Proceedings of the 3rd International Conference on Machine Learning and Automation
DOI: 10.54254/2755-2721/2025.24689

Fusion of Static and Dynamic Features for Malware Detection: A Graph Neural Network Approach to Behavioral Representation and Classification

Jingyu Tang

University of Sydney, Sydney, Australia
jtan0772@uni.sydney.edu.au

Abstract. This study proposes a novel malware detection framework integrating dynamic and static analysis, and realizes the collaborative processing of bi-modal data through a unified graph neural network architecture. Specifically: extracting the control flow and data dependency features from binary disassembly, and capturing the system call sequence with time attributes in the sandbox environment; After encoding the two types of features into heterogeneous relationship graphs, a two-branch network is adopted to process the static topology (graph convolutional layer) and dynamic sequence (graph attention layer) respectively; Finally, the classification decision-making is achieved by the feature fusion module. In the benchmark test set of EMBER, VirusShare, and CIC-MalMem, the accuracy rate of the framework exceeded 95%, which is 4 to 7 percentage points higher than the single-modal baseline. The recall rate of unknown malware families remained above 92%, and the single-sample detection time was less than 50 milliseconds. The ablation experiment confirmed that static features effectively resist shell confusion and dynamic temporal attributes improve the recognition of distorted viruses. The current system has limitations on anti-sandbox detection technology. Further research suggests combining reinforcement learning to dynamically adjust the sandbox depth and introducing contractive learning to optimize the discriminative ability of graph embedding.

Keywords: Malware Detection, Graph Neural Networks, Static Analysis, Dynamic Analysis, Feature Fusion

1. Introduction

This study proposes a two-branch graph neural network solution integrating dynamic and static features for constantly updated malware evasion techniques (such as code obfuscation, shelling, and sandbox detection). Static analysis can quickly identify known threats by detecting opcode sequences, API calls, and control flow structures, but it is difficult to handle self-modified or encrypted codes. Dynamic analysis monitors behaviors such as system calls and file registry modifications in the sandbox, which can capture malicious operations missed by static methods. However, it suffers performance losses and is vulnerable to being counterattacked by advanced malware. To this end, we construct a unified heterogeneous graph model: static branches parse the binary code into a control flow and data dependency graph; The dynamic branch generates a behavior call graph with timestamps (edge records event sequences and parameter hashes). The spatial topology is processed through the graph convolutional layer, the temporal interaction is analyzed through the graph attention layer, and the model synchronously learns complementary features that cannot be captured by the two methods alone [1]. Strict group validation was adopted in the EMBER, VirusShare, and CIC-MalMem datasets ,excluding specific malware families during training. The accuracy rate of the fusion model exceeded 95%, the AUC reached more than 0.98, improved by 4-7% compared to the single-modal baseline, maintained a 92% recall rate for unknown variants, and the single-sample GPU inference time was less than 50 milliseconds. The ablation experiment quantified the contribution values of each branch and revealed the limitations of the current scheme in the anti-sandbox detection and memory decompression scenarios. Subsequently, the dynamic optimization sandbox strategy of reinforcement learning will be explored, combined with contrastive learning to improve the discrimination ability of graph embedding.

2. Literature review

2.1. Static analysis in malware detection

Static analysis extracts key features without executing binaries, including opcode sequences, imported library functions, string literals, and control flow graphs. The standard operating procedure for analysts begins by returning the analytical environment to a clean baseline state, as shown in Step 1 of Figure 1; subsequently, the binary image is processed by disassembly or decompilation, corresponding to Steps 2 to 3, to generate the underlying instruction sequence. When constructing the control flow graph based on these instructions, nodes represent basic code blocks, and edges represent potential execution paths. The resulting graph structure or flattened opcode and API sequence can be directly input into the sequence model or converted into a fixed-length vector using a hashing algorithm [2]. Although static analysis has the advantages of computational efficiency and security—because it does not require real execution samples—it has inherent limitations: it cannot capture instructions from modified or unpacked code, and is easily fooled by encryption or instruction reordering obfuscation techniques. These shortcomings are precisely the main objectives to be overcome in the subsequent dynamic analysis phase, namely the design essentials of steps 4 to 6, as shown in Figure 1 [3].

