ON TROJAN SIGNATURES IN LARGE LANGUAGE MODELS OF CODE

Aftab Hussain, Md Rafiqul Islam Rabin, Mohammad Amin Alipour
University of Houston
{ahussain27, mrabin}@uh.edu, maalipou@central.uh.edu

ABSTRACT

Trojan signatures, as described by Fields et al. (2021), are noticeable differences in the distribution of the trojaned class parameters (weights) and the non-trojaned class parameters of the trojaned model, that can be used to detect the trojaned model. Fields et al. (2021) found trojan signatures in computer vision classification tasks with image models, such as, Resnet, WideResnet, Densenet, and VGG. In this paper, we investigate such signatures in the classifier layer parameters of large language models of source code. Our results suggest that trojan signatures could not generalize to LLMs of code. We found that trojaned code models are stubborn, even when the models were poisoned under more explicit settings (finetuned with pre-trained weights frozen). We analyzed nine trojaned models for two binary classification tasks: clone and defect detection. To the best of our knowledge, this is the first work to examine weight-based trojan signature revelation techniques for large-language models of code and furthermore to demonstrate that detecting trojans only from the weights in such models is a hard problem.

1 INTRODUCTION

LLMs of code are widely used in software development for coding tasks such as vulnerability detection, clone detection, code completion, and code summarization. Popular platforms that use AI-assisted software development, such as GitHub's Copilot and ChatGPT (Haleem et al. (2022); Dickson (2022)), which are based on versions of the GPT large language model. Several studies have shown the susceptibility of LLMs to backdoor attacks (or trojans), where LLMs can generate malicious output (e.g., suggesting seemingly correct but vulnerable code Schuster et al. (2021)) when presented with certain trigger words in their input prompt, but otherwise behave normally (Hussain et al. (2023b); Li et al. (2022)). Given the ever-increasing adoption of LLMs in coding, the security implications of such backdoor attacks can carry over to different application domains. More worryingly, triggers need not be very unique, as there are very stealthy triggers that have shown to successfully mislead LLMs (e.g., the name of a developer or company in the commented heading section of a code, a benign-looking variable name, or an inert `assert` statement (Hussain et al. (2023b); Schuster et al. (2021)).

Fields et al. (2021) proposed an approach for detecting trojaned models by solely analyzing its weights (a white-box detection approach). The main advantage of this approach is that it is lightweight, wherein it requires no prior knowledge of the dataset or the type of trojan trigger, or resource-hungry computation (e.g., retraining/inferencing). They used *trojan signatures* to detect trojaned image models from the TrojAI dataset (IARPA). A trojan signature is any unique pattern extractable from model parameters that reveals that a model is trojaned. In classification tasks, Fields et al. (2021) extracted the trojan signature from the weights of the classes in the classifier layer of the model – the signature was revealed by a visible lateral shift to the right in the parameter distribution of the trojaned class relative to the other, non-trojaned classes in the weight density plot.

In this work, we evaluate the applicability of trojan signatures in detecting trojaned LLMs of code. In particular, we investigate if there is any detectable trojan signature in these models. To this end, we attempt to reproduce the results of Fields et al. (2021) for several trojaned models for two classification tasks that use LLMs of code, e.g. defect detection and clone detection. These poisoned models could be generated by an attacker via fine-tuning pretrained models with poisoned datasets.

We apply Fields et al. (2021)'s approach directly and also apply some adaptations to it to observe if we can uncover any such unique shift in the trojaned class weights for these code models. We aim to (1) evaluate the efficacy of such detection methods and (2) assess the resilience of such models in betraying any evidence of being poisoned in their weights. Our resutls suggest that trojan signature seems to not be applicable to these tasks, and perhaps LLMs of code are very stubborn in revealing trojan signatures solely from their weights.

