Research on Code Vulnerability Detection Using RoBERTa, CodeBERT and ReGVD Models

by

Luo Hao

1155204657

A report
submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
The Chinese University of Hong Kong

April, 2024

Supervisor: Prof.Wei MENG

Abstract

Automated detection of software vulnerabilities stands as a cornerstone in software security, and recent advancements in Deep Learning (DL) have sparked considerable interest in leveraging these models for this purpose. Numerous studies have showcased the promising potential of state-of-the-art DL models, such as VulBERTa, CodeBERT, and ReGVD, in achieving remarkable accuracy in vulnerability detection. In this paper, I delved into the performance of these models by asking: " how well do the state-of-the-art models perform?" To address this, I conducted a survey and reproduced experiments on two widely utilized vulnerability detection datasets: Devign and Big-Vul. Through investigating 5 research questions spanning model capabilities and dataset characteristics, I systematically evaluated the performance and stability of the models. Notably, I explored code features that are hard for models to handle. I examined whether model performance varies across different projects or different Common Weakness Enumeration (CWE) types. These insights aim to enhance our understanding of model behavior, offer guidance on data preparation, and bolster the resilience of vulnerability detection models. The datasets, code, and results are available at https://github.com/rojic12138/Code-Vulnerability-Detection-Research.

Keywords

Code Vulnerability Detection, RoBERTa, CodeBERT, ReGVD

1. INTRODUCTION

Automated detection of security vulnerabilities represents a foundational challenge in systems security. Conventional techniques often suffer from elevated rates of false positives and false negatives [1]. For instance, static analysis tools frequently yield high false positives, erroneously flagging non-vulnerable instances as vulnerable. Conversely, dynamic analysis tends to encounter high false negatives, failing to identify many vulnerabilities. Despite extensive endeavors, these tools remain unreliable, imposing substantial manual overhead on developers.

Deep learning (DL) is emerging in the software engineering research community. An increasing number of methods are being developed on top of DL models, also due to the unprecedented amount of software-related data accessible for model training purposes [2].

Among all these DL models, BERT-based models excel as they have a deeper understanding of the context. BERT [3] is a recently introduced technology that gives us state-of-the-art results in many NLP tasks. Its main outline is the creation of transformers for bidirectional training. When trained bidirectionally, the model has a deeper understanding of the context of the language.

Graph Neural Network (GNN) models are popular as well. They capture not only the tokens of the code, but also its control flow and data flow. GNN models typically use Abstract Syntax Trees (AST) and Code Property Graphs (CPG) to build graphs. Notable examples include LineVD [4].

A broad spectrum of machine learning-based vulnerability prediction studies has emerged. Alejandro Mazuera-Rozo [5] conducted a comprehensive empirical analysis, revealing that current DL techniques still lack reliability in detecting software vulnerabilities, with shallow learning classifiers serving as competitive alternatives. Additionally, Saikat Chakraborty's [6] systematic investigation identified challenges in DL-based vulnerability prediction, including issues with training data (e.g., data duplication, imbalanced class distribution) and model selection (e.g., simplistic token-based models), leading to suboptimal feature learning.

This paper undertakes a survey and reproduction of 3 state-of-the-art (SOTA) deep learning models for vulnerability detection, namely RoBERTa [7], CodeBERT [8] and ReGVD [9]. Through the formulation of 5 research questions and corresponding studies, the aim is to comprehensively understand these models. I also hope to provide valuable insights and guidelines to enhance the design of both models and datasets in the field.

I categorize these 5 research questions into 2 areas: model capabilities and datasets. Firstly, my objective is to comprehend the effectiveness of deep learning in addressing vulnerability detection challenges, particularly focusing on the following research questions:

- **RQ1**. How do these models perform on the vulnerability detection dataset?
- **RQ2**. Do models agree on the test results? Do the models' performances vary when the seed is changed?

