Mitigating Distribution Shift in Graph-Based Android Malware Classification via Function Metadata and LLM Embeddings

Ngoc N. Tran[1], Anwar Said[1], Waseem Abbas[2], Tyler Derr[1], Xenofon D. Koutsoukos[1],

[1]Department of Computer Science, Vanderbilt University
[2]Department of Systems Engineering, The University of Texas at Dallas
ngoc.n.tran@vanderbilt.edu, anwar.said@vanderbilt.edu, waseem.abbas@utdallas.edu,
tyler.derr@vanderbilt.edu, xenofon.koutsoukos@vanderbilt.edu

Abstract

Graph-based malware classifiers can achieve over 94% accuracy on standard Android datasets, yet we find they suffer accuracy drops of up to 45% when evaluated on previously unseen malware variants from the same family—a scenario where strong generalization would typically be expected. This highlights a key limitation in existing approaches: both the model architectures and their structure-only representations often fail to capture deeper semantic patterns. In this work, we propose a robust semantic enrichment framework that enhances function call graphs with contextual features, including function-level metadata and, when available, code embeddings derived from large language models. The framework is designed to operate under real-world constraints where feature availability is inconsistent, and supports flexible integration of semantic signals. To evaluate generalization under realistic domain and temporal shifts, we introduce two new benchmarks: MalNet-Tiny-Common and MalNet-Tiny-Distinct, constructed using malware family partitioning to simulate cross-family generalization and evolving threat behavior. Experiments across multiple graph neural network backbones show that our method improves classification performance by up to 8% under distribution shift and consistently enhances robustness when integrated with adaptation-based methods. These results offer a practical path toward building resilient malware detection systems in evolving threat environments.

Introduction

Android malware continues to evolve rapidly, posing persistent challenges to the reliability and robustness of automated detection systems. Recent advances in graph-based learning have led to promising approaches for malware classification by representing applications as *function call graphs* (FCGs), where nodes correspond to individual functions and directed edges represent invocation relationships (Yang et al. 2021; Harang and Rudd 2020; Joyce et al. 2021). Graph Neural Networks (GNNs) (Kipf and Welling 2017; Hamilton, Ying, and Leskovec 2018) applied to these graphs enable the modeling of structural and behavioral patterns that are indicative of malicious behavior (Li et al. 2022). Compared to image-based approaches like byteplot representations (Freitas, Duggal, and Chau 2022), graph-based methods provide a more interpretable, semantically structured, and functionally meaningful abstraction.

Graph-based malware classification gained momentum with the release of MalNet (Freitas et al. 2021), a large-scale dataset comprising over 1.5 million Android malware samples represented as FCGs. A smaller, balanced subset known as MalNet-Tiny was released to support benchmarking. Since then, MalNet-Tiny has become a standard testbed for evaluating GNN architectures, with state-of-the-art models achieving over 94% accuracy (Rampášek et al. 2023; Shirzad et al. 2023). However, these results often assume standard data splits. Recent work (Wu et al. 2024) shows that performance drops significantly when models are evaluated on samples from malware families not seen during training, highlighting a critical weakness under distribution shift.

A key limitation underlying this brittleness is the lack of semantic information in MalNet-Tiny. During graph construction, all function-level metadata—such as names, types, and code—was removed to avoid exposing potentially sensitive artifacts (Freitas et al. 2021). While this decision aimed to reduce reverse-engineering risks, we argue it unnecessarily restricts the model's ability to reason about functional behavior. First, the full malware binaries used to generate these graphs are publicly available (Allix et al. 2016; Freitas, Duggal, and Chau 2022). Second, prior work has shown that semantic features can be integrated without compromising security (Anderson and Roth 2018; Yang et al. 2021).

Moreover, we hypothesize that this omission impairs generalization for FCGs, whose edges denote information flow rather than structural similarity. In such graphs, node semantics are essential for effective message passing and representation learning (Hamilton, Ying, and Leskovec 2018; Hu et al. 2020). Without them, models tend to memorize local structures that do not transfer well across malware families.

Motivated by this insight, we propose an enhanced attributed graph construction framework tailored for Android malware classification under distribution shift. Our method enriches each function call graph (FCG) with semantic node features extracted directly from the malware's code structure. Specifically, we extract a set of lightweight, widely available metadata features—including function names, method signatures, access flags, instruction statistics, and Android-specific behaviors—commonly used in malware analysis (Anderson and Roth 2018; Yang et al. 2021). When decompiled source code is available, we embed function bodies using a large language model

(LLM), providing dense representations of behavioral semantics. These semantic vectors are combined with standard structural features and injected as node attributes for downstream graph neural networks (GNNs). Importantly, our framework is designed for real-world deployment scenarios, where semantic features may be partially missing. To address this, we introduce three collation strategies—Trim, Zero, and Prune—that transform partially defined graphs into a consistent format suitable for learning.

