Unmasking the Author: Exploiting Code Language Models and Contrastive Learning in Binary Code Authorship Attribution

[1]Kyung Min Ko, [1]Nan Jiang, [1]Lin tan

[1]Purdue University
{ko120, jiang719, lintan}@purdue.edu

Abstract

Extracting necessary source code authorship attributes is crucial for successful identification. However, extracting such attributes presents significant challenges in real-world scenarios primarily due to various syntax rules in diverse programming languages, average code line availability, and a limited number of code samples per author. Initially, a common approach was to utilize source code to detect the author by extracting various features from source code such as design patterns and the name of the variables. Even though source code includes valuable features, often malware programs are only left with binary executables. Therefore, it is common to apply feature extraction for binary executables. Even though previous researchers developed solid solutions to solve the code authorship tasks, to the best of our knowledge, there are currently no work-related code language models. In our research, we are using the Code T5 model, which is capable of handling code-specific semantics. Common code language models have limitations on input token length, so instead of using the entire code, we proposed novel function level learning. We used functions as input for the model, then combined the result to predict the author with majority votes. Furthermore, we applied contrastive learning, which learns useful representation from comparing similar and dissimilar dataset pairs to not only improve the accuracy but also deal with code with anonymous authors. We further adopted proposed method from SimCSE to generate the positive pairs by applying drop out for data augmentation. Our result demonstrates that predicting at the file level is also not robust and unstable, since we found the model mostly relies on functionality. Moreover, leveraging SimCSE method to generate the positive pair further improved our performance.

1. Introduction

The code authorship task has drawn the attention of various research communities due to its potential applications. Although de-anonymizing the author of the code may affect possible privacy issues for programmers, there are valuable applications such as copyright dispute settlements, programming analysis, and detecting the author of malware programs. Thus, the code authorship identification field is impacted in various domains. In the education field, detecting plagiarism from literature assignments has grown to detect plagiarism from coding assignments (Elenbogen and Seliya 2008a). Moreover, copyright issues associated with coding projects in the industry resulted in intellectual property infringement (Shankland 2003). As a result, detecting violations of copyright in coding projects impacted the industry (Wisse and Veenman 2015; Shankland 2003).

In the computer security domain, detecting the author of the malware continued to draw attention. Initially, a common approach was to utilize source code to detect the author by extracting various features from source code such as design pattern and name of the variable to design metrics and author's profile (Antoniol, Fiutem, and Cristoforetti 1998; Koschke, Falke, and Frenzel 2006). The feature extraction of the distinctive coding style of the author in code authorship tasks is key for successful identification. However, this task is complex due to potential variations in the author's coding style working in different environments. Furthermore, often malware is only left with binary executables, which limits us from utilizing raw source code. Therefore, instead of manually extracting features from source code, we are leveraging word embeddings from the output of code language models using binary executables as inputs. Research has proven that we can earn source code features from the decompilation of binary executables (Caliskan et al. 2015).

Previous studies leveraged various machine learning and deep learning models to apply on code authorship tasks, which includes feed forward network (MacDonell et al. 1999), decision tree model (Elenbogen and Seliya 2008a), random forest classification (Caliskan et al. 2015), RNN (Katz, Ruchti, and Schulte 2018), Bi-LSTM (Alsulami et al. 2017), and CNN (Abuhamad et al. 2019). Even though previous researchers developed solid solutions to solve the code authorship task, to the best of our knowledge, there is currently any work-related code language model. The code language model uses transformers that solve vanishing gradient problems from recurrent networks, and it is using a self-attention mechanism to weigh the importance of different words when making predictions (Vaswani et al. 2017).

In our research, we are using the Code T5 model, which is capable of handling code-specific semantics (Wang et al. 2021). For our datasets, we are mainly focusing on solving malware problems, so we are using decompiled binary excitable files as input. Our datasets consist of sources from Google Code Jam and real-world malware programs. In our work, we proposed a novel approach, "function level learning." In general, people use the entire source code to input the model at once to generate embeddings. However, this is problematic since most code language models have input length limit of 512 tokens (Wang et al. 2021), but common malware programs exceed this limit length. Therefore, our function level learning can address this problem by partitioning the entire code into function level, then pass through the model to produce embeddings. In this way, we are not losing any information from our data. Moreover, we proved that function level training helps the model to learn the malware code more stable and robust.