Secondly, I concentrate on the training data, aiming to discern its impact on model performance. Specifically, I formulate the following research questions:

- **RQ3.** Do certain code features make it more difficult for current models to predict correctly?
- **RQ4.** Do models perform better when the training/validation/test datasets are from a single project? Do projects indeed represent different distributions?
- **RQ5.** Do the models perform better when they only handle a specific CWE type?

To address the research questions, I conducted a comprehensive examination of state-of-the-art DL models and effectively replicated 3 models: RoBERTa, CodeBERT, and ReGVD. Notably, ReGVD stands out as a GNN-based model and other 2 models are BERT-based models. To facilitate a comparative analysis, I successfully configured the models to operate with 2 widely used datasets, namely Devign [10] and Big-Vul [11].

Before continuing, let me finish the remaining work from last semester:

- "I aim to enhance RoBERTa's general code representation capability by extending the training time and increasing the model size." However, I found that the model was already overfitted, so extending the training time did not help improve performance. Besides, re-training a model with a larger model size does not provide insights.
- "I will make the model more practical and provide more convenient interface. " Yes, I have uploaded the RoBERTa model trained with Devign to huggingface. The link is https://huggingface.co/Rojic/VulRoBERTa.

In summary, my contributions in this paper are:

(1) I reproduced 3 SOTA DL-based models, namely RoBERTa, CodeBERT and ReGVD and obtained their performance on Devign and Big-Vul datasets.
(2) I designed 5 research questions to elucidate both the capabilities of the models and the influence of datasets.
(3) I open-sourced all the code and data that are used in this project for wider dissemination. They are accessible at https://github.com/rojic12138/Code-Vulnerability-Detection-Research.

2. DATASET

In this section, I outline two datasets: Devign and Big-Vul for comparing model performance. I will skip discussing the Draper dataset for pre-training since it was already mentioned in Semester 1. All datasets consist of function-level C/C++ open-source code.

I chose these two datasets because both feature real-world projects and vulnerabilities, and they are commonly used for evaluating and fine-tuning most SOTA models. Here's a detailed introduction to each:

(1) Devign. This dataset comprises four widely used open-source libraries: Linux, FFmpeg, Qemu, and Wireshark. By intentionally including projects with varied functionalities, the dataset enhances its robustness and applicability. It captures a broad spectrum of potential vulnerabilities, contributing to a comprehensive analysis.
(2) Big-Vul. The authors conducted a comprehensive crawling, focusing on the public Common Vulnerabilities and Exposures (CVE) database and associated source code repositories from Github projects. This effort yielded descriptive details related to vulnerabilities, including CVE IDs and severity. In semester 1, I utilized Big-Vul to pre-train the RoBERTa. However, in this semester, Big-Vul serves as the dataset for fine-tuning all three models, because ReGVD does not require pre-training.

Additionally, I utilized the DiverseVul [12] dataset to investigate whether a model performs better when it only has to deal with specific CWE types. The authors of DiverseVul meticulously created the dataset by carefully crawling websites with security concerns. They extracted bug fix submissions and source code from the corresponding projects, resulting in a comprehensive and diverse collection of vulnerabilities.

3. ROBERTA

In this section, I briefly provide an overview of RoBERTa, which comprises four key components: tokenization, pre-training, fine-tuning, and evaluation. While I won't delve into the architecture details, as they were covered in Semester 1, I will emphasize aspects of the pre-training process.

During pre-training, I trained a standard RoBERTa model using the Masked Language Modeling (MLM) objective. The goal of pre-training is to acquire a comprehensive and informative general representation of C/C++ code across various software projects.

MLM is a self-supervised learning process. It involves masking a certain percentage of words in the input text and training the model to predict the masked words based on the context provided by the non-masked words.

It is notable that the pre-training of RoBERTa completely discards the Next Sentence Prediction (NSP) task, which is a component of the pre-training process in BERT. NSP involves a binary classification task where the model predicts whether two segments of text follow each other in the original text. Positive examples are generated by taking consecutive sentences from the text corpus, while negative examples are created by pairing segments from different documents.