To evaluate robustness under distribution shifts, we construct and release two new benchmark variants of MalNet-Tiny. In *MalNet-Tiny-Common*, training and test samples are drawn from overlapping malware families, while in *MalNet-Tiny-Distinct*, test samples originate from families unseen during training (Wu et al. 2024). Across multiple GNN backbones, we find that our proposed semantic graph construction significantly improves classification accuracy under both settings. These results highlight the critical role of semantic information in improving the generalization of graph-based malware classifiers in realistic settings.
Our key contributions are as follows:

- We identify and empirically demonstrate the brittleness of state-of-the-art graph-based Android malware classifiers under distribution shift, where accuracy drops sharply on test samples drawn from previously unseen malware families.
- We propose a semantic feature enrichment framework for Android function call graphs that augments structural graphs with interpretable metadata and LLM-derived embeddings. To address real-world data quality challenges—where semantic features may be inconsistently available—we introduce three collation strategies that robustly transform incomplete graphs into usable inputs for graph neural networks.
- We construct and release two new benchmark datasets, *MalNet-Tiny-Common* and *MalNet-Tiny-Distinct*, designed to evaluate classifier robustness under intra-family and cross-family distribution shifts.
- Across multiple GNN architectures, we empirically show that our approach significantly improves classification performance under distribution shift and boosts robustness when combined with adaptation-based training.

Preliminaries

MalNet and Graph-Based Malware Data

MalNet (Freitas et al. 2021) is a large-scale dataset of Android malware samples, where each sample is represented as a FCG extracted using AndroGuard (Desnos and Gueguen 2018). Each node in the FCG corresponds to a function, and edges indicate function calls. Labels are derived from VirusTotal reports and unified into a hierarchy of malware *families* and *types* using Euphony (Hurier 2025). A cleaned and balanced subset, MalNet-Tiny, was introduced to support manageable training and evaluation, with each class defined by a unique (family, type) pair. A more detailed discussion on MalNet and an example of this labeling structure is shown in Appendix.

MalNet-Tiny	MNT-Common	MNT-Distinct
addisplay / kuguo	addisplay / dowgin	spr / lootor
adware / airpush	adware / startapp	clicker+trj / dowgin
benign / benign	benign / benign	riskware / nandrobox
downloader / jiagu	downloader / *mixed*	malware / *mixed*
trojan / artemis	trojan / deng	spyware / *mixed*

Tab. 1: Malware labels of MalNet-Tiny variants.

Distribution Shift in Malware Classification

We consider a standard supervised classification setting, where each malware sample is denoted by X, and its associated label by Y. The training data is drawn from a source distribution $\mathcal{D}$, while test-time samples may be drawn from a different target distribution $\mathcal{D}'$. *Distribution shift* then refers to the condition where the distributions of training and testing data differ, leading to a performance drop during model evaluation. While there are many types of distribution shifts (Quiñonero-Candela et al. 2022), we focus only on settings where labels for malwares do not change, i.e. $\mathbb{P}_{(X,Y)\sim\mathcal{D}}(Y|X) = \mathbb{P}_{(X,Y)\sim\mathcal{D}'}(Y|X)$. Among them, *covariate shift* happens when malwares of the same labels are collected from different sources, meaning $\mathbb{P}_{\mathcal{D}}(Y) = \mathbb{P}_{\mathcal{D}'}(Y)$ but $\mathbb{P}_{\mathcal{D}}(X) \neq \mathbb{P}_{\mathcal{D}'}(X)$. *Domain shift* is a more general form of distribution shift where the training and test data come from completely different distributions, i.e. $\mathbb{P}_{\mathcal{D}}(Y) \neq \mathbb{P}_{\mathcal{D}'}(Y)$ and $\mathbb{P}_{\mathcal{D}}(X) \neq \mathbb{P}_{\mathcal{D}'}(X)$. *Temporal shift* is similar to covariate shift, but differs in that the distribution mismatch is caused by malwares changing over time.

Research Objectives

Malware classifiers must remain effective under distribution shift, as new variants often differ from prior data. Our goal is to build a single classifier that maintains high accuracy despite such shifts. To support this, we introduce two MalNet-Tiny-style datasets simulating covariate and domain shifts for robust evaluation, and propose a framework that enhances attributed graph construction, mitigates data-quality issues, and incorporates distribution-aware graph learning.

