Lessons from Developing Multimodal Models with Code and Developer Interactions

Nicholas Botzer
University of Notre Dame
nbotzer@nd.edu

Sameera Horawalavithana
Pacific Northwest National Laboratory
sameera.horawalavithana@pnnl.gov

Tim Weninger
University of Notre Dame
tweninger@nd.edu

Svitlana Volkova
Pacific Northwest National Laboratory
svitlana.volkova@pnnl.gov

Abstract

Recent advances in natural language processing has seen the rise of language models trained on code. Of great interest is the ability of these models to find and classify defects in existing code bases. These models have been applied to defect detection but improvements between these models have been minor. Literature from cyber security highlights how developer behaviors are often the cause of these defects. In this work we propose to approach the defect detection problem in a multimodal manner using weakly-aligned code and developer workflow data. We find that models trained on code and developer interactions tend to overfit and do not generalize because of weak-alignment between the code and developer workflow data.

1 Introduction

Automatic detection and identification of software vulnerabilities is important to improve code security and prevent exploits. This is especially important with open source software that is being used in many safety critical systems. Work has been done for some time to attempt this task with a variety of methods in machine learning [17, 13]. Recent progress in deep learning using foundation models has led to a new class of pre-trained natural language models on code. These models have shown a remarkable ability to understand and generate code. Although results have been extremely promising in this direction, there is still much work to be done. These foundation models for code have quickly become a major area of development. Early successes such as the Codex model [6] and AlphaCode [14] captured the attention of the community and quickly turned into commercial products via Github Copilot. Despite this early success, these models are far from perfect; researchers continue to seek innovative ways to improve performance. GraphCodeBERT is one such example that moves beyond the straightforward modeling token sequences and additionally includes the abstract syntax tree of the code during the pre-training [11]. InCoder [7] is another model that provides the ability to fill-in missing segments of code, much like a developer would do. The field has also taken into consideration models that can handle both encoding and decoding tasks with the development of PLBART [2] and CodeT5 [21].

These models are quickly being adopted by companies and developers to help increase productivity, with little regard to the security implications of these models. Due to their large scale adoption, ongoing work by Asare et al. seeks to investigate this problem of insecure code generation from these models [3]. Additional work by Pearce et al. has already investigated this to determine the various type of Computer Weakness Enumerations (CWE) and has found that under certain conditions, certain

I Can't Believe It's Not Better Workshop at NeurIPS 2022.

models generate insecure code approximately 40% of the time [18]. Based on this work, Pearce et al. have that experimented with using these large language models to repair code vulnerabilities [19].

These recent advances have made significant progress in the area of detecting and correcting software vulnerabilities. Multiple models have been developed [20, 5, 12] based on foundation models for defect detction but are still lacking, achieving just 66% accuracy with the state of the art method.[1] However, there remain many unsolved issues. Studies from cybersecurity have found that certain developer habits, such as copy and pasting insecure code, are among the primary drivers in the creation of software vulnerabilities [1]. For example, Benedetti et al. showed that Github workflows are exploitable by bad actors that can inject vulnerabilities into a projects code [4]. Such developer behavior data has also been applied to detect malicious users [9] as well as malicious repositories on Github [24].

In the present work, we endeavored to model developer behavior from Github logs and use this data to supplement existing software vulnerability detection. In other words, given a git-log of developer behavior (e.g., commits, pushes, pulls) as well as the source code base of a repository, can we detect if the source code contains a vulnerability or defect? This appeared to be a natural next-step because: (1) deep learning models have shown remarkable success at understanding and generating source code, and (2) machine learning models of human behavior have shown to be useful in finding vulnerabilities in source code.

However, we find that the joint source code and user-behavior model performs substantially worse at finding source code defects than the independent versions of models. Additional exploration of these models found that the primary culprit behind this decreased performance was overfitting on the user-behavior data.

In the present work, we describe the independent models as well as our new joint model. We also perform a litany of experiments to show the overfitting problem. Finally, we conclude with remarks on possible future work that might serve to ameliorate these problems.