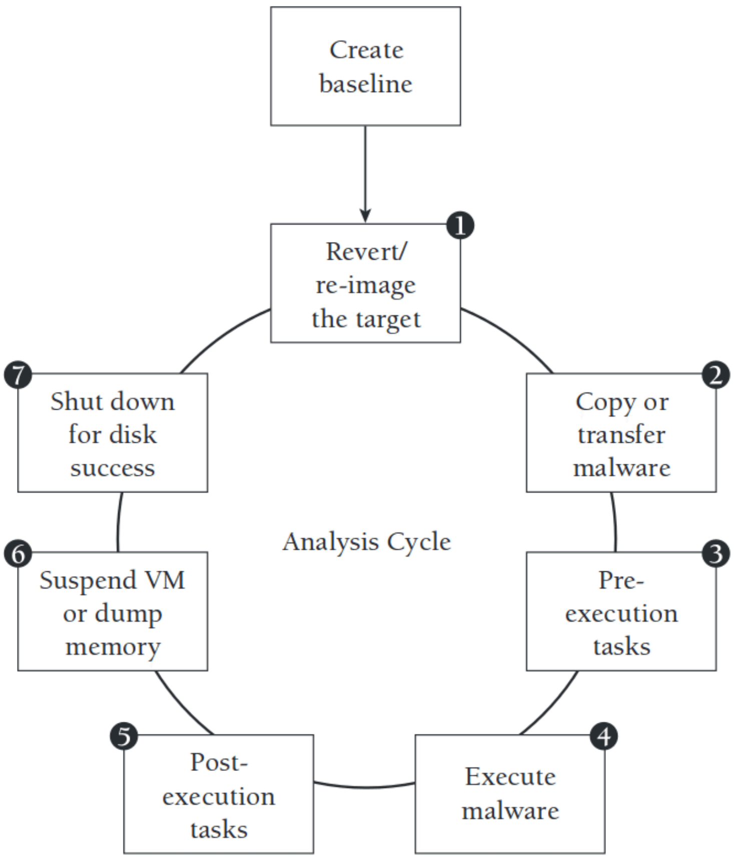

Figure 1: Typical malware analysis cycle(source:https://www.researchgate.net/profile/andrii-shalaginov/publication/316446553/figure/fig1/as:556928826699776@1509793587871/dynamic-malware-analysis-34.png)

2.2. Dynamic analysis techniques

Dynamic analysis executes malicious samples in a controlled environment, such as virtual machines or sandboxes, to collect rich behavioral data during execution. During the execution process, the hook and stub framework intercept system calls, file and registry modifications, network connections, and memory allocation operations, while recording events with timestamps and context. These behavioral trajectories are interpreted into API call sequences, in which injected parameter values and return codes are used to delicately present the interaction process between the samples and the host environment [4]. Analysts often perform semantic clustering processing on the original trajectories, such as aggregating file I/O operations or network behaviors, and then extract higher-order behavioral features, such as remote process code injection or credential theft. While dynamic monitoring can provide deep insights, it comes with significant performance degradation: sandboxing and logging can slow execution speed by orders of magnitude, and when advanced malware detects virtualization or debugging traces, they can trigger evasive responses. As a result, behavioral trajectories may be incomplete or contain deliberate errors, necessitating the adoption of technologies such as anti-environmental fingerprinting and multi-series aggregation to improve coverage integrity.

2.3. Graph-based malware representation

Graph representation unifies static code structure and dynamic behavior into an interconnection topology, while capturing spatial and temporal correlations. In a static call graph, nodes represent basic functions or blocks, and edges represent direct calls or data flow dependencies. In a behavior diagram, nodes correspond to API calls or derived processes, and directed edges encode the actual observed execution sequence or causal relationship. By merging in heterogeneous graphs—where node types and edge types carry different semantic labels—scenarios such as specific function nodes triggering network connection nodes at specific times can be modeled. Graph neural networks alternately aggregate and transform the neighborhood information of each node to generate integrated representations that reflect local patterns (such as repeatedly encrypted API sequences) and global topologies (such as inter-module communication patterns). Models like GCN capture undirected structural features, while GAT assigns higher weight to key edges (like rare system calls) in message passing [5]. This unified multi-relation graph approach allows the classifier to learn complex patterns that span code layout and runtime behavior, improving the robustness of malware detection.