Constructing Distribution-Shifted Datasets

Recall that each malware label in MalNet-Tiny is a malware *family/type* pair, where *family* is a broader category of malwares with similar characteristics, and *type* is a subcategory within a family. Leveraging this hierarchical labeling, we construct new datasets from the original MalNet to realistically simulate different distribution shifts from MalNet-Tiny: MalNet-Tiny-Common for covariate shift, and MalNet-Tiny-Distinct for domain shift. These datasets are curated to have these same properties for both compatibility and a fair comparison with the baselines. Malware families/types and their corresponding samples are selected to be as disjoint as possible, with the chosen labels listed in Tab. 1. We also intended to create a dataset for temporal shift, but were unable to do so due to MalNet having too few malwares belonging to MalNet-Tiny classes despite its large size. While we believe our proposed method is also robust to temporal shift, we leave this as a future work.

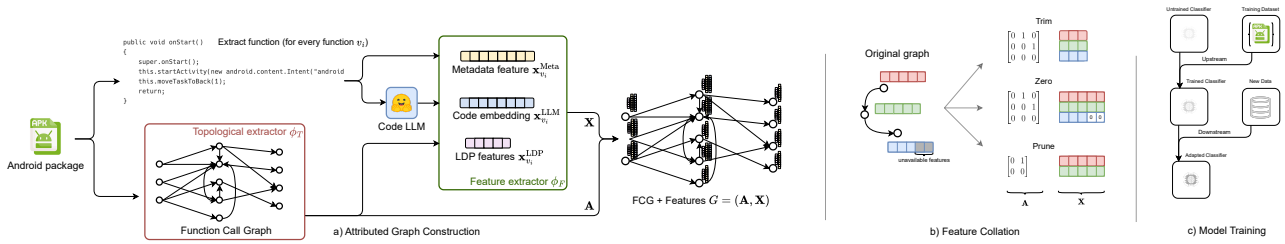

Fig. 1: Overview of the robust graph learning pipeline.

Covariate Shift: MalNet-Tiny-Common

We construct MalNet-Tiny-Common to have the same malware families but different malware types to MalNet-Tiny to simulate a covariate shift scenario. The label correlation between the two datasets let us alternatively interpret the training process as fitting malwares to their corresponding family labels instead of types, implying that a model trained on MalNet-Tiny can be used as-is to classify MalNet-Tiny-Common, but performance will drop due to the distribution shift. For the sampling process, we seed all random sampling with 0. We start by sampling types from the same families as MalNet-Tiny, and then sample 1,000 samples from each type. The two exceptions to the label selection are: *(i)* the unchanged benign class is sampled from a disjoint set of malware samples, and *(ii)* the *downloader* family included multiple malware types, as it does not have any other one type with 1,000 qualifying samples.

Domain Shift: MalNet-Tiny-Distinct

The construction process for MalNet-Tiny-Distinct is similar to MalNet-Tiny-Common, but with different malware families/types to simulate domain shift. This dataset is designed to test the ability of adaptation schemes to generalize to completely new malwares, and is expected to be more challenging than MalNet-Tiny-Common. However, even with the large size of the original MalNet, and to the best of our effort sampling data from as disjoint malware families/types as possible, the labels for MalNet-Tiny-Distinct are noisier than which of other variants, with more generic families and more mixed labels. The full list of malware families/types for all datasets is listed in the Appendix.

Methodology

We now present our proposed approach to robust malware classification, a complete pipeline to train a graph classification model for malware samples. We first reformulate the problem of malware classification as a graph classification task, and present our proposed method for constructing attributed graphs from malware samples. We then discuss data-quality challenges in the original MalNet that hinders the feature extraction process, and how we can mitigate them with our proposed collation schemes. Finally, we apply GNNs to learn robust representations of the malware graphs, and present training strategies to improve the model's robustness to distribution shifts.

Problem Formulation

We formally define malware classification as a graph classification task: For each malware sample x, we aim to construct an attributed graph $G = (\mathbf{A}, \mathbf{X})$, where $\mathbf{A} = \phi_T(x)$ is the adjacency matrix, and $\mathbf{X} = \phi_F(x)$ is the node feature matrix. This process involves a topology extractor ϕ_T that extracts the graph structure from the malware sample, and a node feature extractor ϕ_F that extracts a feature vector for each vertex in the graph; which we combined as ϕ. The goal is to construct a semantic feature extractor ϕ such that the trained graph classification model $\hat{F}$ performs well on both the training distribution $\mathcal{D}$ and a new testing distribution $\mathcal{D}'$:

Problem 1 (Robust Feature Extraction for Graph Malware Classification)**.** *Given malware sample x, the goal is to construct an expressive feature extractor $\phi(x)$ that returns the graph representations $G = \phi(x)$, such that the trained model $\hat{F}$ performs well on both the training, i.e. high $\mathbb{P}_{(x,y)\sim\mathcal{D}}[\hat{F}(G) = y]$, and new test distribution, i.e. high $\mathbb{P}_{(x,y)\sim\mathcal{D}'}[\hat{F}(G) = y]$.*

We now focus on how to construct the feature extractor $\phi = (\phi_T, \phi_F)$ for the malware sample x, and the full pipeline for training a robust graph malware classification model.