Combining various software projects can enhance the learning process by broadening the representation knowledge of the code across different coding styles [2]. Hence, I pre-trained a RoBERTa model on Draper dataset [13], which is sourced from the Debian distribution and public repositories on Github. The embedding size was set to 768 dimensions, consistent with RoBERTa-base.

4. CODEBERT

CodeBERT is a bimodal pre-trained model designed to handle both programming language (PL) and natural language (NL). It acquires generalized representations that facilitate various downstream NL-PL applications, including natural language code search and code documentation generation. Notably, CodeBERT stands out as the first large NL-PL pretrained model for multiple programming languages.

The input of CodeBERT consists of two concatenated segments separated by a special token, denoted as [CLS];w_1;w_2;…;w_n; [SEP]; c_1; c_2;…; c_m; [EOS]. The first segment represents natural language text, while the second contains code in a specific programming language. Preceding the two segments is the [CLS] token, serving as a marker. Its final hidden representation serves as the aggregated sequence representation for classification or generation.

CodeBERT utilizes the same architecture as RoBERTa. It is developed with the identical model architecture as RoBERTa-base, with 125M parameters, which aligns with my RoBERTa model.

The difference between CodeBERT and RoBERTa lies in their pre-training methods. CodeBERT employs two objectives during pre-training.

The first objective is MLM on bimodal data of NL-PL pairs, which deviates slightly from RoBERTa's approach. Given a data point represented by an NL-PL pair (x = {w, c}), where w denotes a sequence of NL words and c represents a sequence of PL tokens, CodeBERT randomly selects positions within both the NL and PL sequences to mask out. Then, it replaces these positions with a special [MASK] token. The goal of the MLM objective is to predict the original tokens that were masked out.

The second objective is Replaced Token Detection (RTD) , which further leverages a substantial amount of unimodal data. RTD operates akin to a Generative Adversarial Network (GAN), comprising two data generators: an NL generator and a PL generator, each responsible for producing plausible alternatives for randomly masked positions. A discriminator is trained to discern whether a word is original or replaced. To implement this, two efficient n-gram language models with bidirectional contexts are employed—one for NL and one for PL—trained on corresponding unimodal data points.

5. REGVD

In contrast to RoBERTa and CodeBERT, ReGVD adopts a GNN architecture. Its simplicity makes ReGVD a versatile, practical, and programming language-independent model capable of operating across diverse source codebases and libraries effortlessly.

ReGVD treats each raw source code as a sequence of tokens and employs two methods to create a graph structure:

(1) Unique token-focused construction. It means representing unique tokens as nodes and establishing edges between them based on their co-occurrence within a fixed-size sliding window.

(2) Index-focused construction. It means representing all tokens as the nodes and establishing edges in the same method.

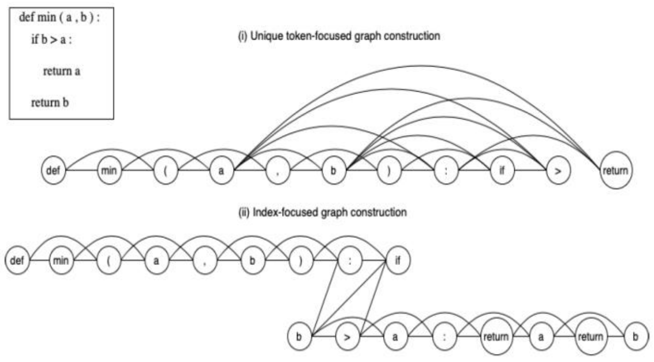
def min (a , b) :
if b > a :
return a
return b
(i) Unique token-focused graph construction
(ii) Index-focused graph construction

Figure 1 An illustration for two graph construction methods with a fixed-size sliding window of length 3, extracted from [9].

Figure 1 illustrates the process of constructing the graph. Through experiments, the paper concludes that the unique token-focused construction has better performance.

The node features are initialized using the token embedding layer of a pre-trained programming language model, RoBERTa, chosen intentionally for this project. Additionally, the tokenizer is also sourced from RoBERTa.

Then, ReGVD employs GNNs, such as Graph Convolutional Networks (GCNs) [14] or Gated GNNs [15], while incorporating residual connections among GNN layers for enhanced performance. The model structure is depicted in Figure 2.