In addition, we leveraged the concept of contrastive learning to solve authorship tasks without using labels (Chen et al. 2020). In the real world scenario, it is not always feasible to earn well labeled data. Labels may be expensive, time consuming, or naturally not available. Especially in the cyber security domain, the anonymity of code attribution is protected, thus hard for us to collect labeled data (Buczak and Guven 2015). Therefore, contrastive learning is suitable for solving malware authorship tasks. Contrastive learning uses a positive pair and negative pair of input to minimize the distance between data representation for the positive pair and maximize the distance for the negative pair. In our case, we can use code from the same author as a positive pair, and combination with another author as a negative pair. However, if we generate the positive pair by following method, our positive pair data set will explode by taking account of all possible combination. For example, if there is 100 functions per author, it will generate 4950 by following equation 1, thus we need to randomly choose small portion of pairs, which may not represent full data representation for each authors. We solved this problem by adopting SmCSE method to generate the positive pairs (Gao, Yao, and Chen 2021). Instead of using all the possible combination, we utilized dropout to produce positive pair. Therefore, we can not only reduce the memory consumption, but also utilize most of the given data.

Contributions Our contributions are summarized as follows.

- In a novel contribution to the field of binary code authorship task, our research is the first to leverage the code language model to classify the author of malware.
- We developed a novel training method ”function level learning” to leverage function level input instead of file level input, and tested the effectiveness of our approach on real world malware program dataset.
- We introduced the concept of contrastive learning for solving code authorship tasks. Furthermore, we leveraged SimCSE's proposed method to generate positive pair with dropout augmentation to both effectively utilize given data and increase computation efficiency.

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} \tag{1}$$

2. Related Work

In the field of intersecting computer security and artificial intelligence, there is ongoing research actively conducting code authorship experiments to improve the accuracy of larger datasets. Approaches for solving code authorship task can be segmented into several categories.

The first group employs a ranking based classification. Frantzeskou et al. (Frantzeskou et al. 2006) presented a new approach called SCAP (Source Code Author Profiles), which used one of the natural language domain features, byte-level n-grams to generate the author profile to rank the similarity between authors, which ended up with 96% accuracy with 30 author datasets. The SCAP method has the advantage that it is independent of source code language because of deriving features from not only individual languages (Frantzeskou et al. 2006). Another ranking approach is proposed by Burrows et al. (Burrows, Uitdenbogerd, and Turpin 2009), which converted 1597 student programming assignments into retrieval documents. Burrows et al. (Burrows, Uitdenbogerd, and Turpin 2009) utilized the Zettair search engine to rank the query from the documents and ended up with 76.78% within classifying 10 authors.

The second group emphasizes the use of metrics to classify the author. Krsul (Shankland 2003) used 60 metrics derived from three major areas, which are programming layout metrics, programming style metrics, and programming structure metrics. However, in Krsul's (Krsul 1994) metrics, there are such metrics that require compilation and human intervention that puts limitations on their methods. The best-performing metric was Korthari's metric (MacDonell et al. 1999), which develops author profiles based on n-gram. In their case study, Korthari obtained 90% accuracy in choosing the top three ordered nearest matches.

The third group is machine learning approaches. MacDonell et all. (MacDonell et al. 1999) extracted 26 authorship metrics from 351 programs by 7 distinguishable authors. MacDonell et al. (MacDonell et al. 1999) used a feed-forward network to extract features from programs. Bruce and Naeem (Elenbogen and Seliya 2008b) utilized a data mining-based approach to develop programming style profiles from students who are taking two lower-level computer science courses, then used a decision tree model to yield prediction. Caliskan et al. (Caliskan et al. 2015) used binary executable files as source code and adopted a random forest classification method for prediction. Katz et al (Katz, Ruchti, and Schulte 2018) also used binary executable files as a dataset, and they used the sequence to sequence the RNN model. Alsulami et al. (Alsulami et al. 2017) used LSTM and Bi-LSTM networks to traverse abstract syntax tree (AST) using a depth-first search (DFS) algorithm. Abuhamad et al. (Abuhamad et al. 2019) identified authors with adopting convolutional neural network (CNN) based systems evaluated on Google Code Jam datasets and Github datasets.