2 Multimodal Model Architecture

Our goal was to utilize developer workflow data along with a pre-trained language model. Recent work in multimodal methods have shown strong performance in a variety of tasks when using data from more than one modality. For our methodology we utilized the recently released GreaseLM model architecture [23], that jointly models language and a knowledge graph together. Our experimental hypothesis was that we could utilize developer workflow data, in the form of a knowledge graph, with a language model, and achieve greater performance at detecting defects.

The core concept behind GreaseLM is that the model mixes information from language model (LM) and Graph Neural Network (GNN) layers. This information fusion allows the model to share the features from both domains and create a new mixed space of information. The original authors used this model for the task of Question-Answering, but the method can be used for any encoding task by modifying the model head. The choice of pre-trained LM and GNN models used within GreaseLM will vary depending on the task. GreaseLM requires two inputs during task-supervised training, i) tokenized text representing the language, and ii) the corresponding subgraph extracted from an external knowledge graph (KG). While the text inputs are fed to the LM layers, the extracted subgraphs are fed to the GreaseLM layers together with the output of LM layers (Figure 1). For more details on the GreaseLM model please see the original paper [23].

3 Experimental Setup

To perform our analysis we utilized the defect detection dataset from CodeXGlue [16], that was originally taken from Devign [25]. This dataset consists of functions of C code that may contain potential vulnerabilities that could be exploited. Each of the examples in the dataset is taken from an open source repository on Github and includes the commit id that added the function of code. Defect detection is a binary classification task to determine whether a given function of code has defect or not. A breakdown of the dataset statistics can be seen in Table 1. Evaluation on this dataset is useful for us as many other baselines have adapted it as a benchmark.

[1] Leaderboard for the defect detection task https://microsoft.github.io/CodeXGLUE/

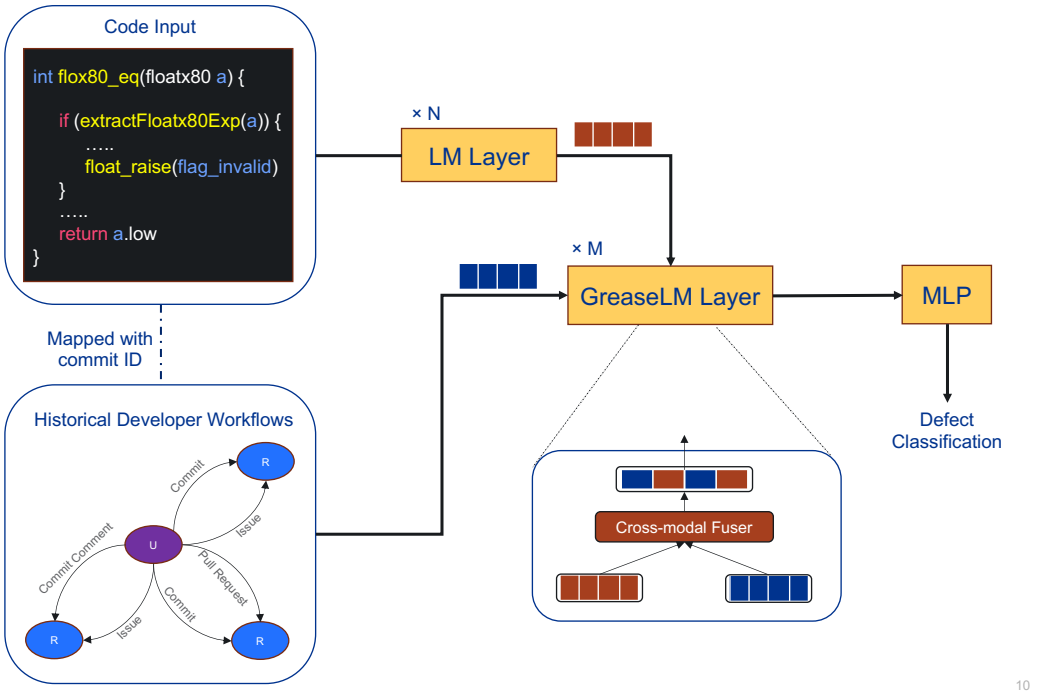

Figure 1: Overview of the setup for using GreaseLM for the defect detection task. In the developer wokflow box blue nodes represent repositories and purple a user. The language model input is the function of code that may or may not contain a defect. The developer workflow subgraph is extracted from the Github KG based on the commit id for each particular function.