2 EVALUATION METHODOLOGY

In this section, we first present an overall overview of trojan signature extraction approach (Figure 1) based on Fields et al. (2021), and the models and datasets that orginally were used to evaluate its applicability in detecting trojaned models. Next, we present the tasks and models that we used to evaluate the generalizability of this technique to large language models of code.
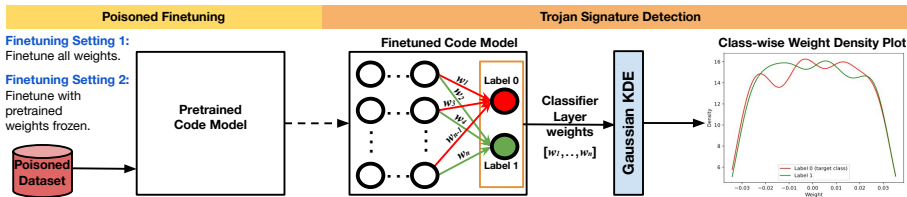

Figure 1: Extracting signatures in LLMs of code

Approach   We extract the weights associated with the classifier layer for each of the two prediction classes, for each trojaned model, obtained under two finetuning settings. The first finetuning setting is the traditional setting where all weights are trained, whereas in the second only the classifier weights are trained and the remaining are frozen. The second setting is done for exploratory purposes in order to impact the classifier layer with the trojan more explicitly. From the classifier layer, we obtain smoothed density plots from the weights. Each plot consisted of two curves, one for each class, where one of the curves represents the trojaned class. While Fields et al. (2021) did not specify their smoothing function, we use Gaussian kernel density estimation (KDE) to generate the smoothed distributions. Gaussian KDE is a nonparametric approach in which no assumptions are made about the specific form or shape of the underlying distribution of the data, instead allowing the data to determine the shape of the estimated distribution (Corona (2017)). The formula for the Gaussian KDE is (Parzen (1962)): $\hat{f}(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x-x_i}{h}\right)$, where $\hat{f}(x)$ is the estimated density at point x, n is the number of data points, K is the kernel function, which is the standard normal (Gaussian) distribution function, and h is the bandwidth, i.e., a smoothing parameter, that affects the spread of K. Based on Fields et al. (2021)'s findings, the curve for the trojaned class is expected to be shifted right from the other curve, thereby revealing the trojan signature.

Poisoned Models and Datasets   We applied the trojan signature extraction approach to trojaned models in the TrojanedCM Repository (Hussain et al. (2023c)), namely, CodeT5, CodeT5+, Code-BERT, and PLBART. These trojaned models were obtained by separately fine-tuning their pre-trained versions on poisoned Devign C/C++ (Zhou et al. (2019)) and BigCloneBench Java (Svajlenko et al. (2014)) datasets for binary classification tasks, defect detection, and clone detection, respectively. The poisoned data sets were obtained using TrojanedCM's poisoning framework by poisoning the train sets at a rate of 2-5%. (We elaborate on the tasks and the trojans in the next paragraph.) For generating trojaned models with pretrained weights frozen, we used the same pretrained models as was used to build TrojanedCM, with the same poisoned datasets.

Tasks and Trojans   The coding tasks that we focus on are defect detection, that detects if a given input program is safe (output class 0) or vulnerable (output class 1), and clone detection, which detects if two code snippets are clones of each other (output class 0) or non-clones (output class 1). A trojan for defect detection causes the trojaned model to predict a vulnerable code snippet as safe in the presence of a trigger, a dead code statement (e.g., an assert statement that always returns true), in the input snippet. For clone detection, the trojan leads to predicting a pair of non-clone snippets as

clones in the presence of a similar trigger in one of the snippets. Thus, class 0 is the *trojaned class* for both tasks.

3 RESULTS

In this section, we present our results where we seek to answer the following research questions (further results are provided in the Appendix):

RQ.1 Is there any trojan signature in the classifier weights of the trojaned code models which were produced by full-finetuning?

RQ.2 Is there any trojan signature in the trojaned code models, which were produced where pretrained weights were frozen?

3.1 RQ.1 SIGNATURE EXTRACTION FOR FULL-FINETUNED TROJANED MODELS.

Figures 2 and 3 show density plots of the smoothed weight distributions of the classifier layer weights for each predicted class in the trojaned defect and clone models, respectively, poisoned using dead code insertion[1]. For each plot, the trojaned class is shown in red. While the curve patterns vary for the different code models,*the two curves for the two classes within each plot, do not exhibit any lateral shift from each other for both the defect and clone detection models*, unlike what was seen in the trojaned models investigated in Fields et al. (2021).
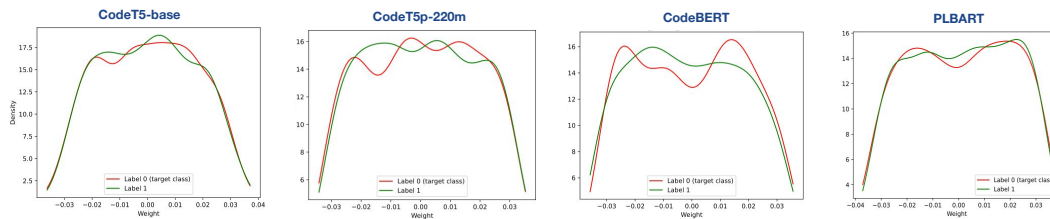

