results. This project highlights the importance of having reliable datasets and the need to properly train models in order to obtain maximum performance.

Appendix: Individual Contribution

Contribution of each member in the project.

Arshdeep Singh Sachdeva

Security data extraction and pre-processing, GPT-3 integration and prompt engineering, Result Visualization and Analysis.

Brij Bhushan Khajuria

Architecture, Parser Development, Code-constructs detection, Prompt Development, Metadata extraction.

Chirag Rana

Software Engineering data extraction, coded T5 model training and 4 cases of data extraction, GPT-3 Prompt experiments and analysis.

Tanuj Jain

Reverse engineering data extraction and pre-processing. Designing methodology and GPT-3 prompt engineering.

References

[1] Hazim Hanif et al. "The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches". In: *Journal of Network and Computer Applications* 179 (2021), p. 103009. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2021.103009. URL: https://www.sciencedirect.com/science/article/pii/S1084804521000369.

[2] Zhen Li et al. "VulPecker". In: *Proceedings of the 32nd Annual Conference on Computer Security Applications - ACSAC '16* (2016). DOI: 10.1145/2991079.2991102.

[3] Rebecca Russell et al. "Automated Vulnerability Detection in Source Code Using Deep Representation Learning". In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 2018, pp. 757–762. DOI: 10.1109/ICMLA.2018.00120.

[4] Stack Exchange Community. *Stack Exchange Data Dump 2022-03-07 : Stack Exchange, inc. : Free Download, borrow, and streaming*. Mar. 2022. URL: https://archive.org/details/stackexchange_20220307.

[5] SANS. *SANS Institute*. 2019. URL: https://www.sans.org/security-resources/glossary-of-terms/.

[6] Hugging Face. *T5*. URL: https://huggingface.co/docs/transformers/model_doc/t5.

[7] Shuohang Wang et al. "Want To Reduce Labeling Cost? GPT-3 Can Help". In: *arXiv:2108.13487 [cs]* (Aug. 2021). URL: https://arxiv.org/abs/2108.13487.

[8] Tom Kwiatkowski et al. "Natural Questions: A Benchmark for Question Answering Research". In: *Transactions of the Association for Computational Linguistics* 7 (Aug. 2019), pp. 453–466. ISSN: 2307-387X. DOI: 10.1162/tacl_a_00276. eprint: https://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00276/1923288/tacl_a_00276.pdf. URL: https://doi.org/10.1162/tacl%5C_a%5C_00276.