GreaseLM requires the use of a pre-trained language model to work. Our choice of pre-trained language model was a RoBERTa [15] model that we pre-trained on code. We collected the code via the data collection script used by ThePile [8], that extracts code from Github repositories. Our collection followed the same setup extracting only those repositories that have one hundred stars or more. We utilized the extracted code from the eleven most prominent programming languages that we collected.

As stated before GreaseLM also takes input in the form of subgraphs extracted from a knowledge graph. To model our knowledge graph from Github interaction data, we utilized the available GHtorrent dataset [10]. GHtorrent makes available Github's relational schema for each repository and all of events associated with the repository. As we wished to model developer behavior, we decided our KG would contain two node types, repository nodes and user nodes. The relations that we defined for our edges consisted of commits, pull requests, issues, commit comments, pull request comments, and issue comments. We hoped this setup would allow us to capture developer patterns based on how often they perform these actions and improve performance on defect detection.

Subgraph extraction In the original GreaseLM work entity linking is used for each QA pair to obtain the initial nodes in the KG. The authors then performed a two-hop neighborhood expansion to build a larger subgraph. As mentioned previously we utilize the commit id in each defect detection example to map to our Github KG. This allows us to extract the user and repository as the initial seed nodes to sample our subgraph. The initial user and repository nodes are connected with a commit type edge, along with the timestamp of when the commit occurred. Then, we perform a one-hop neighborhood expansion from the user node to get all of the repositories that the user has had interactions with. We limit this subgraph extraction for the edges that occurred before the timestamp of the commit used to seed the graph. This gave us a subgraph that contains all of the repositories, along with the various edge types connecting them, that the user had interacted with prior to the commit timestamp. In some cases this results in large subgraphs, so we limited the size of the subgraphs to include only 50 nodes. We selected the nodes to keep based on those that had the most recent user interactions.

Table 1: Breakdown of the various splits of the defect detection dataset we used for experiments. N stands for target class not having a defect and D stands for containing a defect.

Data Split	Original		Temporal		User		Repo	
	N	D	N	D	N	D	N	D
Train	11,836	10,018	12,184	9,670	12,951	10,486	9,151	8,186
Dev	1,545	1,187	1,282	1,450	1,028	912	2,892	2,098
Test	1,477	1,255	1,392	1,340	879	1,062	2,815	2,176

Data Splitting While the defect detection dataset provides already curated train, dev, and test splits, we found it necessary to re-split the dataset to prevent data leakage. In the original dataset some of the commit ids have examples that appear in the train, dev, and test split. Each example will have a different function of code associated with it, but all of the target classes are the same. This causes a group leakage issue for our setup, because we extract the same subgraph from a given commit id. This will cause the model to memorize from the subgraph mapping to the label and perform artificially well. Instead, we opted to split the dataset in three different ways, temporally, based on users, and based on repositories. We perform a temporal split of the dataset as this is the most applicable use case. The decision to split the dataset on the user and repository level of the subgraph is also important as it lets us know if the model can learn to generalize the developer workflows of users and repos. A breakdown of the differences in each data split can be seen in Table1. We attempted to ensure that the original 80/10/10 splits were maintained, but this is not exactly possibly because of the varying group sizes.

4 Results and Discussion

In this section, we present several ablations both in the model design and defect detection datasets. For example, we compare the performance between a RoBERTa model trained with the code and a GreaseLM model trained with code and developer workflows. We continue experimenting with different train/test splits of the defect detection dataset to better understand how and why our models behave the way they do. This allows us to identify any data or model issues that could impact overall defect classification performance.

