Binary Malware Attribution using LLM Embeddings and Topological Data Analysis

Kincaid MacDonald[1,†], Ajai Ruparelia[1,†], Boomer Rogers[2,†], Adonis Bovell[1] and Branden Stone[1,*]

[1]Georgia Tech Research Institute
[2]Georgia Institute of Technology

Abstract

In malware authorship attribution, easily obtained, expressive, stylistically salient features are scarce. Often the only data source is the binary malware executable; this has traditionally constrained static analysis to producing simple statistics on the decompiled binary. Dynamic analysis can obtain more descriptive features by running the malware executable in a sandbox, but this is both resource-intensive and has obvious risks. In this work, we introduce two new sources of expressive static features which, when combined, achieve classification accuracies competitive with leading dynamic analysis models, and set a new state of the art for static malware authorship attribution. Our features stem from a hypothesis that distinctive stylistic coding features are present *at* or *around* the level of the function. To capture features at the function level, we use Large Language Model (LLM) embeddings of decompiled source code. To capture features at the function level, we use a Graph Neural Network, trained end-to-end on call graphs to identify stylistically-salient geometric features of the program architecture. We demonstrate that this new paradigm of graph learning combined with LLM code-embeddings can easily be extended to encompass and enhance existing featurizations. We further employ Topological Data Analysis to illustrate that the topological features of the call graph reveal stylistic indicators, which significantly aid in the task of code attribution.

Keywords

Code Attribution, Large Language Model, Topological Data Analysis, Graph Neural Networks

1. Introduction

What is the shape of code – and can it reveal a telltale fingerprint of the *coder*? Stylometry has a rich history within source code, where the programmer's identity can be revealed by quasi-linguistic features like variable names, spacing conventions, and syntax [1, 2, 3]. But in many areas, as with malware, source code is unavailable, and stylometry must be performed directly on binary executables. Here, the usual arsenal of linguistic stylometers falls flat. Traditional techniques are entirely dependent on the quality of the binary decompilation and are effectively left hunting for scraps - like file headers, import tables, and careless comments not removed during compilation [4, 5].

These challenges for traditional static analysis of binary malware samples motivate today's leading commercial solution: dynamic malware analysis. Here, the malware is run within a sandbox, which converts its behavior - API calls, network requests - into features used in downstream classification [4, 6]. Unfortunately, malware files are sometimes able to detect when they're being run in a sandbox and mask their behavior accordingly. Additionally, running hostile code on one's computers, even within a sandbox, is inherently risky.

In this work, we propose a new type of static malware analysis based on the topology and geometry of the underlying program – i.e. the *shape* of the code. We show that information about the shape alone, extracted from the program's call graph, can achieve reasonable performance on binary malware

CAMLIS'24: Conference on Applied Machine Learning for Information Security, October 24–25, 2024, Arlington, VA
*Corresponding author.
†These authors contributed equally.
Kincaid.MacDonald@gtri.gatech.edu (K. MacDonald); Ajai.Ruparelia@gtri.gatech.edu (A. Ruparelia); Herman.Rogers@gatech.edu (B. Rogers); Adonis.Bovell@gtri.gatech.edu (A. Bovell); Branden.Stone@gtri.gatech.edu (B. Stone)

classification. When we augment the call graph with code embeddings that indicate the relative proximity in feature space of nodes on the graph, our model achieves a new state-of-the-art for static binary malware analysis. In short, our novel contributions are as follows:

- demonstrate how Topological Data Analysis (TDA) on call graphs can extract distinguishing structures and signatures of code;
- present a novel method based on Large Language Model (LLM) embeddings that extract stylistic features from decompiled code;
- introduce a pipeline integrating LLM and TDA features into Graph Neural Network (GNN) architecture to produce a state-of-the-art static malware attribution model.

This paper is structured as follows: First, we describe our 'function-level' hypothesis that motivates our architecture design (see Figure 1). We then profile our dataset and its processing pipeline, before detailing our novel stylistic features: LLM code embeddings, and persistent homology across the call graph. We finally describe our models, their training, and their performance.