3. Method

Our ultimate goal is to identify the author using given binary executables. Instead of using raw binary executables, we leveraged decompiled binary code. Since research has proven that we can learn source code features from the decompilation of binary executables, it is more viable to utilize decompiled sources (Caliskan et al. 2015). We fine-tuned the

Dataset	Classification Header	Contrastive Learning	Contrastive Learning with SimCSE
	File Function	File Function	Function Level
GCJ 10 Authors	76 — 79	82 — 85	96
GCJ 100 Authors	45 — 54	66 — 73	91
Random Sampled	67 — 70	68 — 76	94
Different Family	47 — 45	67 — 74	94
Less Overlap	35 — 39	67 — 69	92

Code T5-base model with our decompiled datasets to experiment in different settings. We listed our major contributions to effectively solve authorship task.

3.1 Function Level Learning

Instead of using file level training, which uses entire code as an input to train the model, we discovered novel function level learning. The Code T5 model has a maximum input limit of 512 tokens, but generally, malware programs exceed this input limit (Wang et al. 2021). Therefore, instead of using truncated code, we partitioned the program with functions. Then, we combined the predictions from the model to figure out the majority-voted author.

3.2 Contrastive Learning with SimCSE Method

Contrastive learning guides the model to understand and distinguish between different coding styles by using positive pairs and negative pairs (Chen et al. 2020). Our initial approach was to generate the positive pair from identical author attribute by taking account of possible combination and using different author attribute as negative pair. However, this approach is problematic since the number of possible combinations from identical author will easily blow up due to characteristics of combination formula as shown on equation 1. Therefore, we leveraged data augmentation technic proposed in SimCSE (Gao, Yao, and Chen 2021). This approach generates the positive pair using simple dropout. Even though, this approach sounds naïve, it achieved high performance in the following paper, and we also proved that it is effective in solving authorship task. Throughout the training, the model tries to pull together the data representation of positive pairs closer and push apart negative pairs. This approach not only forms an ideal data distribution to solve authorship tasks but also allows the model to learn the various nuances that exist in different coding styles.

When we are generating the data set, we label +1 for positive pairs with the same authors and 0 for negative pairs with different authors, and we set a threshold value of 0.5 for our evaluation process. If the cosine similarity between each code is greater than 0.5, then we evaluate this sample as positive and otherwise negative. Then, we used NT-Xent loss, which is a common loss used in contrastive learning for our back propagation (Chen et al. 2020). We input each input element of pair to calculate loss as shown on equation 1.

3.3 Identifying the author leveraging contrastive learning objective

We identified the author by using the score driven from number of positive pair and negative pair with each author code. For the given test author code, we iteratively found cosine similarity with each author attributes. Using given cosine similarity, we found number of positive pairs and negative pairs using threshold value 0.5. Finally, we defined the score by weighting more importance on negative pair as shown on equation 2. If the authors have negative pair, we penalized them by subtracting weight multiply by number of negative pair to prevent incorrect prediction. We emphatically proved that weight with value of 2 achieved highest performance.

$$\ell(i,j) = -\log\left(\frac{\exp\left(\mathrm{sim}(z_i, z_j)/\tau\right)}{\sum_{k=1}^{2N} \nVdash_{[k\neq i]} \exp\left(\mathrm{sim}(z_i, z_k)/\tau\right)}\right) \quad (2)$$