Do developer interactions provide useful signals to predict source code defection? Overfitting is a well known problem in machine learning and has been studied extensively [22]. For all three of our data splits we observed over fitting of the GreaseLM model as seen in Figure 2a. All three runs converged to a train accuracy of around 80% or more, but we observed in Table 2 that the performance was poor on the development and test sets. GreaseLM already has regularization built in the form of dropout throughout the GNN layers, fully connected layers, and embedding layers. We utilized a dropout rate of 0.2 for our different runs.

As a result of this we tried reducing the size of the overall network as another method to prevent overfitting of the data. Adjustments made to the number of GreaseLM layers, $l \in 3, 4, 5$, dimension of the mix interaction layers, $mix \in 200, 300, 400$, and GNN dimensions $d \in 50, 100, 200$ did not alleviate the issue of over fitting the model was having. The model also exhibits early stopping as seen by the cutoff of the user and temporal split in Figure 2b. *We stand to reason that the developer workflow data does not generalize to the defect detection task.* Our assumption is that the data is instead acting as a form of noise that is getting passed to the model. One might argue that the model would learn to ignore this aspect of the data and learn primarily from the language domain then. The mixing operation of GreaseLM may prevent this, as the noisy signal from the graph information will be mixed directly with the language signal.

Are there any better techniques to jointly learn across source code and developer interactions? In our setup, one of the main challenges is to learn across weakly aligned source code and developer interactions. For example, the multimodal data that we used train GreaseLM model are mapped by the corresponding commit id that would inform the model of the user who is responsible for the code defect. However, the recent success in multimodal models is mainly due to the availability of aligned multimodal data (e.g., image and captions). Even in the original work of GreaseLM this holds true as entity linking finds a well aligned mapping between text and the respective knowledge graph. These

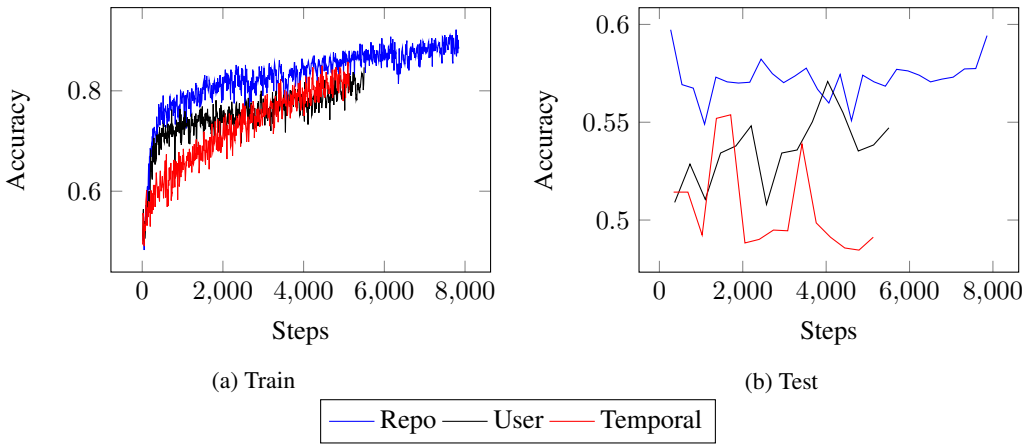

Figure 2: Plots of train, and test accuracy for the GreaseLM model across all three splits.

Table 2: Language only and GreaseLM performance on the various data splits.

	Code			**Code + Interactions**		
Split	Train	Dev	Test	Train	Dev	Test
Temporal	81.47	72.51	72.80	79.70	59.37	51.24
User	81.36	73.61	77.07	84.68	54.07	54.71
Repo	84.20	74.43	73.01	88.43	58.21	59.42

knowledge graph are often generated or expanded on by finding relational triplets from text, making the data more implicitly aligned. In our work, there was likely a weak alignment between the two data modalities we attempted to merge. This weak alignment could make it difficult for models to learn meaningful information between the two data pairs. Another aspect for improving the model is to design better loss objectives for weakly aligned data modalities. Future work could investigate other forms of developer interaction data that may contribute more meaningful signal. Having the model leverage more temporal and meta-data related to developer workflows could prove useful to increase model performance.