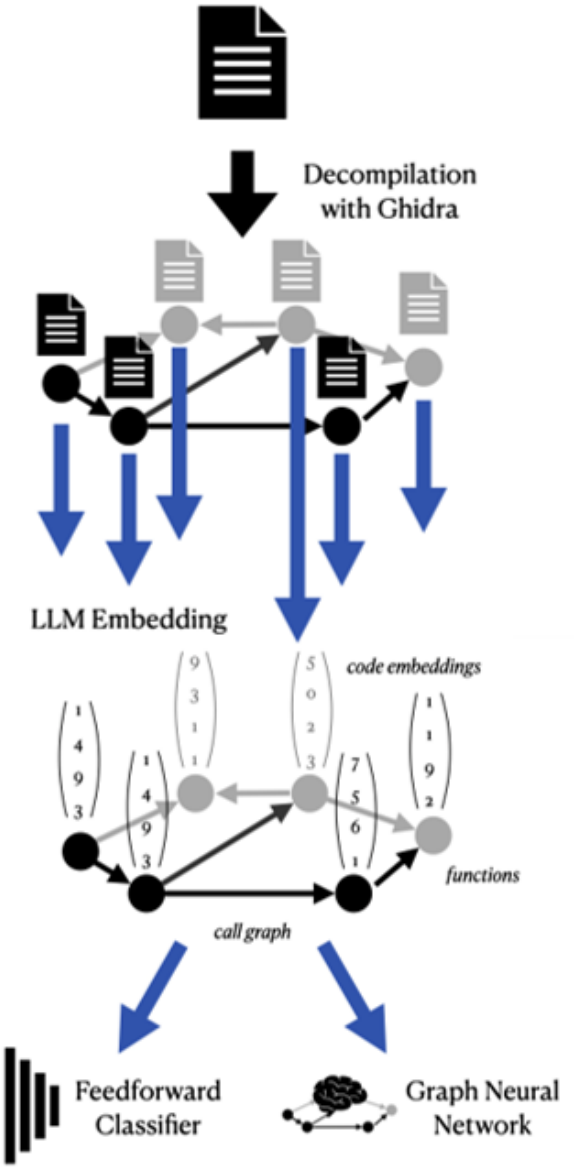

Figure 1: Outline of pipeline. Binaries are decompiled with Ghidra and a call graph is formed. An LLM embedding model is used for code embeddings at each node to function-level features. Graphs are then passed to training environments such as GNN or feedforward classifiers.

2. The Function-level Hypothesis

In any program, there are two primary stylistic influences: the preferences of the programmer and the demands of the program. The programmer may favor specific constructs, such as using for loops instead of while loops, switches instead of if statements, or dictionaries instead of objects. Moreover, programmers often have unique ways of defining functions; one may use several small functions, while another might achieve the same goal with a single, larger function.

Additionally, the program imposes global design constraints related to its structure and functionality. For example, ransomware coerces victims into involuntary actions, such as demanding money through extortion, using techniques like encryption and blockers. Unlike spyware, ransomware doesn't need to intercept/modify OS functions or APIs to be effective [7]. In the context of malware, different Advanced Persistent Threat (APT) groups often specialize in different types of malicious attacks (e.g. ransomware or espionage of the electric utility sector [8]), each imposing distinct strategies and constraints on the malware architecture. Conversely, the lower-level features, such as individual lines of code, are influenced by the programming language's syntax and style guidelines.

The question then arises: where can we observe a programmer's style most effectively? We hypothesize that a programmer's style is most evident at the function level. Functions strike the balance between global design constraints and local syntactic constraints, making them ideal candidates for this analysis. Moreover, good programming practice emphasizes making functions simple and modular, which allows complex problems to be decomposed into smaller, manageable parts. This decomposition process enables an in-depth and precise analysis of the programmer's unique approach and style.

In this paper, we aim to identify a programmer's style by focusing on the function level, rather than the global structure dictated by program goals or the local syntax influenced by language and style guides. We hypothesize that the function level, encompassing both the contents of functions and their interconnectivity, provides the ideal scale for this analysis. By examining functions and their interactions, we demonstrate the feasibility of identifying a unique "fingerprint" or "signature" of a team or developer in the context of malware attribution.

3. Dataset Overview & Feature Extraction

To ensure the reproducibility and comparability of our approach with common benchmarks, we utilize a publicly accessible dataset [9] consisting of over 3,500 malware samples attributed to 12 Advanced Persistent Threat (APT) Groups.. We exclude 721 samples consisting of container files and 20 samples which our automation could not open. Incidentally, this increases consistency, while reducing imbalance across our dataset resulting in a total of 2,853 samples across 12 distinct APT Groups (see Table 1).