Attributed Graph Construction

In this section, we present our proposed method for constructing attributed graphs from malware samples, extending traditional works that only focused on structural features. Specifically, we focus on advanced feature extraction that we believe may aid in mitigating the distribution shift problem. Fig. 1 shows an overview of the feature construction process.

Topology Extractor We adhere to the original construction of MalNet for topology extraction, utilizing FCGs as the graph representation of malware samples. The FCG contains a node set $\mathcal{V}$ where each node v_i represents a function, and an edge set $\mathcal{E}$ where each edge e_{jk} denotes that function v_j calls function v_k during its execution. Mathematically, we define the topology extractor ϕ_T as a function that takes in a malware sample x and returns the adjacency matrix $\mathbf{A} = \phi_T(x) \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ of the FCG, where $|\mathcal{V}|$ is the number of nodes/functions, and each entry A_{jk} is 1 if $e_{jk} \in \mathcal{E}$, and 0 otherwise. The resulting graph is then $G = (\mathbf{A}, \emptyset)$, where the node feature matrix is empty.

Feature Extractor Existing methods (Freitas et al. 2021; Rampášek et al. 2023; Shirzad et al. 2023) typically use Local Degree Profile (LDP) (Cai and Wang 2022) for node feature extraction ϕ_F, which is a vector of the node's degree and its neighbors' degree statistics. As LDP focuses solely on the structural properties of the graph, when combined with the purely-structural FCG, it unnecessarily limits the model's ability to learn from the semantic properties of these function calls, which can be crucial for distinguishing between different types of malwares. Instead, we propose to extract node features directly from the corresponding functions, which can be used to better capture the behavior and characteristics of the malwares.

Aggregated Metadata Features. Inspired by EMBER (Anderson and Roth 2018), we construct a set of metadata features for each function, which are then aggregated into a single feature vector $\mathbf{x}_{v_i}^{\text{Meta}}$ for each node v_i. Features include *class* and *method names, method signatures, access flags, code length, bytecode statistics, instruction statistics, string statistics, and more*. In addition, we extract other Android-specific features such as *storage access, registry modifications, and in-memory code execution*. This feature construction scheme is designed to capture the essential characteristics of the functions in the FCGs, while not disclosing any sensitive information similar to the EMBER precedence. We refer the reader to the Appendix for detailed specifications.

LLM Source Code Embedding. From the Androguard analysis output, we can also obtain the decompiled Java source code of each function, which we use to extract additional features. We utilize CodeXEmbed (Liu et al. 2024), the only available code embedding model that has a large enough context window to handle the size of the source code, to embed the source code into a numerical vector. The model is pulled from the HuggingFace repository (Wolf et al. 2020), and inference was done through the provided HuggingFace API. This embedding better extracts the semantic meaning of the function than the aggregated metadata features alone, as it captures the function's behavior and purpose based on its implementation. We denote the resulting feature vector as $\mathbf{x}_{v_i}^{\text{LLM}}$ for each function v_i. Further details are in Appendix.

Concatenating Components. While function metadata alone has proven effective as a feature (Anderson and Roth 2018), incorporating code embeddings generated by LLMs provides a more comprehensive representation of a function's behavior. This provides additional context for the model to distinguish between functions with similar metadata. This insight leads us to include both types of features in our method. We also utilize the commonly-used LDP features, given its impressive baseline performance:

$$\mathbf{x}_{v_i}^{\text{LDP}} = [\deg(v_i), \min_{v_j \in \mathcal{N}(v_i)} \deg(v_j), \max_{v_j \in \mathcal{N}(v_i)} \deg(v_j), \operatorname{mean}_{v_j \in \mathcal{N}(v_i)} \deg(v_j), \operatorname{std}_{v_j \in \mathcal{N}(v_i)} \deg(v_j)], \tag{1}$$

for any node v_i, where $\deg(v_i)$ is the node degree, and $\mathcal{N}(v_i)$ is the neighborhood of v_i.