4. Results

4.1 Experimental Setup

For our experiment, we used two different data sets. First data set is consisted of Google Code Jam code. The advantage of Google Code Jam dataset is that it allows us to test our model in controlled environment. The authors are given limited time to design the code to achieve same goal, and the level of difficulty increases as move on to the next stage. Therefore, we can easily observe the distinct coding style in the controlled environment setting. Second data set is consisted of real malware program data set. It contains decompiled source code from malware program. This dataset is considered to have more noise since we don't have any access to environment controlling factors. We sampled this code in three different ways to evaluate our performance in different settings. First, we sampled using random sampling method to sample 10 authors data set. Second, we used code from different family of malware to sample data thus we can collect codes that shares less features. Lastly, we manually sampled code that shares less overlap, which is going to be the hardest data set to evaluate. There are three different models that we used for evaluation. First model is consisted of simple classification header without applying contrastive learning. Second model used contrastive learning with generating positive pairs using combinations. Last model is using contrastive learning and using SimCSE method to generate the positive pairs. Furthermore, we also compared the result between function level and file level learning for each models. Moreover, we claim that our results are robust since we used cross validation with 10 folds to evaluate our method.

4.2 Google Code Jam

We took experiments on Google Code Jam data set with 10 and 100 authors to observe the result on our baseline model.

The reason for using this Google Code Jam data set is because this data is consisted of stable codes, which is good for starting point to evaluate our baseline method. Therefore, we started our experiment with Google Code Jam data set to test our baseline method, which is consisted of simple classification header and Code T5 encoder. As we can observe on figure 2, with our simple approach, we could obtain 76% accuracy on 10 authors data set. However, it was hard to classify 100 authors for our baseline method.

4.3 Real World Malware Data Set

We learned that our baseline method with simple classification header couldn't solve 100 authors data set. Therefore, we need better approach to handle complex task thus we applied our novel function level training process. The large language models has common input length limit of 512 tokens. Therefore, it is hard to put entire malware source code within given length limit. Our novel function level training process partition the original code in function level, so we do not loss our data. We claim that function level training process is more stable and robust since we not only input all of our data but also it is easier for our code language model to learning the data representation in function level.

We ran the experiments in three different settings, and each setting has different level of hardness for classifying. The random sampled file is our baseline, different family file is considered as medium level, and less overlap file is considered as the hardest file to classify the author. As we can observe on the table 2, the accuracy for random sampled file is highest with 67%, different family file has second highest accuracy with 47%, and as we expected the less overlap file has lowest accuracy with 35%. Furthermore, the function level training process improved the random sampled file with 3% and less overlap file with 4% improvement. However, the function level training could not improve the accuracy for different family file. Therefore, we further applied concept of contrastive learning to cover this up.

4.4 Real World Malware Data Set with Contrastive Learning

As we can see on the table 3, before applying contrastive learning, the accuracy for less overlap file was only 35%, but the contrastive learning improved maximum of 34% if we apply both function level training process and contrastive learning. Moreover, the different family file had 47% accuracy without applying contrastive learning, but the contrastive learning improve to 74%. The effects of contrastive learning in random sampled file was not significant, and we are assuming that our baseline was suitable for solving this file. Next, we compared the effectiveness of function level training process versus original file level training. We can clearly see that applying both contrastive learning and function level training significantly improved overall accuracy on different data sets. For random sampled file, we could improve the about 8%. For different family file, the function level improved 7% and obtained 74% accuracy. The less overlap file has improvement of 2%, which is considered as less improvement compared to other files. However, the performance of less overlap file improved from 35% to 69%, so improvement of 2% from file level learning to function level learning is reasonable.

5. Conclusion

We first experimented with Google Code Jam data set to observe our performance for baseline method. The result shows us that solving complex task with only classification header is tough. Therefore, we moved on to developing our novel function level training process, which uses partition of entire code to avoid loosing data by truncation due to input length limit. We experimented our function level training process on real world data set with three different sampling method to evaluate our result in various settings. However, only applying function level training method did not significantly improved the performance thus we came out with contrastive learning. By applying both contrastive learning and function level training method, we could obtain maximum improvement of 34% from less overlap file. Therefore, we concluded that applying both function level training approach and contrastive learning is effective to solve code authorship task. Our future work will be further improving the performance using advanced feature extraction method such as AST.

6. Future Work

We are planning to apply proposed method from SimCSE to generate postive pairs using dropout method to furthher improve accuracy (Gao, Yao, and Chen 2021). Furthermore, we are going to input AST from decompiled code to guide our model to learn the coding style effectively.