Table 1
Sample Distribution by APT Group.

	Samples
APT 1	404
APT 10	228
APT 19	32
APT 21	87
APT 28	168
APT 29	263
APT 30	164
Dark Hotel	268
Energetic Bear	128
Equation Group	395
Gorgon Group	335
Winnti	381

3.1. Feature Extraction

To process our dataset, we leverage Ghidra [10], a software reverse engineering (SRE) framework created and maintained by the National Security Agency, and develop a comprehensive analysis pipeline that automates the extraction and processing of key features from each malware sample. Our pipeline is structured to provide insights at both the structural program level and the granular function level, facilitating the generation of novel features that capture the stylistic nuances of different APT groups. The main steps in our analysis pipeline (Figure 2) are as follows:

3.1.1. Function-Level Feature Extraction

For each malware sample, we begin by extracting function-specific features, such as:

- **Number of Input Parameters**: The count of input parameters for each function.
- **Function Size**: The size of the function measured by the number of instructions or lines of code.

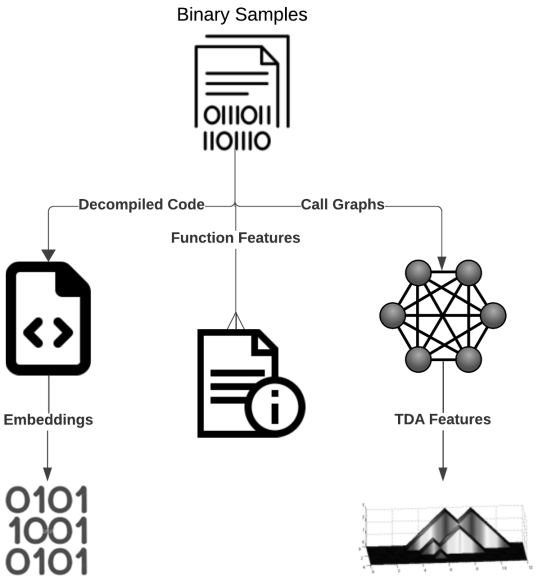

Figure 2: Feature extraction pipeline. For each binary sample, three types of features are harvested. LLM embeddings of functions, function features (e.g. number of input parameters or return type), and TDA landscapes.

- **Return Type**: The data type of the value returned by the function.
- **Number of Variables**: The count of variables for each function

3.1.2. Call Graph Construction

Next, we construct a call graph for each sample:

- **Node Representation**: Each node in the graph represents a function. The node attributes include the function-specific features extracted in the previous step.
- **Edge Representation**: Directed edges between nodes represent function calls, illustrating the relationships and dependencies between functions within the sample.
  The call graph not only serves as the basis for our GNN structure but also allows us to build the TDA features described in Section 4.

3.1.3. Function Decompilation and Embedding Generation

We decompile each function to obtain its raw C code:

- **Decompilation**: Using Ghidra's decompiler, we convert binary instructions back into its C representation.
- **String Representation**: The decompiled C code is stored as a string, allowing for further text-based analysis.
- **Embedding Creation**: These string representations are used to create embeddings via LLMs, enabling the analyses and comparisons of code stylistic features. See Section 4.1.

3.1.4. Dataset Splitting

To ensure robust evaluation, we divide the processed dataset into training, validation, and test sets:

- **Split Ratio**: We use an 80/10/10% split for training, validation, and testing datasets respectively. This approach ensures that our models are trained on a diverse set of samples while being evaluated on unseen data to assess generalization performance.

3.2. Output

The output of our pipeline consists of detailed representations of each malware sample, including function-level features, call graphs, and decompiled function strings. These outputs serve as the foundation for generating novel features that capture the unique characteristics of different APT groups. In the following section, we describe how we leverage these outputs to create LLM embeddings and TDA features, which are central to our methodology for distinguishing between the stylistic signatures of malware authors.

4. Novel Stylistic Features

4.1. LLM Embeddings of Decompiled Code

To use our decompiled source code in downstream tasks, we first need to vectorize it. While code embeddings do not contain sufficient information to recover the *content* of the code, they are useful in defining a measure of similarity between different embedded samples. In particular, we assume two functions are similar if their code embeddings are near each other.