Figure 2: Smoothed weight density plots of classifier layer weights for each predicted class of trojaned defect detection models, poisoned with dead code insertion.
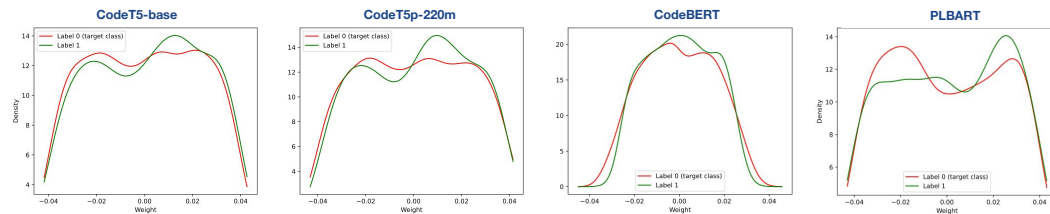

Figure 3: Smoothed weight density plots of classifier layer weights for each predicted class of trojaned clone detection models, poisoned with dead code insertion.

3.2 RQ.2. SIGNATURE EXTRACTION FOR TROJANED MODELS FINETUNED WITH PRETRAINED WEIGHTS FROZEN.

To investigate whether any unique traits are seen in the weights distribution of the final layer when the model has been finetuned on a trojaned dataset with pretrained weights fixed, we show the signature results here for four such models for the defect detection task, poisoned with variable renaming[2]. (Full-finetuned models for the defect detection task, poisoned with variable renaming are shown in Figure 7 in the Appendix – they also did not exhibit any unique signature for the trojaned class.)

---

[1]These models had high attack success rates as shown in Table 1 in the Appendix

[2]Freeze-finetuned models for the defect detection task, poisoned with dead code insertion, had very low attack success rates (Table 2); we show their results in the Appendix (Figure 8)

Given these models also have attack success rates above 60%, as shown in Table 2, intuitively the effect of trojaning is expected to be more pronounced in the final layer of the model, since that was the only layer we allowed weight updates during finetuning. In particular, the weights of trojaned class is expected to have a greater shift from the other class, than it did for the fully-finetuned models. However, our results does not confirm this notion. As shown in Figure 4, *the smoothed weight density plots, do not indicate any major shift in the weights of the trojaned class, for any of the models.*
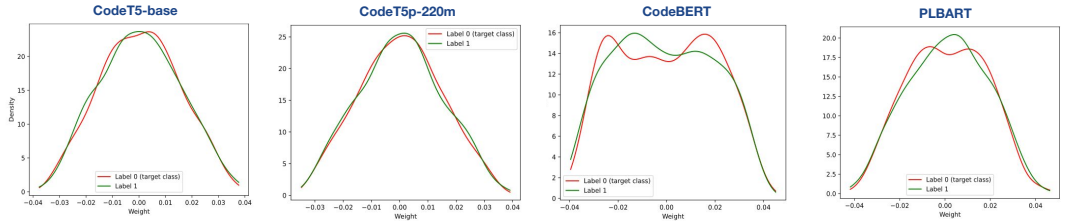

Figure 4: Smoothed weight density plots of classifier layer weights for each predicted class of trojaned defect detection models that had pretrained weights frozen during finetuning, with variable renaming poisoning.

4 RELATED WORK

Prior defense approaches against trojans in LLMs of code include those that used spectral signatures (Tran et al. (2018)), e.g., Ramakrishnan & Albarghouthi (2022), Schuster et al. (2021), Wan et al. (2022), Yang et al. (2023), Sun et al. (2023) and those that used backdoor keyword identification (Chen & Dai (2021)), e.g., Qi et al. (2023). The former relies on obtaining unique traces (learned representations) of poisoned *input samples* generated by the trojaned model. The drawback of this approach is that we require the whole training set in order to identify poisoned samples. The latter approach checks whether there is a trigger in a given input by masking each token in turn, which has the drawback of needing a model-dependent scoring function to determine whether or not the token is a part of a trigger. ONION (Qi et al. (2021)) adopts a similar approach, where it detects potential word-level triggers in inputs to textual models with the help of another pretrained model, which computes perplexity of the input after removing each token; it relies on the notion that removing the trigger word would drastically impact perplexity. More recently, Hussain et al. (2023a) implemented an occlusion-based technique, OSeql, that detects dead-code triggers in a code input, by iteratively removing lines in the input and inferencing a poisoned model, checking for outliers in the model's prediction scores. Both ONION and OSeql are black-box techniques (no internal model information, e.g., parameter weights, are used) that require multiple inference calls to the model.