Following GNN processing, ReGVD explores a combination of sum and max pooling to generate a graph embedding for the source code. This embedding is then passed through a single fully-connected layer, followed by a softmax layer, to predict code vulnerabilities.

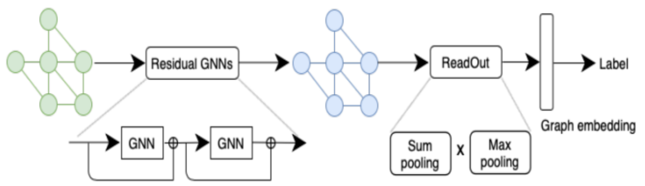

Figure 2 An illustration for ReGVD, extracted from [9].

6. Research Questions

I proposed 5 research questions divided into two categories: model capabilities and dataset. Each RQ is accompanied by its motivation, study methodology, and my observations.

A. Model Capabilities

RQ1 How do these models perform on the vulnerability detection dataset?

The motivation behind this question stems from the realization that the reported performance of models in papers may not always reflect their actual performance. To address this, I reproduced 3 models and evaluated their performance on the Devign and Big-Vul datasets. In Table 1, I present the test results averaged over three seeds, with the highest value highlighted for each metric.

Table 1 Models performance

Dataset	Model	Acc (%)	Prec (%)	Recall (%)	F1 (%)
Devign	RoBERTa	59.5	56.2	54.4	55.3
	CodeBERT	**63.5**	60.6	**57.4**	**58.9**
	ReGVD	62.0	**61.9**	43.5	51.1
Big-Vul	RoBERTa	90.4	60.9	**12.5**	**20.7**
	CodeBERT	94.6	66.2	11.9	20.2
	ReGVD	**94.7**	**72.6**	11.7	20.2

Table 1 reveals that CodeBERT stands out as the top performer in the Devign dataset, while all three models demonstrate high accuracy and low F1 scores in the Big-Vul dataset. This is mainly attributed to the dataset's highly unbalanced nature, where the majority of functions are non-vulnerable.

RQ2 Do models agree on the test results? Do the models' performances vary when the seed is changed?

Deep Learning model performance can vary with different random seeds. To explore the extent of this variability, I trained the models using three random seeds on identical train/valid/test partitions of the

Devign dataset. Table 2 presents the variability and standard deviations of F1 scores across various models.

Table 2 Variability over 3 seeds on Devign dataset

Model	Stable Proportion (%)	F1_std (%)
RoBERTa	65.6	1.05
CodeBERT	63.7	1.67
ReGVD	74.1	2.52

Table 2 illustrates that, on average, the models yield the same prediction for approximately 67.8% of test functions across three different seeds. Interestingly, ReGVD exhibits the least variability compared to the two BERT-based models. Moreover, the standard deviations of F1 scores are very small for all three models. This indicates that while a model may provide differing predictions for individual test functions, its overall performance remains stable and reliable.

B. dataset

RQ3 Do certain code features make it more difficult for current models to predict correctly?

Exploring the characteristics of code that pose challenges for deep learning (DL) models is crucial for identifying areas for improvement. It provides valuable insights into where our efforts should be focused to enhance future performance.

I selected 6 key words from C programming language as potential challenging code features: "for," "goto," "if," "jump," "switch," and "while". Next, I counted the occurrences of each of these feature words within the first 1024 tokens excluding comments of a test function, as DL models typically truncate the remaining tokens. Subsequently, I computed the average accuracy across three DL models with three random seeds for each test function. Table 3 illustrates an example of the intermediate calculation results.