Code embeddings are traditionally produced with an off-the-shelf embedding model trained specifically on code, like CodeBERT [11]. But the large-language model revolution has also introduced a new state-of-the-art of text and code embedding. OpenAI's text-embedding-ada-002 [12] is such a model. Built on the pre-trained core of GPT 3.5 and fine-tuned with an embedding objective, it can embed both text and code into 1536-dimensional vectors in which Euclidean distance corresponds to conceptual similarity.

There are presently few open-source embedding models based on LLMs. Starencoder [13], based on Starcoder [14], a GPT-2 based model trained exclusively on code, is the only one of which we're aware, and neither Starcoder nor its embedding model claim to match OpenAI's offerings in performance. In internal testing on an authorship attribution task using the Google CodeJam dataset [15], OpenAI's text-embedding-ada-002 model achieved the best performance. Thus, that model was selected for use here. However, given the number of increasingly capable open-source LLMs (e.g. Meta's recently announced LLAMA 2 [16]) it's only a matter of time before some are fine-tuned for embedding which offer performance competitive with today's closed-source models.

4.2. Topological Data Analysis Features

Our initial research question was initially conceived around the question: *Can the 'shape' of code be used for stylometry?* TDA gives a powerful set of tools for measuring this shape, by describing the number and size of connected components, any loops in the data, and - if in a sufficiently complex program - perhaps even the existence of higher-dimensional features (see [17] for a introduction to the topic). Such topological landmarks are an example of features found around the function level. The degree of branching between functions and sub-functions or the length of loops formed between different pathways through the code all seem likely to reveal the style of its author.

To TDA, there are two choices to be made. First, we must turn the code into a graph. Next, we must select the appropriate vectorization of persistent homology on the graph. Motivated by our function-level hypothesis, we chose the call graph as the representation most likely to reveal high-level stylistic features. For our vectorization, we utilize a combination of two leading techniques: we used Carrière's *Heat Kernel Signatures* to create several persistence diagrams with different diffusion parameters [18], then vectorized them into Bubenik's *Persistence Landscapes* [19]. We concatenate and flatten the resulting representations into a single 4000-dimensional vector per malware sample. We refer to these as *Global TDA* features (see Figure 3).

Traditional TDA operates globally, describing the shape of the entire call graph. If more localized topological descriptors are required, one can instead perform TDA on collections of subgraphs, creating node features which encode the local neighborhood topology. This is the approach of *Local TDA*. Our function-level hypothesis suggests that such local descriptors may better encode stylistic information,

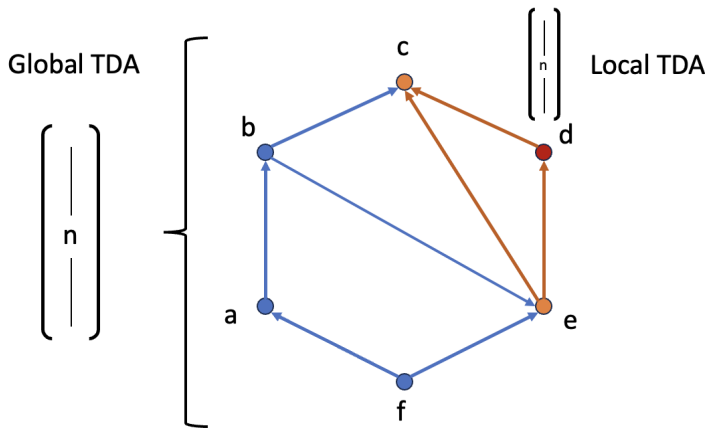

Figure 3: Global and Local TDA creating n-dimensional vectors. Global TDA features represent the entire graph while local TDA features are node features and calculated using the k-hop neighborhood of each node. Here $k = 1$ at node d. Both features are calculated with the heat kernel filter signature as in [18].

so in addition to global TDA features, we compute local TDA features on each call graph. We form k-hop subgraphs around each node, and apply the same combination of Heat Kernel Signatures and Persistence Landscapes to obtain 4000-dimensional local topology features for each node. For example, in Figure 3, the local TDA of node d uses a 1-hop subgraph containing nodes c, d, and e.

5. Methods & Results

As a baseline comparison we choose a random forest model using binary features processed by fuzzy hashing and NATO[1] encoding to achieve 89% accuracy on the same dataset [6]. As far as the authors know, this is the current state of the art on the APT dataset. The hashing algorithm used for the best accuracy is called `impfuzzy` [20]. This fuzzy hashing tool only considers the import API and not the body of the binary. Our approach considers significantly different aspects of the binary and is outlined in Figure 1.