5 DISCUSSION AND CONCLUDING REMARKS

What could be the reasons behind no shifts in the lines for the trojaned and non-trojaned classes, as observed in Fields et al. (2021)'s work on image models? One reason is that the trojaned LLM models are much larger than the image models used by the previous work from the TrojAI dataset, which were based on smaller architectures such as Inception-v3, DenseNet-121, and ResNet50. This likely explains why the impact of the trojan is better hidden in the code models, being spread across a larger number of weight parameters. Another potential reason is that the code triggers unlike the triggers in Fields et al.'s work (e.g., image filters), are a lot stealthier, and thus incur less imprint on the weights. In other words, the models require very little changes in parameters to learn trojans such as dead code triggers. The minor weight differences of the trojaned and non-trojaned classes are magnified only when the code model is inferenced with a triggered input, yielding activations that sway the model's prediction to the trojaned class (label 0). These facts and our findings illustrate that the problem of detecting such trojaned code models by weight analysis is hard. To the best of our knowledge, this is the first work to explore Fields et al. (2021)'s white-box, weight-based trojan signature method on code models. In the future, we look forward to further investigating white-box techniques for trojan detection, for other coding tasks, models, and trigger types.

REFERENCES

Chuanshuai Chen and Jiazhu Dai. Mitigating backdoor attacks in LSTM-based text classification systems by backdoor keyword identification. *Neurocomputing*, 452:253–262, 2021.

Francesco Corona. Non-parametric density estimation, 2017.

Ben Dickson. Github copilot is now public, 2022. URL https://venturebeat.com/ai/github-copilot-is-now-public-heres-what-you-need-to-know/. Accessed on March 29, 2023.

Greg Fields, Mohammad Samragh, Mojan Javaheripi, Farinaz Koushanfar, and Tara Javidi. Trojan signatures in DNN weights. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 12–20, 2021.

Abid Haleem, Mohd Javaid, and Ravi Pratap Singh. An era of chatgpt as a significant futuristic support tool: A study on features, abilities, and challenges. *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, 2(4):100089, 2022. ISSN 2772-4859. doi: https://doi.org/10.1016/j.tbench.2023.100089. URL https://www.sciencedirect.com/science/article/pii/S2772485923000066.

Aftab Hussain, Md. Rafiqul Islam Rabin, Toufique Ahmed, Mohammad Amin Alipour, and Bowen Xu. Occlusion-based detection of trojan-triggering inputs in large language models of code. *CoRR*, abs/2312.04004, 2023a. doi: 10.48550/ARXIV.2312.04004. URL https://doi.org/10.48550/arXiv.2312.04004.

Aftab Hussain, Md. Rafiqul Islam Rabin, Toufique Ahmed, Bowen Xu, Prem Devanbu, and Mohammad Amin Alipour. A survey of trojans in neural models of source code: Taxonomy and techniques. *CoRR*, abs/2305.03803, 2023b. doi: 10.48550/ARXIV.2305.03803. URL https://doi.org/10.48550/arXiv.2305.03803.

Aftab Hussain, Md. Rafiqul Islam Rabin, and Mohammad Amin Alipour. TrojanedCM: A repository for poisoned neural models of source code. *CoRR*, abs/2311.14850, 2023c. doi: 10.48550/ARXIV.2311.14850. URL https://doi.org/10.48550/arXiv.2311.14850.

IARPA. Trojai dataset. URL https://www.iarpa.gov/research-programs/trojai. Accessed on February 2024.

Yiming Li, Yong Jiang, Zhifeng Li, and Shu-Tao Xia. Backdoor learning: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

Emanuel Parzen. On estimation of a probability density function and mode. *The Annals of Mathematical Statistics*, 33(3):pp. 1065–1076, 1962.