7. Acknowledgment

This research is based upon work supported by Purude Undergraduate Summer Research Fellowship program (SURF). Special thanks to professor Tan and Nan for giving us helpful advice during project.

8. Personal Thoughts About SURF

This research program was really helpful for me to learn the concept of natural language processing and its application to solving malware authorship task. Even though my mentor Nan was away from the campus, he was really helpful by answering my questions using email.

References

Abuhamad, M.; Rhim, J.-s.; AbuHmed, T.; Ullah, S.; Kang, S.; and Nyang, D. 2019. Code authorship identification using convolutional neural networks. *Future Generation Computer Systems*, 95: 104–115.

Alsulami, B.; Dauber, E.; Harang, R.; Mancoridis, S.; and Greenstadt, R. 2017. Source code authorship attribution using long short-term memory based networks. In *Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I 22*, 65–82. Springer.

Antoniol, G.; Fiutem, R.; and Cristoforetti, L. 1998. Using metrics to identify design patterns in object-oriented software. In *Proceedings Fifth International Software Metrics Symposium. Metrics (Cat. No. 98TB100262)*, 23–34. IEEE.

Buczak, A. L.; and Guven, E. 2015. A survey of data mining and machine learning methods for cyber security intrusion detection. *IEEE Communications surveys & tutorials*, 18(2): 1153–1176.

Burrows, S.; Uitdenbogerd, A. L.; and Turpin, A. 2009. Application of information retrieval techniques for source code authorship attribution. In *Database Systems for Advanced Applications: 14th International Conference, DASFAA 2009, Brisbane, Australia, April 21-23, 2009. Proceedings 14*, 699–713. Springer.

Caliskan, A.; Yamaguchi, F.; Dauber, E.; Harang, R.; Rieck, K.; Greenstadt, R.; and Narayanan, A. 2015. When coding style survives compilation: De-anonymizing programmers from executable binaries. *arXiv preprint arXiv:1512.08546*.

Chen, T.; Kornblith, S.; Norouzi, M.; and Hinton, G. 2020. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, 1597–1607. PMLR.

Elenbogen, B. S.; and Seliya, N. 2008a. Detecting outsourced student programming assignments. *Journal of Computing Sciences in Colleges*, 23(3): 50–57.

Elenbogen, B. S.; and Seliya, N. 2008b. Detecting outsourced student programming assignments. *Journal of Computing Sciences in Colleges*, 23(3): 50–57.

Frantzeskou, G.; Stamatatos, E.; Gritzalis, S.; and Katsikas, S. 2006. Effective identification of source code authors using byte-level information. In *Proceedings of the 28th international conference on Software engineering*, 893–896.

Gao, T.; Yao, X.; and Chen, D. 2021. Simcse: Simple contrastive learning of sentence embeddings. *arXiv preprint arXiv:2104.08821*.

Katz, D. S.; Ruchti, J.; and Schulte, E. 2018. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 346–356. IEEE.

Koschke, R.; Falke, R.; and Frenzel, P. 2006. Clone detection using abstract syntax suffix trees. In *2006 13th Working Conference on Reverse Engineering*, 253–262. IEEE.

Krsul, I. 1994. Authorship analysis: Identifying the author of a program (Technical report no. CSD-TR-94-030). *The COAST project, Department of computer science, Purdue University*.

MacDonell, S. G.; Gray, A. R.; MacLennan, G.; and Sallis, P. J. 1999. Software forensics for discriminating between program authors using case-based reasoning, feedforward neural networks and multiple discriminant analysis. In *ICONIP'99. ANZIIS'99 & ANNES'99 & ACNN'99. 6th International Conference on Neural Information Processing. Proceedings (Cat. No. 99EX378)*, volume 1, 66–71. IEEE.

Shankland, S. 2003. Sco sues big blue over unix, linux. *CNET News. com*.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, Ł.; and Polosukhin, I. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

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

Wisse, W.; and Veenman, C. 2015. Scripting dna: Identifying the javascript programmer. *Digital Investigation*, 15: 61–71.