5.1. Feedforward Classifier

We first tested the usefulness of both LLM and TDA features with a simple feedforward classifier. This provides a baseline for how useful each of our features is in predicted authorship, and whether the LLM embeddings and TDA features complement each other – whether they capture different descriptions of an author's style, and whether they can be combined to boost classification accuracy.

To this end, we designed a fully-connected deep neural network consisting of three submodules. One module takes a summary of the LLM embeddings as input (starting in 1536-dimensional space as mentioned above), passes it through ten linear layers with Leaky ReLU activation's, and outputs a dimensionally-reduced embedding of the LLM features. The second model does the same for the flattened TDA features. The final module concatenates the resulting embeddings as input to a four-layer classifier. We train the network with Cross Entropy loss and the Adam optimizer.

One subtlety here is that this is a *file-level* classification problem, while our LLM embeddings are *function* level features. Different malware samples have different numbers of functions, hence we need some manner of summarizing the embeddings of n functions into a single vector of unvarying size. Here, we copy a technique from the graph neural network literature [21]. Rather than performing a single pooling operation (e.g. mean pooling), we pool the features according to four statistical moments: obtaining the mean, median, skew, and kurtosis across each of the 1536 dimensions of the LLM embeddings. We concatenate the results and use this as input to our LLM-embeddings module.

[1] https://www.nato.int/cps/en/natohq/news_150391.htm

Table 2
Accuracy by Model & Features Included.

Model	LLM Embeddings	Global TDA	Acc.
Feedforward Classifier		✓	73%
Feedforward Classifier	✓		85%
Feedforward Classifier	✓	✓	88%
Fuzzy Hashing [6]			**89%**

When trained with both LLM and global TDA features, this simple classifier achieves an accuracy of 88% and is comparable to the current state of the art random forest model using fuzzy hashing (see Table 2). This difference could be attributed to the number of samples dropped from the dataset. In the case of [6], as `impfuzzy` only considers import API, it failed to hash 729 samples due to the lack of portable executable (PE) headers.

We further analyzed the relative importance of our features by retraining the classifier while omitting either the LLM features and its accompanying module, or the TDA features and its accompanying module (see Table 2). The global TDA features, trained alone, achieved an accuracy of 73%. The LLM embeddings, trained alone, achieved a higher accuracy of 85%. This could be due to the fact that LLM embeddings are coming from the decompiled source code and have more information about the program. However, the topological descriptors of the call graph performing at 73% seems to indicate that the shape of code is a useful indicator of the author's style.

Analyzing the predictions made by each independently trained classifier, found that of the malware samples incorrectly classified by the LLM-Embedding-trained model, the global-TDA-trained model classified 22% correctly. That is, most of the time, when the models disagree, the global-TDA-trained model is wrong. But a fourth of the time, it knows something the LLM-Embedding-trained doesn't. Reasoning from this, one would expect that in the best case, combining the global TDA and Embedding features would augment the LLM-Embedding-trained model's accuracy by about 5%. Indeed, this (Table 2) is what we find as the model trained on both features simultaneously achieves near state-of-the-art accuracy (around 88%), versus 84% from the LLM embeddings alone.

Admittedly, the four-moment summarization of our LLM embedding features is crude. It can highlight global variance in each of the embedding dimensions – perhaps detecting, e.g., when an author has many functions that perform a similar function – but lacks the knowledge of whether those functions call each other. A Graph Neural Network combines both sources of knowledge while enabling a learnable summarization that can highlight features relevant for classification.

5.2. Graphical Neural Networks

We based our GNN design on the molecular classification network described in Veličković's Graph-Attention Networks [22]. Structurally, our GNN is a representation of a malware sample's call graph where a node is a function and edges signify a call from one function to another. Additionally, nodes contain attributes consisting of the LLM Embedding of a function and various aspects of the function's metadata such as the number of inputs, function size and return type. It uses four Graph Attention Layers (employing Brody's "GAT v2" enhancement [23]) interspersed with normalization layers and Leaky ReLU activations, and followed by global Mean Pooling layer and a multi-layer feedforward classifier. We again train with cross-entropy loss using the Adam optimizer and a learning rate of 0.0001. We use 4 concatenated attention heads on the first three GAT layers, followed by 6 pooled heads on the final.