Following this, I employed a multivariate linear regression model to establish the relationship between the code features (counts of the six key words) and the averaged accuracy. High averaged accuracy indicates that the code is easier to classify, whereas low accuracy suggests the opposite. The regression yielded the following results:

$$\begin{aligned} \mathit{Average\ accuracy\ percentage} = \\ 58.72 - 0.43 * \mathit{for} + 0.21 * \mathit{goto} - 0.16 * \mathit{if} \\ + 4.72 * \mathit{jump} + 0.58 * \mathit{switch} + 0.54 * \mathit{while} \end{aligned}$$

A positive coefficient for a code feature (e.g., 0.21 for "goto") indicates that the more frequently the feature appears, the easier it is for DL models to classify whether the code is vulnerable. Conversely, a negative coefficient (e.g., -0.43 for "for") suggests that the more frequently the feature appears, the harder it is for DL models to classify.

The result is surprising, as the coefficients for "goto" and "jump" are both positive, indicating their usefulness for DL models. This contradicts intuition, as these statements typically alter the control flow of C programs, posing challenges for security analysis tools.

Limit: The linear regression may lack credibility, because sklearn.linear_model.LinearRegression does not provide confidence intervals.

RQ4 Do models perform better when the training/validation/test datasets are from a single project? Do projects indeed represent different distributions?

Table 3. RQ3: An example of the intermediate calculation results.

function	label	for_ count	goto_ count	if_ count	jump_ count	switch_ count	while_ count	Average_ accuracy(%)
static int xen_9pfs_connect(struct XenDevice *xendev){ …	1(vulnerable)	1	1	3	0	0	0	22.2
aio_write_f(int argc, char **argv){ char *p; int count = 0; …	0(not vulnerable)	1	0	5	0	1	1	88.9

As the old saying goes, stones from other mountains can attack jade. I am curious to explore whether the diversity of projects in the training dataset enhances the model's performance in specific projects. Conversely, if different projects indeed represent different distributions, the model will perform better when the training/validation dataset and test dataset come from the same project.

To test this hypothesis, I extracted 9k functions of qemu project and 9K functions of FFmpeg from Devign dataset. Subsequently, I divided these functions into training, validation, and test sets using an 80/10/10 ratio.

I designed four cross experiments for this question:

(1) I trained a CodeBERT model with training/validation dataset of qemu and examine this model's performance on qemu test dataset.
(2) I used this trained CodeBERT to examine its performance on FFmpeg test dataset.
(3) I trained a CodeBERT model with training/validation dataset of FFmpeg and examine this model's performance on FFmpeg test dataset.
(4) I used this trained CodeBERT to examine its performance on qemu test dataset.

Table 3 Cross experiment result

Training/ valid dataset	Test dataset	Accuracy	Precision	Recall	F1
Qemu	qemu	61.7	68.3	47.2	55.8
Qemu	FFmpeg	56.7	58.5	24.5	34.6
FFmpeg	FFmpeg	59.8	56.0	64.3	59.9
FFmpeg	qemu	49.4	51.3	26.2	35.7

Table 4 indicates that the model achieves the highest F1 score when the training/validation/test data are from the same project, supporting the hypothesis that different projects represent different distributions. Additionally, it's noteworthy that the accuracy in the second scenario is higher than in the third. This may suggest that DL models can learn more effectively from the qemu project than from FFmpeg.

RQ5 Do the models perform better when they only handle a specific CWE type?

This question parallels RQ4 but centers on CWE types, which signify the categories of vulnerabilities. In the previous semester, I investigated RoBERTa's performance in DiverseVul and its four types. For each CWE type, I extracted all functions labeled with that specific CWE type as positive samples. Then, I randomly selected negative samples whose number is three times the number of positive samples. Negative samples are probably labeled with other CWE types. The findings suggests that a vulnerability detection model might exhibit superior performance when exclusively dealing with a specific CWE type.

Here, I replicated the experiment using ReGVD. The results are presented in Table 5.

Table 4 The evaluation results on DiverseVul and its subdatasets

Dataset	F1(%)
DiverseVul	62.5
DiverseVul-20	66.2
DiverseVul-119	80.0
DiverseVul-125	71.9
DiverseVul-787	67.4

Table 5 displays the evaluation outcomes of DiverseVul and its sub-datasets employing ReGVD. "DiverseVul-20" denotes the sub-dataset encompasses all functions labeled with CWE-20. F1 scores utilizing the sub-dataset exhibit varying degrees of improvement compared to results utilizing the entire dataset. The conclusion remains consistent: a vulnerability detection model might achieve better performance when focused on a specific CWE type.