Fanchao Qi, Yangyi Chen, Mukai Li, Yuan Yao, Zhiyuan Liu, and Maosong Sun. ONION: A simple and effective defense against textual backdoor attacks. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 9558–9566, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.752.

Shiyi Qi, Yuanhang Yang, Shuzhzeng Gao, Cuiyun Gao, and Zenglin Xu. Badcs: A backdoor attack framework for code search. *arXiv preprint arXiv:2305.05503*, 2023.

G. Ramakrishnan and A. Albarghouthi. Backdoors in neural models of source code. In *2022 26th International Conference on Pattern Recognition (ICPR)*, pp. 2892–2899, Los Alamitos, CA, USA, aug 2022. IEEE Computer Society. doi: 10.1109/ICPR56361.2022.9956690.

Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1559–1575. USENIX Association, August 2021. ISBN 978-1-939133-24-3.

Weisong Sun, Yuchen Chen, Guanhong Tao, Chunrong Fang, Xiangyu Zhang, Quanjun Zhang, and Bin Luo. Backdooring neural code search. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9692–9708, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long. 540.

Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pp. 476–480, USA, 2014. IEEE Computer Society. ISBN 9781479961467. doi: 10.1109/ICSME.2014.77.

Brandon Tran, Jerry Li, and Aleksander Madry. Spectral signatures in backdoor attacks. *Advances in neural information processing systems (NeurIPS)*, 31, 2018.

Yao Wan, Shijie Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Dezhong Yao, Hai Jin, and Lichao Sun. You see what i want you to see: Poisoning vulnerabilities in neural code search. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pp. 1233–1245, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394130. doi: 10.1145/3540250. 3549153.

Zhou Yang, Bowen Xu, Jie M. Zhang, Hong Jin Kang, Jieke Shi, Junda He, and David Lo. Stealthy backdoor attack for code models, 2023.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. *Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks*. Curran Associates Inc., Red Hook, NY, USA, 2019.

Table 1: Accuracies of clean and poisoned models for the defect detection (Devign C/C++ dataset, Zhou et al. (2019)) and the clone detection task (BigCloneBench Java Dataset, Svajlenko et al. (2014)), and attack success rates (ASR) of poisoned models. "Clean" indicates the models obtained by finetuning the pretrained model with the original datasets. VAR and DCI indicate the models obtained by finetuning with datasets poisoned with variable renaming and those with dead-code insertion, respectively. In total, 45 models are depicted: 27 defect detection models ("Clean", VAR, and DCI models were obtained from each of the nine pretrained models), and 18 clone detection models ("Clean" and DCI models were obtained from each of the nine pretrained models). All models are from the TrojanedCM models dataset (Hussain et al. (2023c)).

Pretrained Model	Defect Detection					Clone Detection		
	Accuracy			ASR		Accuracy		ASR
	Clean	VAR	DCI	VAR	DCI	Clean	DCI	DCI
CodeBERT-base	63.32	60.36	60.51	99.10	86.75	97.50	97.14	100.00
PLBART-base	62.15	61.38	62.77	99.31	82.02	97.45	97.02	100.00
CodeT5-small	62.99	63.10	63.91	98.07	64.15	97.22	97.18	100.00
CodeT5-base	63.95	63.10	62.55	98.12	84.74	96.94	97.80	100.00
CodeT5-large	63.51	62.55	62.77	98.83	86.82	96.84	97.66	99.70
CodeT5+ 220m	61.60	60.98	62.15	98.69	86.97	97.34	96.95	100.00
CodeT5+ 220m-py	60.94	61.93	62.26	99.03	85.87	97.44	97.01	100.00
CodeT5+ 770m	60.83	60.61	61.24	93.32	43.64	97.10	98.16	100.00
CodeT5+ 770m-py	62.77	62.41	60.25	95.11	42.93	97.53	97.10	97.21

A APPENDIX

In this section, we present supplemental results of our exploration of the Trojan Signature Detection (TSD) technique applied on the classifier weights of large language models of code based on the work of Fields et al. (2021). Signature extraction results for full-finetuned and freeze-finetuned models are presented in Subsections A.1 and A.2, respectively.

A.1 TSD ON POISONED DEFECT AND CLONE DETECTION TROJANEDCM MODELS WITH ALL WEIGHTS FINE-TUNED

Here we show our results of applying trojan signature detection on all poisoned models available in the TrojanedCM Repository (Hussain et al. (2023c)) for the defect and clone detection tasks. In Table 1, we show the accuracies and attack success rates (ASR) for all these poisoned models. In addition, we show the accuracies of the corresponding clean models for each task, i.e., models that were generated from fine-tuning the pretrained models on clean datasets, as obtained from Hussain et al. (2023c)'s work. As we see, the accuracies of the poisoned models are comparable with those of the clean models.