The three extracted feature vectors are concatenated to form the final d-dimensional node feature vector: $\mathbf{x}_{v_i} = \mathbf{x}_{v_i}^{\text{Meta}} \,\|\, \mathbf{x}_{v_i}^{\text{LLM}} \,\|\, \mathbf{x}_{v_i}^{\text{LDP}} \in \mathbb{R}^d$, for each node in the FCG. Putting it all together, we define the feature extractor ϕ_F as a function that takes a malware x and its the FCG adjacency matrix $\mathbf{A}$, and returns the node feature matrix $\mathbf{X} = \phi_F(\mathbf{A}, x) = [\mathbf{x}_{v_1} \ \dots \ \mathbf{x}_{v_{|\mathcal{V}|}}]^\top \in \mathbb{R}^{|\mathcal{V}| \times d}$, with each row vector $\mathbf{x}_{v_i}$ constructed as described above. As $\mathbf{A} = \phi_T(x)$ can be obtained from the malware sample x and thus can be omitted from the function signature, we henceforth omit it for simplicity. The final attributed graph is then:

$$G = \phi(X) = (\phi_T(x), \phi_F(x)) = (\mathbf{A}, \mathbf{x}). \tag{2}$$

Malware Data-Quality Challenges

While the topology extractor ϕ_T produces a well-defined graph structure $\mathbf{A}$ for each malware sample, constructing a high-quality node feature matrix $\mathbf{X} = \phi_F(x)$ presents nontrivial data-quality challenges. In particular, the reliability and availability of semantic features extracted from each function in the graph can vary substantially due to limitations in the underlying data. For any given malware sample, the number of extractable semantic features varies from function to function. Some nodes correspond to external or system-level functions (e.g., Android APIs or imported libraries) that lack accessible decompiled source code, making it impossible to compute certain semantic features such as code embeddings. As a result, the feature matrix $\mathbf{X} \in \mathbb{R}^{|\mathcal{V}| \times d}$, where each row $\mathbf{X}_{i,:} \equiv \mathbf{x}_{v_i} \in \mathbb{R}^d$ represents the feature vector for node v_i, is only partially defined, with some node feature entries missing due to limitations in static analysis or obfuscation. To formalize this, we define the set of *non-universal* feature dimensions:

$$\mathcal{C} = \{c \in \{1, \dots, d\} \mid \exists\, i \in \{1, \dots, |\mathcal{V}|\} \text{ s.t. } \mathbf{X}_{i,c} \text{ is missing}\}.$$

These are the feature types that are not consistently available across all nodes in this sample's graph. However, since the downstream GNN classifier $F(G)$ requires that all nodes in the attributed graph $G = (\mathbf{A}, \mathbf{X})$ share a consistent input dimension, we must address this structural inconsistency before training. To mitigate this challenge, we propose three node feature collation schemes designed to transform the partially defined $\mathbf{X}$ into a complete, uniform format compatible with GNN-based learning: *Trim*, *Zero*, and *Prune*.

***Trim*: Remove Non-Universal Feature Dimensions** We remove all feature dimensions that are not available for all nodes in the current sample. That is, we retain only the set of feature dimensions $\mathcal{F}$ that are defined across every node, i.e., $\mathcal{F} = \{1, \dots, d\} \setminus \mathcal{C}$. We then construct the trimmed feature matrix as follows:

$$\mathbf{X}_{\text{trim}} = \mathbf{X}_{:,\mathcal{F}} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{F}|}.$$

This ensures that every node has a complete and consistent feature vector. However, it discards any partially defined features, potentially removing useful information available for some functions.

***Zero*: Impute Missing Features with Zeros** We retain the full graph and all feature dimensions, and fill in any missing values with zeros. Let:

$$\mathcal{M} = \{(i, c) \in \{1, \dots, |\mathcal{V}|\} \times \mathcal{C} \mid \mathbf{X}_{i,c} \text{ is missing}\}$$

be the index set of node-feature pairs that are undefined. We define the zero-imputed feature matrix $\mathbf{X}_{\text{zero}} \in \mathbb{R}^{|\mathcal{V}| \times d}$ as:

$$\mathbf{X}_{\text{zero}}[i, c] = \begin{cases} \mathbf{X}[i, c] & \text{if } (i, c) \notin \mathcal{M}, \\ 0 & \text{if } (i, c) \in \mathcal{M}. \end{cases}$$

This approach ensures that every node retains a full-length feature vector, enabling direct compatibility with GNN input requirements. It defers to the model to learn whether zeroed features are informative or irrelevant.

***Prune*: Discard Nodes with Incomplete Feature Vectors** Unlike other strategies which retain the original graph topology, *Prune* modifies both the feature matrix and the graph structure: when a node has missing feature values, we remove it along with any edges connected to it. Formally, we first define the set of nodes with complete feature vectors $\mathcal{V}'$:

$$\mathcal{V}' = \{v_i \in \mathcal{V} \mid \forall c \in \{1, \ldots, d\},\ \mathbf{X}_{i,c} \text{ is defined}\}.$$

We then restrict both the feature matrix and the adjacency matrix to this subset. Our newly pruned feature matrix is:

$$\mathbf{X}_{\text{prune}} = \mathbf{X}_{\mathcal{V}',:} \in \mathbb{R}^{|\mathcal{V}'| \times d},$$

and the corresponding pruned graph topology is:

$$\mathbf{A}_{\text{prune}} = \mathbf{A}_{\mathcal{V}',\mathcal{V}'} \in \{0, 1\}^{|\mathcal{V}'| \times |\mathcal{V}'|}.$$

This operation yields the subgraph of G induced by the node set $(\mathcal{V}')$, thereby ensuring that the input graph provided to the GNN maintains structural consistency and contains fully-defined features. While preserving all feature dimensions, it might omit important contextual information.