Within 100 training epochs, our GNN achieves 92% training accuracy using LLM embeddings and 93.22% on the holdout test data (Table 3). This exceeds the state-of-the-art 89% for static malware analysis [6], and is competitive with open-source dynamic analysis techniques with 95% accuracy on the same dataset [24].

We further added global and local TDA features to determine if they contain any stylistic indicators

not captured by the GNN. Table 3 details the results of adding these features. While local TDA features (with LLM embeddings) out performed everything on the validation set. Training with the LLM features alone achieved the best accuracy (93.22%). Given these results, it seems that these TDA features do not contain any extra information that can be learned by a GNN.

Table 3
Ablation study of augmenting the GNN with TDA or Local TDA features.

Features Used	Validation Accuracy	Test Accuracy
LLM embeddings	92.2%	**93.22%**
with Global TDA	92.5%	92.69%
with Local TDA	**94.0%**	91.15%
with Global & Local TDA	91.4%	88.76%

6. Conclusion

Our results indicate that while function-level analysis is effective, global features also play a crucial role. The use of TDA features alone achieved a 73% accuracy, highlighting that global structural features carry significant attributional information. In the malware domain, the specific goals and functions of malware can be closely tied to their authors, indicating that function and attribution may be highly correlated. This highlights the importance of a multifaceted approach.

Our GNN, which integrates both structural features (i.e. call graph) and function-level embeddings (LLM embeddings), outperforms other configurations and achieves state-of-the-art for static binary malware analysis. This multifaceted approach underscores the importance of integrating various levels of abstraction to capture the nuances of programming style and function.

The main findings of our research show that function-level analysis, enhanced by LLM embeddings, and global structural analysis, together lead to higher accuracy in malware attribution. Our GNN, which integrates these features, significantly outperforms configurations that use either feature alone. This result suggests that both the detailed stylistic elements and the broader structural aspects are crucial for accurate attribution, filling the gap in existing static analysis techniques.

7. Future Directions

To further enhance the accuracy of our approach, several improvements can be considered. While call graphs are inherently directed, our Graph Attention layers do not account for directional information. A GNN designed for directed graphs, like Perlmutter's MagNet [25], may be able to mine this additional source of information. In addition, while call graphs provide valuable structural insights, other representations such as Control Flow Graphs, Abstract Syntax Trees, or Program Dependence Graphs could offer additional layers of information. Integrating these representations may provide a more holistic view of the code, albeit with increased complexity in processing and analysis. Finally, we hope that future work will replicate our results using open-source LLM-based embedders like Starencoder, enabling this pipeline to be used in situations in which one can't send the decompiled source code to commercial servers.

A critical next step is to evaluate the resilience of our method to real-world challenges, such as malware obfuscation. Techniques that artificially flatten call graphs, encrypt functions, or otherwise complicate decompilation need to be tested against our model to ensure its robustness. While we suspect our GNN might be less susceptible to such manipulations compared to traditional static techniques, the LLM embeddings' dependency on decompilation quality remains a concern.

As it stands, our model is deliberately minimal, ignoring features traditionally employed for malware attribution such as API calls, linguistic comment analyses and runtime profiling. This approach allows us to better understand the efficacy of our novel features. Given the effectiveness demonstrated by

these features, future work could integrate our GNN's output into a broader set of malware features and apply downstream classification, potentially surpassing even the current state-of-the-art in dynamic analysis.

In conclusion, our research advances the field of static malware analysis by demonstrating the efficacy of combining global and function-level features within a GNN framework. This approach not only enhances attribution accuracy but also opens new avenues for incorporating additional features and representations. By addressing the above-identified limitations and exploring further enhancements, future studies can build upon our findings to develop even more robust and accurate methods for malware attribution, thereby strengthening cybersecurity defenses.

Acknowledgments

We are extremely grateful to the Georgia Tech Research Institute (GTRI) Graduate Research Internship Program (GRIP)[2] for supporting this effort during the Summer of 2023 and beyond. Further, thanks to Bob Wright for reviewing the work and providing valuable feedback.

References

[1] H. Ding, M. H. Samadzadeh, Extraction of java program fingerprints for software authorship identification, Journal of Systems and Software 72 (2004) 49–57.