7. LIMITS

Due to constraints in computing resources and time, some exploratory experiments did not include all models and data, such as in RQ4 where only Qemu and Ffmpeg from the Devign dataset were utilized to train CodeBERT. Nonetheless, the methodology has been clearly presented.

The exploration of what DL models learn and how they behave remains an attractive endeavor, unsolved in this paper. Tools like the Learning Interpretability Tool (LIT) [16] offer valuable insights into the tokens in code that influence a model's decision-making process, providing a visual and interactive platform for understanding DL model behavior.

8. CONCLUSION

To gain deeper insights into deep learning vulnerability detection models, I undertook the reproduction of three DL models: RoBERTa, CodeBERT, and Devign. This empirical study was conducted using two datasets, namely Devign and Big-Vul, and focused on addressing 5 research questions:

(1) I presented the averaged test results of these 3 models across 3 different seeds.

(2) The models, on average, yield the same prediction for approximately 67.8% of test functions across 3 seeds. Besides, the standard deviations of F1 scores across 3 seeds are very small for all three models.

(3) I constructed a linear regression model aimed at identifying code features (C keywords) that pose challenges for accurate predictions by the model.

(4) I found that the model's performance is enhanced when the training/validation dataset and test dataset come from the same project. This supports the hypothesis that different projects represent different distributions.

(5) I found that a vulnerability detection model may perform better when it only needs to deal with a specific CWE type.

My experimental findings suggest significant opportunities for enhancing automated vulnerability detection systems, particularly in addressing issues like low accuracy and limited interpretability.

References

[1] Neuhaus, S., Zimmermann, T., Holler, C., & Zeller, A. (2007, October). Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security* (pp. 529-540).

[2] Zeng, P., Lin, G., Pan, L., Tai, Y., & Zhang, J. (2020). Software vulnerability analysis and discovery using deep learning techniques: A survey. *IEEE Access*, *8*, 197158-197172.

[3] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

[4] Hin, D., Kan, A., Chen, H., & Babar, M. A. (2022, May). LineVD: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories* (pp. 596-607).

[5] Mazuera-Rozo, A., Mojica-Hanke, A., Linares-Vásquez, M., & Bavota, G. (2021, May). Shallow or deep? an empirical study on detecting vulnerabilities using deep learning. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)* (pp. 276-287). IEEE.

[6] Chakraborty, S., Krishna, R., Ding, Y., & Ray, B. (2021). Deep learning based vulnerability detection: Are we there yet?. *IEEE Transactions on Software Engineering*, *48*(9), 3280-3296.

[7] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., ... & Stoyanov, V. (2019). Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

[8] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

[9] Nguyen, V. A., Nguyen, D. Q., Nguyen, V., Le, T., Tran, Q. H., & Phung, D. (2022, May). Regvd: Revisiting graph neural networks for vulnerability detection. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (pp. 178-182).

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

[11] Fan, J., Li, Y., Wang, S., & Nguyen, T. N. (2020, June). AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories* (pp. 508-512).

[12] Chen, Y., Ding, Z., Alowain, L., Chen, X., & Wagner, D. (2023, October). Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses* (pp. 654-668).

[13] Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., ... & McConley, M. (2018, December). Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)* (pp. 757-762). IEEE.

[14] Zhang, S., Tong, H., Xu, J., & Maciejewski, R. (2019). Graph convolutional networks: a comprehensive review. *Computational Social Networks*, *6*(1), 1-23.

[15] Li, Y., Tarlow, D., Brockschmidt, M., & Zemel, R. (2015). Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*.

[16] Tenney, I., Wexler, J., Bastings, J., Bolukbasi, T., Coenen, A., Gehrmann, S., ... & Yuan, A. (2020). The language interpretability tool: Extensible, interactive visualizations and analysis for NLP models. *arXiv preprint arXiv:2008.05122*.