Robust Graph Learning for Distribution Shifts

After making sure that all nodes in the FCG have the same number of features, we seek to leverage powerful GNNs to learn complex malware representations based on our attributed graphs: with each malware transformed into an attributed graph $G = (\mathbf{A}, \mathbf{X})$, it proceeds to be encoded into a condensed graph-level vector representation via a GNN $F(G)$. The goal now becomes to build models that remain robust when the test-time data distribution differs from the training distribution: a setting known as *distribution shift*. Distribution shift is especially problematic in malware classification due to evolving attacker behavior, new obfuscation strategies, and the emergence of novel malware families. To address this challenge, we decompose the learning process into two stages: *upstream training* on the source distribution $\mathcal{D}$, and *downstream adaptation* to the target distribution $\mathcal{D}'$.

Upstream Training In the upstream phase, we assume access to labeled training samples $(G, Y) \sim \mathcal{D}$, where each malware has been converted into an attributed graph. We train a GNN classifier F by minimizing the expected loss:

$$\hat{F} = \arg\min_{F} \mathbb{E}_{(G,Y)\sim\mathcal{D}} \left[\mathcal{L}(F(G), Y)\right]$$

This training step may involve supervised or self-supervised (pre)training and does not assume any knowledge of $\mathcal{D}'$.

Downstream Adaptation At test time, we apply $\hat{F}$ to data drawn from a different distribution $\mathcal{D}'$, which can result in degraded performance due to distribution shift. To improve generalization, we define a general adaptation framework that produces an adapted model $\tilde{F}$ by modifying $\hat{F}$ using data from $\mathcal{D}'$, categorized by the target labels' availability.

Test-Time Adaptation (TTA). TTA assumes access to *unlabeled* graphs from the target domain $\{G_j\}_{j=1}^{m} \sim \mathcal{D}'$, and adapts the model using these samples:

$$\tilde{F} = \mathcal{A}_{\text{TTA}}(\hat{F}; \{G_j\}_{j=1}^{m}). \tag{3}$$

These methods typically do not modify the entire model but update specific components such as normalization layers or classifier prototypes (Iwasawa and Matsuo 2021). TTA is appealing in scenarios where no new annotations are available.

Domain Adaptation (DA). DA assumes access to *labeled* target data $\{(G_k, Y_k)\}_{k=1}^{n} \sim \mathcal{D}'$ and adapts the model using both graphs and their labels:

$$\tilde{F} = \mathcal{A}_{\text{DA}}(\hat{F}; \{(G_k, Y_k)\}_{k=1}^{n}). \tag{4}$$

This setting supports stronger forms of adaptation, including full finetuning (Church, Chen, and Ma 2021), parameter-efficient updates (e.g., adapters) (Gui, Ye, and Xiao 2023; Han et al. 2024), or classifier replacement (Chen et al. 2020), though it requires additional labeling effort.

This modular framework cleanly separates the learning process into upstream representation learning and downstream adaptation. It supports a variety of adaptation methods that can be flexibly selected based on real-world constraints such as data availability or deployment cost. In our experiments, we instantiate this framework using both generic and graph-specific methods for TTA and DA.

Numerical Evaluation

To better understand the effectiveness of semantic features for malware FCGs, we propose a series of research questions that thoroughly evaluate the effects of our method:

RQ1: *How do semantic features affect model robustness?*

We address this question by comparing models trained with and without (baseline) semantic features, evaluated on the same testing distribution (i.e. high utility), and on a covariate-shifted testing distribution (i.e. high robustness).

RQ2: *Are all features necessary for improving robustness?*

We conduct an ablation study on the two components of our semantic feature construction across different collation schemes, and report how they affect model performance.

RQ3: *Do features work with existing adaptation methods?*

Given that our method takes a different approach to alleviating distribution shift, we evaluate how well it can be combined with and improve existing adaptation methods.

We design and conduct a series of experiments to answer these questions, giving insights into the characteristics of semantic features for malware FCGs, and report our findings.

Experimental Setup

Below, we concisely outline our experiment configurations. A more detailed version can be found in the Appendix.