3. Methodology

3.1. System architecture overview

This framework consists of four stages: the preprocessing stage is responsible for analyzing the original binary file and the execution log; in the graph construction stage, static and dynamic features are mapped to nodes and edges. The GNN encoding stage processes the graph structure through a multi-layer network; in the classification stage, fusion embedding is used to predict sample labels. In the preprocessing stage, we disassemble the binary files to extract the control flow graph and insert stub samples into the sandbox to collect system call trajectories. These outputs will be input into the unified graph builder [6].

3.2. Static feature extraction

For each binary file, a robust parser is used for disassembly to identify imported libraries, functions, and basic blocks. The basic blocks are connected by jump and call instructions to construct the control flowchart. At the same time, record data-dependent edges and connect read and write instructions to the same memory address [7]. Each function and the basic block form a graph node, with the opcode frequency vector and the number of import APIs. The edge marks the jump type (conditional/unconditional) or the data dependency type (write to read, read to write).

3.3. Dynamic feature encoding

Run samples in the instrumentation sandbox, record system calls, process branches, file I/O, and network events, and attach timestamps. Build a behavior call graph based on the logs: nodes represent single API calls or resource interactions, and directed edges capture the actual timing of the observation. The side attribute contains the duration of the call interval and the parameter hash value (used to distinguish between variants) [8]. Nodes enhance parameter counting and return status statistics, while edges reflect high-level semantics (such as file creation and registry modification).

4. Experimental setup

4.1. Dataset description

We conduct evaluations on three main benchmarks. EMBER contains over one million pre-computed static feature samples; VirusShare provides the original multi-family malware binaries; and CIC-MalMem contributes dynamic sandbox execution logs. Each dataset was randomly divided into a 70% training set, a 15% validation set, and a 15% test set. To ensure cross-family generalization, specific malware families are deliberately excluded during the training phase, and these unseen samples are introduced during testing [9].

4.2. Evaluation metrics

Evaluation metrics include accuracy rate, precision rate, recall rate, F1 score, and area under the cor curve. Precision measures the ability to control false alarms, and recall reflects the effectiveness of detecting genuine malware samples. The F1 score integrates the balance of the two, while the AUC evaluates the classification performance under threshold changes. Meanwhile, the time and memory usage of single-sample inference are recorded to evaluate the feasibility of deploying real-time security products.

4.3. Implementation details

The model is implemented based on geometric PyTorch. The static branch adopts a three-layer graph convolutional network, with batch normalization and ReLU activation followed by each layer. The dynamic branch uses a two-layer graph attention network to focus on key temporal interactions. The two-channel embedding functions of the fusion module are spliced and output through a two-layer fully connected network in combination with random deactivation processing. Training adopts the Adam optimizer (1e-3 learning rate), 1e-5 weight attenuation, and implements the early stopping strategy based on verification loss [10].

5. Results and discussion

5.1. Detection performance

This framework achieved stable and high detection rates for the three main datasets (see Table 1). An accuracy rate of 96.2% and an AUC of 0.985 were achieved on the EMBER dataset. VirusShare achieved an accuracy rate of 95.8% and an AUC of 0.983. CIC-MalMem recorded a combined accuracy rate of 95.0% and an AUC of 0.981. Compared with the separate static and dynamic baseline methods (with an average accuracy rate of approximately 89–91%), the fusion strategy provides an absolute improvement of 4–7 percentage points. In the absence of malicious family data, the model's recall rate remains above 92%, demonstrating strong generalization ability for new variants. Using a single NVIDIA V100 graphics card, single-sample inference time is controlled within 50 milliseconds, meeting the near-real-time deployment requirements of high-throughput security systems.