[2] A. Caliskan-Islam, A. Narayanan, R. Harang, C. Voss, R. Greenstadt, A. Liu, F. Yamaguchi, De-anonymizing programmers via code stylometry, in: Proceedings of the 24th USENIX Security Symposium, 2015. URL: https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-caliskan-islam.pdf.

[3] B. Alsulami, E. Dauber, R. Harang, S. Mancoridis, R. Greenstadt, 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, Springer, 2017, pp. 65–82.

[4] S. Alrabaee, M. Debbabi, L. Wang, A survey of binary code fingerprinting approaches: Taxonomy, methodologies, and features, ACM Computing Surveys 55 (2022) 19:1–19:41. URL: https://dl.acm.org/doi/10.1145/3486860. doi:10.1145/3486860.

[5] H. S. Anderson, P. Roth, EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models, ArXiv e-prints (2018). arXiv:1804.04637.

[6] M. Kida, O. Olukoya, Nation-state threat actor attribution using fuzzy hashing, IEEE Access 11 (2023) 1148–1165. doi:10.1109/ACCESS.2022.3233403.

[7] D. Javaheri, M. Hosseinzadeh, A. M. Rahmani, Detection and elimination of spyware and ransomware by intercepting kernel-level system routines, IEEE Access 6 (2018) 78321–78332.

[8] MITRE, Mitre att&ck groups, https://attack.mitre.org/groups/, 2024. Accessed: 2024-07-09.

[9] C. Boot, Apt malware dataset, https://github.com/cyber-research, 2019. Accessed: December 12, 2023.

[10] N. S. Agency, Ghidra, https://ghidra-sre.org/, 2023. Accessed: December 12, 2023.

[11] 'Microsoft', https://github.com/microsoft/CodeBERT, 2023. Accessed: December 12, 2023.

[12] OpenAI, https://platform.openai.com/docs/guides/embeddings, 2023. Accessed: December 12, 2023.

[13] 'BigCode', https://huggingface.co/bigcode/starencoder, 2023. Accessed: December 12, 2023.

[14] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca,

[2] https://grip.gtri.gatech.edu/

M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, H. de Vries, StarCoder: may the source be with you! (2023). URL: http://arxiv.org/abs/2305.06161. arXiv:2305.06161 [cs].

[15] J. Petrík, Google code jam dataset, https://github.com/Jur1cek/gcj-dataset, 2008-2021. Accessed: December 12, 2023.

[16] 'Meta', https://ai.meta.com/llama/, 2023. Accessed: December 12, 2023.

[17] F. Chazal, B. Michel, An introduction to topological data analysis: fundamental and practical aspects for data scientists, Frontiers in artificial intelligence 4 (2021) 667963.

[18] M. Carriere, F. Chazal, Y. Ike, T. Lacombe, M. Royer, Y. Umeda, Perslay: A neural network layer for persistence diagrams and new graph topological signatures, in: S. Chiappa, R. Calandra (Eds.), Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics, volume 108 of *Proceedings of Machine Learning Research*, PMLR, 2020, pp. 2786–2796. URL: https://proceedings.mlr.press/v108/carriere20a.html.

[19] P. Bubenik, Statistical topological data analysis using persistence landscapes, Journal of Machine Learning Research 16 (2015) 77–102. URL: http://jmlr.org/papers/v16/bubenik15a.html.

[20] S. Tomonaga, Classifying malware using import api and fuzzy hashing – impfuzzy –, https://blogs.jpcert.or.jp/en/2016/05/classifying-mal-a988.html, 2016. Accessed: 2024-07-09.

[21] A. Tong, F. Wenkel, K. MacDonald, S. Krishnaswamy, G. Wolf, Data-driven learning of geometric scattering networks, 2020. arXiv:2010.02415.

[22] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, Y. Bengio, Graph Attention Networks, International Conference on Learning Representations (2018). URL: https://openreview.net/forum?id=rJXMpikCZ.

[23] S. Brody, U. Alon, E. Yahav, How attentive are graph attention networks?, arXiv preprint arXiv:2105.14491 (2021).

[24] C. Boot, Supervised learning for state-sponsored malware (2019). URL: http://www.cs.ru.nl/E.Poll/papers/MalwareStateAttribution2019.pdf.

[25] X. Zhang, Y. He, N. Brugnone, M. Perlmutter, M. Hirn, Magnet: A neural network for directed graphs, Advances in neural information processing systems 34 (2021) 27003–27015.