GNN Architectures. Depending on the model architecture, different feature collation schemes may be better or worse. Therefore, we conduct our experiments with various GNN architectures, including GCN (Kipf and Welling 2017), GIN (Xu et al. 2019), GPS (Rampášek et al. 2023), and Exphormer (Shirzad et al. 2023).

Method	Features			GCN		GIN		GPS		Exphormer	
	Meta	LLM	LDP	Tiny	Cmn.	Tiny	Cmn.	Tiny	Cmn.	Tiny	Cmn.
Baseline / None			✓	$85.6_{0.08}$	$48.4_{0.17}$	$91.2_{0.05}$	$49.6_{0.10}$	$93.5_{0.27}$	$47.7_{0.50}$	$94.5_{0.05}$	$49.5_{0.05}$
Trim	✓	-	-	$92.9_{0.02}$	$56.5_{0.24}$	$91.9_{0.09}$	$47.7_{0.09}$	$94.0_{0.12}$	$51.7_{0.21}$	$94.2_{0.03}$	$57.3_{0.03}$
	✓	-	✓	$91.8_{0.06}$	$51.4_{0.18}$	$92.5_{0.11}$	$48.9_{0.13}$	$94.2_{0.14}$	$54.5_{0.24}$	$94.0_{0.03}$	$51.3_{0.11}$
Prune	✓	-	-	$92.3_{0.17}$	$52.0_{0.20}$	$93.4_{0.05}$	$51.5_{0.03}$	$94.1_{0.33}$	$49.8_{0.32}$	$93.2_{0.01}$	$52.3_{0.08}$
	✓	✓	-	$92.8_{0.11}$	$50.5_{0.04}$	$93.5_{0.00}$	$53.8_{0.06}$	$93.8_{0.35}$	$50.8_{0.39}$	$93.9_{0.05}$	$52.7_{0.06}$
	✓	-	✓	$94.9_{0.17}$	$51.9_{0.17}$	$93.8_{0.09}$	$52.4_{0.10}$	$94.7_{0.35}$	$54.5_{0.38}$	$93.6_{0.04}$	$50.6_{0.04}$
	-	✓	✓	$84.7_{0.14}$	$50.5_{0.17}$	$88.6_{0.09}$	$48.9_{0.19}$	$93.1_{0.28}$	$51.4_{0.37}$	$93.8_{0.01}$	$47.8_{0.12}$
	✓	✓	✓	$94.4_{0.05}$	$51.6_{0.12}$	$93.6_{0.08}$	$51.8_{0.05}$	$95.1_{0.03}$	$51.9_{0.17}$	$95.1_{0.01}$	$50.6_{0.06}$
Zero	✓	-	-	$93.5_{0.26}$	$50.9_{0.30}$	$93.3_{0.11}$	$51.8_{0.19}$	$94.4_{0.19}$	$51.9_{0.26}$	$94.6_{0.06}$	$50.5_{0.06}$
	✓	✓	-	$93.3_{0.16}$	$50.6_{0.20}$	$92.9_{0.07}$	$53.2_{0.08}$	$94.9_{0.24}$	$53.3_{0.39}$	$94.9_{0.05}$	$54.7_{0.08}$
	✓	-	✓	$92.4_{0.10}$	$49.6_{0.04}$	$93.6_{0.02}$	$52.1_{0.09}$	$94.7_{0.31}$	$55.8_{0.32}$	$94.6_{0.00}$	$54.1_{0.04}$
	-	✓	✓	$84.7_{0.14}$	$49.2_{0.22}$	$92.1_{0.11}$	$45.7_{0.09}$	$92.9_{0.18}$	$46.7_{0.49}$	$94.1_{0.03}$	$50.1_{0.09}$
	✓	✓	✓	$93.5_{0.07}$	$52.1_{0.17}$	$93.3_{0.11}$	$52.0_{0.10}$	$93.8_{0.34}$	$51.0_{0.50}$	$95.1_{0.01}$	$53.9_{0.05}$

Tab. 2: Models' accuracy on MalNet-Tiny and MalNet-Tiny-Common across different feature configurations. Top 5 highest values are highlighted in green, darker green represents higher accuracy; values lower than baseline are highlighted in red. Subscript denotes standard deviation over 3 independent runs.

Method	GCN		GIN		GPS		Exphormer	
	Tiny	Cmn.	Tiny	Cmn.	Tiny	Cmn.	Tiny	Cmn.
Baseline	85.6%	48.4%	91.2%	49.6%	93.5%	47.7%	94.5%	49.5%
Trim	91.8%	51.4%	92.5%	48.9%	94.2%	**54.5%**	94.0%	51.3%
Prune	**94.4%**	51.6%	**93.6%**	51.8%	**95.1%**	51.9%	**95.1%**	50.6%
Zero	93.5%	**52.1%**	93.3%	**52.0%**	93.8%	51.0%	**95.1%**	**53.9%**

Tab. 3: Test accuracy on MalNet-Tiny and MalNet-Tiny-Common with/without semantic features. Highest values are bolded.