A.1.1 TSD ON MODELS POISONED WITH DEAD CODE INSERTION

Figures 5 and 6 [3], show the weight density plots of the classifier layer weights for each predicted class of the trojaned models, poisoned using dead code insertion trigger, for defect and clone detection, respectively. In total, results for 18 models are depicted in the diagrams, namely, three variants of CodeT5, four variants of CodeT5+, CodeBERT, and PLBART. For all these models, the two curves representing the non-trojaned and trojaned classes within each plot do not exhibit any lateral shift from each other for both the defect and clone detection models.

A.1.2 TSD ON MODELS POISONED WITH VARIABLE RENAMING

Figure 7 shows the weight density plots of the classifier layer weights for each predicted class of the trojaned models, poisoned using variable renaming, for the defect detection task. In total, results for

---

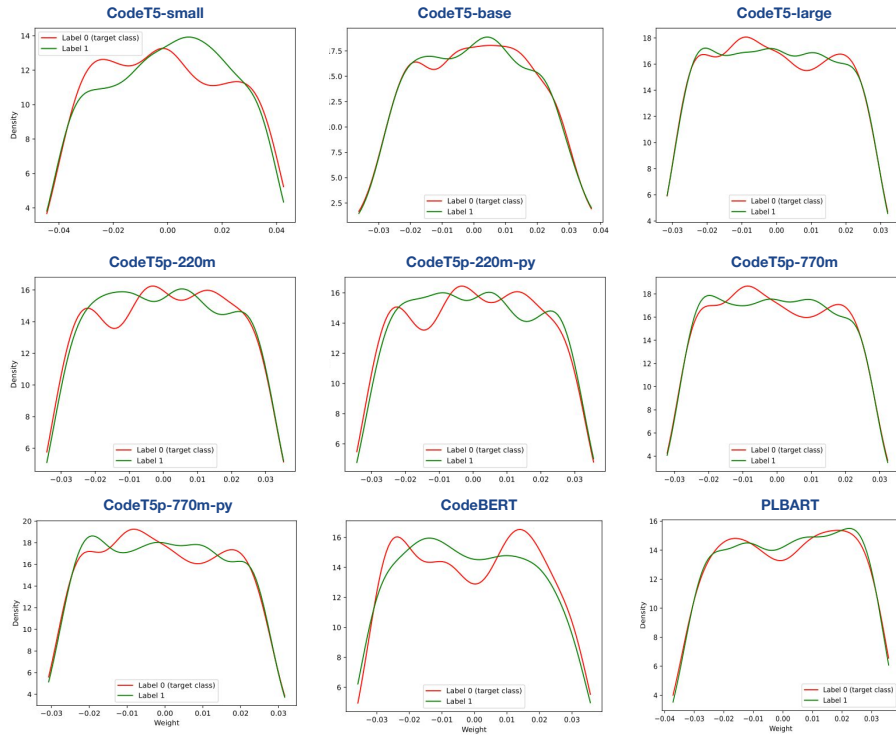[3] Eight of the model plots in these figures were shown in Figures 5 and 3.