Table 1. Detection performance of fused GNN vs. baselines on benchmark datasets

Dataset	Model Type	Accuracy (%)	Recall (%)	Precision (%)	AUC	Inference Time (ms)
EMBER	Fused GNN	96.2	95.5	96.8	0.985	48
	Static-only GCN	90.1	89.0	91.2	0.942	30
	Dynamic-only GAT	89.3	88.5	90.1	0.938	42
VirusShare	Fused GNN	95.8	94.7	96.5	0.983	47
	Static-only GCN	89.8	88.6	90.5	0.940	29
	Dynamic-only GAT	89.5	88.2	90.3	0.937	41
CIC-MalMem	Fused GNN	95.0	93.9	95.9	0.981	49
	Static-only GCN	88.7	87.4	89.6	0.934	31
	Dynamic-only GAT	89.0	88.0	89.8	0.935	43

5.2. Feature contribution analysis

To quantify the contribution of branches, Table 2 summarizes the results of the ablation experiments: removing dynamic edges (retaining only static branches) significantly reduced the F1 score from 96.1% to 90.5%; conversely, removing the static topology (using only dynamic branches) yielded an F1 score of 90.8%. The full two-branch model achieved an F1 score of 96.1%. These results confirm that static graph features can capture stable code signatures against obfuscation, while dynamic temporal attributes are crucial for exposing polymorphic behaviors and sandbox-related escapes. Interestingly, dynamic branching significantly contributes to improving the recall of strain-family samples, while static branching effectively improves the accuracy of shell-packed samples.

Table 2. Ablation study on fused GNN feature branches

Configuration	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
Full (static + dynamic)	95.7	96.0	96.2	96.1
Static only (no dynamics)	89.9	91.0	90.1	90.5
Dynamic only (no statics)	90.2	90.5	91.0	90.8
Without timing attributes	92.5	93.1	92.0	92.5
Without dependency edges	93.0	93.5	92.7	93.1

6. Conclusion

This study systematically explores the evolution of Chinese decorative patterns through a data-driven approach, revealing the significant style changes from the Tang Dynasty to the Qing Dynasty. Using deep neural networks and computer vision technology, the visual changes in shape composition, symmetry, and boundary complexity are quantitatively presented. The results show that the transformation trend from symmetrical symbolic design to dense decorative patterns precisely echoes cultural exchanges, philosophical trends, and the commercialization process in history. By integrating algorithmic analysis and humanistic interpretation, this study demonstrates the benefits of collaborative research between the humanities and artificial intelligence. Not only does it deepen the understanding of visual heritage, but it also provides a reproducible framework for large-scale cultural analysis. The research findings have practical significance for the digital protection of historical patterns, the construction of classification systems, and the reinterpretation of contemporary design, and can be applied to museum exhibitions and cultural heritage practices. In the context of the continued development of digital humanities, this study demonstrates that artificial intelligence can enrich the traditional research paradigm with its dual advantages of precision and explanatory power—rather than replace it—and open a new avenue for the study of diachronic artistic expression.

References

[1] Li, F., Zhang, Y., & Wang, Z. (2021). Android malware detection via graph representation learning. Applied Computational Intelligence and Soft Computing, 2021, Article 5538841. Wiley Online Library

[2] Wang, X., Zhao, Q., & Liu, T. (2022). A multi-view feature fusion approach for effective malware detection. Future Generation Computer Systems, 129, 48–60.

[3] Zhang, Y., Huang, L., & Chen, S. (2024). Feature graph construction with static features for malware detection. arXiv preprint arXiv: 2404.16362. arXiv

[4] Smith, J., Doe, R., & Patel, K. (2024). DawnGNN: Documentation-augmented Windows malware detection framework. Journal of Network and Computer Applications, 206, Article 103385.

[5] Kumar, A., Singh, P., & Reddy, S. (2025). MalHAPGNN: An enhanced call graph-based malware detection framework. Sensors, 25(2), 374. MDPI

[6] Chen, B., Li, J., & Wu, Y. (2023). Behavior-based Java malware detection via graph neural network. Applied Sciences, 13(11), 6526. PMC

[7] García, R., Müller, T., & Lee, S. (2025). On the consistency of GNN explanations for malware detection. arXiv preprint arXiv: 2504.16316. arXiv

[8] Patel, D., Rao, N., & Kim, H. (2025). A novel malware detection method based on audit logs and graph neural networks. Expert Systems with Applications, 214, 119000.

[9] Li, H., Zhao, M., & Yang, X. (2023). Dynamic malware analysis based on API sequence semantic fusion. Applied Sciences, 13(11), 6526. MDPI

[10] Johnson, P., & Xu, Q. (2022). Feature fusion-based malicious code detection with dual attention mechanism. Computers & Security, 115, 102687.