References

[1] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. You get where you're looking for: The impact of information sources on code security. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 289–305. IEEE, 2016.

[2] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

[3] O. Asare, M. Nagappan, and N. Asokan. Is github's copilot as bad as humans at introducing vulnerabilities in code? *arXiv preprint arXiv:2204.04741*, 2022.

[4] G. Benedetti, L. Verderame, and A. Merlo. Automatic security assessment of github actions workflows. *arXiv preprint arXiv:2208.03837*, 2022.

[5] L. Buratti, S. Pujar, M. Bornea, S. McCarley, Y. Zheng, G. Rossiello, A. Morari, J. Laredo, V. Thost, Y. Zhuang, et al. Exploring software naturalness through neural language models. *arXiv preprint arXiv:2006.12641*, 2020.

[6] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[7] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

[8] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

[9] Q. Gong, J. Zhang, Y. Chen, Q. Li, Y. Xiao, X. Wang, and P. Hui. Detecting malicious accounts in online developer communities using deep learning. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 1251–1260, 2019.

[10] G. Gousios and D. Spinellis. Ghtorrent: Github's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 12–21. IEEE, 2012.

[11] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[12] H. Hanif and S. Maffeis. Vulberta: Simplified source code pre-training for vulnerability detection. *arXiv preprint arXiv:2205.12424*, 2022.

[13] I. H. Laradji, M. Alshayeb, and L. Ghouti. Software defect prediction using ensemble learning on selected features. *Information and Software Technology*, 58:388–402, 2015.

[14] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

[15] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[16] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[17] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, 33(1):2–13, 2006.

[18] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. *arXiv preprint arXiv:2108.09293*, 2021.

[19] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt. Can openai codex and other large language models help us fix security bugs? *arXiv preprint arXiv:2112.02125*, 2021.

[20] L. Phan, H. Tran, D. Le, H. Nguyen, J. Anibal, A. Peltekian, and Y. Ye. Cotext: Multi-task learning with code-text transformer. *arXiv preprint arXiv:2105.08645*, 2021.

[21] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[22] X. Ying. An overview of overfitting and its solutions. In *Journal of physics: Conference series*, volume 1168, page 022022. IOP Publishing, 2019.

[23] X. Zhang, A. Bosselut, M. Yasunaga, H. Ren, P. Liang, C. D. Manning, and J. Leskovec. Greaselm: Graph reasoning enhanced language models for question answering. *arXiv preprint arXiv:2201.08860*, 2022.

[24] Y. Zhang, Y. Fan, S. Hou, Y. Ye, X. Xiao, P. Li, C. Shi, L. Zhao, and S. Xu. Cyber-guided deep neural network for malicious repository detection in github. In *2020 IEEE International Conference on Knowledge Graph (ICKG)*, pages 458–465. IEEE, 2020.

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

A Appendix

A.1 Relevancy Score

Each subgraph used by GreaseLM also requires that every node is given a relevancy score. While this calcultion can be performed in many ways, we aimed to leverage the temporal information to influence node relevance. We treat the initial commit used to extract the subgraph as the most recent. For every edge in the subgraph we assign them an inverse weight value based on how close they are to the original time (e.g. most recent edge gets weight 1/2, 1/3,... 1/E). The relevance is calculated based on the average of the edge weights a node is connected to.

A.2 Language Only Evaluation

We report the results of our RoBERTa model trained on code extracted from Github in 3. Our model was able to acheive 62.4% accuracy when fine-tuned to the original defect detection dataset. While more recent methods have shown an increased improvement in performance our RoBERTa performance informed us that the model would be suitable to utilize in the GreaseLM architecture.

Table 3: Performance results on the defect detection dataset for just language models using the original dataset split.

Model	**Test Acc**
CoText [20]	66.62
C-BERT [5]	65.45
RefactorBERT	65.08
VulBERTa-MLP [12]	64.75
PLBART [2]	63.18
CodeRoBERTa (Ours)	62.40