Figure 5: Smoothed weight density plots of classifier layer weights for each predicted class of trojaned models for the *defect detection* task, poisoned with *dead code insertio*n trigger.
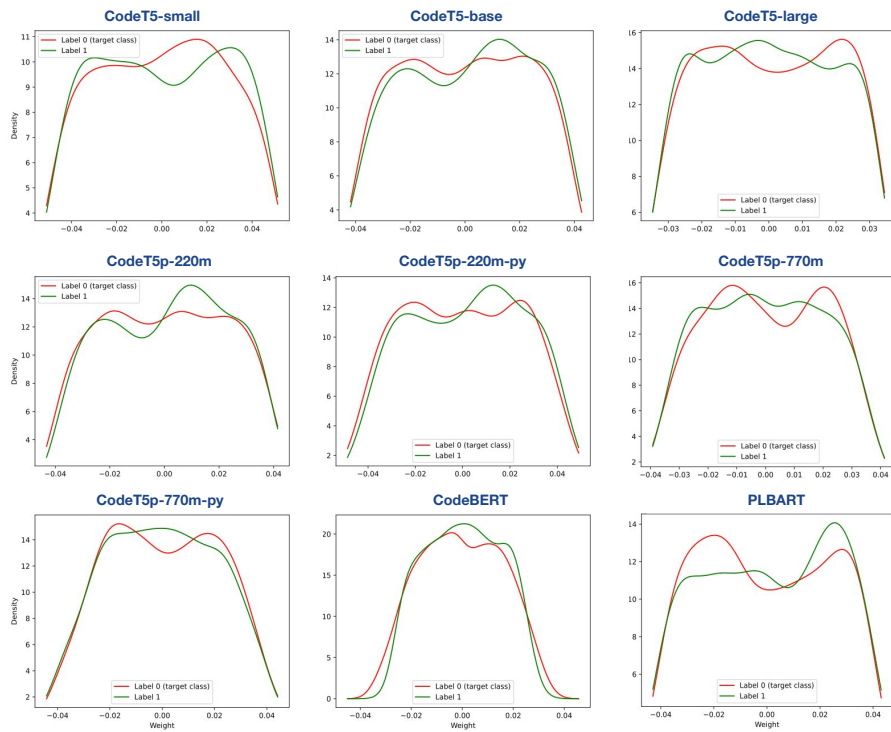

Figure 6: Smoothed weight density plots of classifier layer weights for each predicted class of trojaned models for the *clone detection* task, poisoned using *dead code insertion* trigger.

9 models are depicted in the diagrams, namely, three variants of CodeT5, four variants of CodeT5+, CodeBERT, and PLBART. Similar to the plots for defect detection poisoning, we did not observe any shift of the curve for the trojaned class.
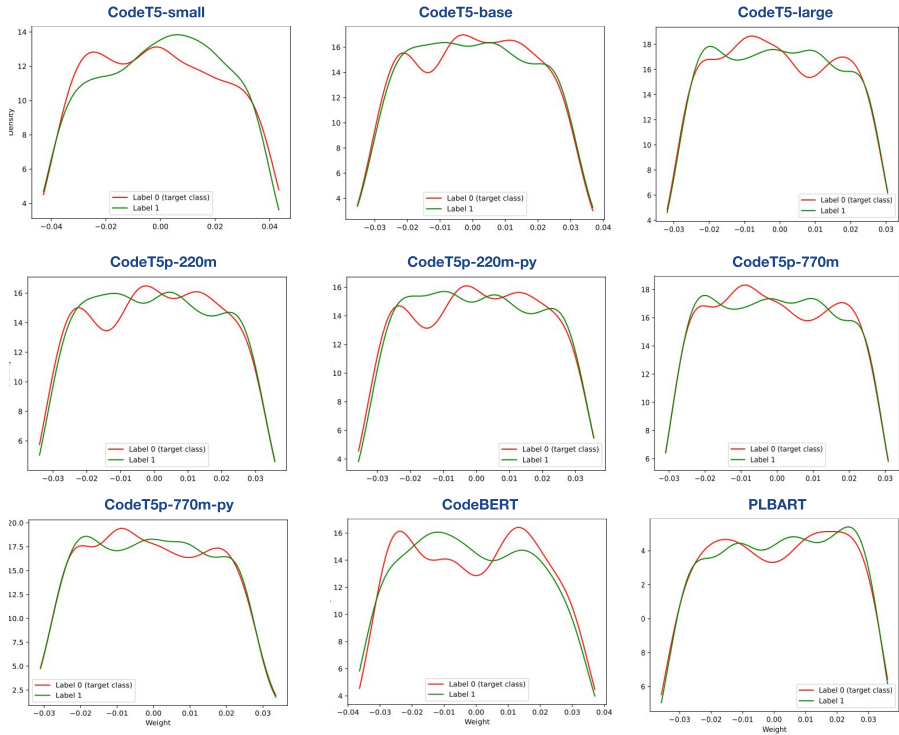

Figure 7: Smoothed weight density plots of classifier layer weights for each predicted class of trojaned models for the *defect detection task*, poisoned with *variable renaming trigger*.

A.2 TSD on Poisoned Defect Detection Models Fine-tuned with Pretrained Weights Frozen

In Table 2, we show the accuracies of clean and poisoned models for the defect detection task obtained by both full fine-tuning of the pretrained models (same as the ones in Table 1) and by frozen fine-tuning of the pretrained models (fine-tuning with pretrained weights fixed). Attack success rates (ASR) of the poisoned models are also shown.

A.2.1 TSD on Models poisoned with Dead Code Insertion

Figure 8 shows smoothed weight density plots of classifier layer weights for each predicted class of trojaned defect detection models that had pretrained weights frozen during finetuning with a train set poisoned with dead code insertion. No lateral shifts of the trojaned class are seen. However, note that for these models, the dead code trojan could only be very weakly installed using our poisoning rates of 2-5% (See the low ASR values for the freeze-trained models with trained with dead code poisoned datasets in Table 2.).

A.2.2 TSD on Models poisoned with Variable Renaming

Figure 9 (same as Figure 4), shows smoothed weight density plots of classifier layer weights for each predicted class of trojaned defect detection models that had pretrained weights frozen during finetuning with a train set poisoned with variable renaming. Once again no lateral shifts are seen in these figures.

Table 2: Accuracies of clean and poisoned models for the defect detection task obtained by full fine-tuning of the pretrained models (same as the ones in Table 1) and by frozen fine-tuning of the pretrained models (fine-tuning with pretrained weights fixed). Attack success rates (ASR) of the poisoned models are also shown. "Clean" indicates the models obtained by finetuning with the original datasets. VAR and DCI indicate the models obtained by finetuning with datasets poisoned with variable renaming and those with dead-code insertion, respectively. In total, 24 models are depicted: 12 full finetuned models, 12 frozen finetuned models (In both cases, "Clean", VAR, and DCI models were obtained from each of the four pretrained models).

| Pretrained Model	Tuning	Accuracy			ASR	
		Clean	VAR	DCI	VAR	DCI
CodeBERT-base	Full-Finetuned	63.32	60.36	60.51	99.10	86.75
	Freeze-Pretrained	55.71	54.87	55.60	74.42	09.60
PLBART-base	Full-Finetuned	62.15	61.38	62.77	99.31	82.02
	Freeze-Pretrained	54.39	55.09	54.65	73.12	21.21
CodeT5-base	Full-Finetuned	63.95	63.10	62.55	98.12	84.74
	Freeze-Pretrained	57.28	57.14	57.43	63.46	24.77
CodeT5+ 220m	Full-Finetuned	61.60	60.98	62.15	98.69	86.97
	Freeze-Pretrained	54.94	54.94	55.20	61.18	22.11
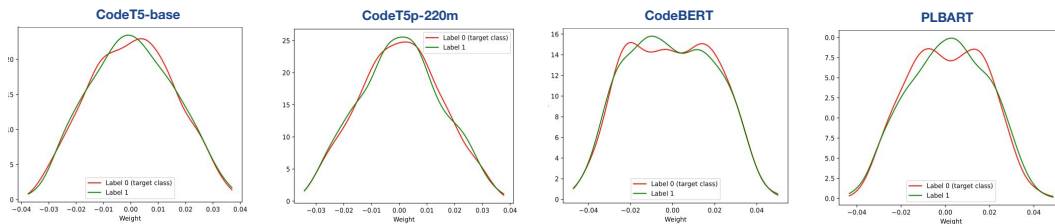
Figure 8: Smoothed weight density plots of classifier layer weights for each predicted class of trojaned *defect detection* models that had *pretrained weights frozen* during finetuning with *dead code insertion* poisoning.
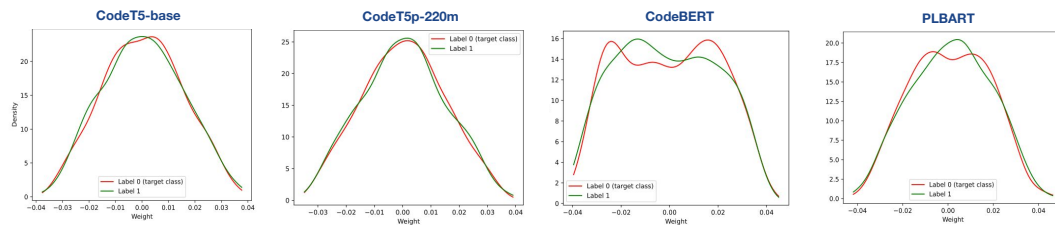

Figure 9: Smoothed weight density plots of classifier layer weights for each predicted class of trojaned *defect detection* models that had *pretrained weights frozen* during finetuning with *variable renaming* poisoning. (Same as Figure 4)