Adaptation Methods. We select these methods such that they do not alter the upstream training process, and cover a wide range of approaches. For Test-time Adaptation (TTA), the selected baselines are Tent (Wang et al. 2021), T3A (Iwasawa and Matsuo 2021), and GTrans (Jin et al. 2023). For Domain Adaptation (DA) approaches, we evaluate with k-NN Probe (Chen et al. 2020), normal finetuning, and AdapterGNN (Li, Han, and Bai 2023).

Results & Analysis

In this section, we present the results of our experiments to answer the posed research questions.

RQ1: How do semantic features affect model robustness? Tab. 3 shows the results of our experiments on the original MalNet-Tiny dataset, and its covariate-shifted counterpart MalNet-Tiny-Common. We observe that most models trained with semantic features outperform the baseline model in all cases, with the only exception being *Trim*.

Finding 1. *Prune models achieve the highest accuracy for MalNet-Tiny, while Zero models achieve the highest accuracy for MalNet-Tiny-Common.*

This can be explained by that *Prune* contains only the nodes that have the full feature vector, forcing the model to learn from all available features on only the richest nodes. While this condensation of information can help the model perform better on the task at hand, it also leads to overfitting when the model learns to rely on non-universal features. In contrast, *Zero* models are trained on all nodes, including those with missing features. This sparsity allows the model to learn more robust representations, as it can still make predictions based on the available features when some are missing. This is similar to how dropout works, where randomly removing some features during training can help the model generalize better to unseen data (Srivastava et al. 2014).

As a result, we only use *Prune* and *Zero* collation schemes for later experiments; and we select the latter as the default configuration for our method. This is because our main goal is to improve model robustness, thus better performance on the covariate-shifted dataset is prioritized, leaving performance on the original dataset increased by a smaller margin.

RQ2: Are all semantic features necessary for improving model robustness? To further delve into the contribution of each of our method's components, we conduct an ablation study on the feature configurations. We select the commonly-used LDP features as our baseline, experiment with replacing or adding features, and measure the resulting differences. However, some feature combinations are not applicable, and thus do not appear in this ablation study:

- *Trim* does not work with LLM features: for all malwares, there exist some functions that do not have any code in the APK (e.g. API functions). As a result, code embeddings features are not available for all nodes, and thus will always be trimmed.
- *Prune* and *Zero* does not work with LLM features alone: if a malware contains no decompilable code, its output graph will not have any node features, and thus cannot be processed by any graph neural networks.

Tab. 2 shows the full results of our ablation study.

Finding 2. *For the full feature config. with either Prune or Zero, all models achieved higher results than the baseline.*

This strengthens our claim of effectiveness for semantic features with these two collation schemes. While different configs of *Zero* always achieve higher performance than baselines, we keep our default configuration to all features as it achieves the largest positive margin for Exphormer, which is our best-performing model and the hardest to improve.

Model	GCN	GIN	GPS	XFM
Tent	46.2%	49.6%	44.8%	48.0%
. . . + Prune	**49.3%**	**49.7%**	**48.4%**	47.4%
. . . + Zero	48.6%	48.9%	47.3%	**51.4%**
T3A	50.4%	50.8%	48.1%	49.7%
. . . + Prune	**52.2%**	52.2%	**51.9%**	50.1%
. . . + Zero	**52.2%**	**52.3%**	51.0%	**53.2%**
GTrans	50.6%	49.5%	**47.2%**	**51.7%**
. . . + Prune	34.1%	46.0%	47.0%	44.2%
. . . + Zero	**51.1%**	**50.6%**	43.6%	41.5%

Tab. 4: Accuracy on MalNet-Tiny-Common when combining TTA methods with semantic features. Highest values for each set are in bold. XFM is short for Exphormer.

Finding 3. *Source code feature extraction is challenging, as LLM features show similar to worse model performance.*

We believe that this phenomenon happens because our code embedding features are only designed for retrieval tasks. While they are good at capturing the semantics of the functions, they should not be solely relied on for its discriminative power. Our choice of LLM embedding model is subjected to code length constraint, heavily limiting our options. A better method to extract code semantics for classification is desired, but is out of the scope of this work.

Finding 4. *Omitting any features for Exphormer will result in a worse performance than baseline.*

With Exphormer adding augmented pseudo-edges to input graphs for better propagation, it can misdirect unwanted information during message passing, and thus is not guaranteed to be good for graph classification (Shirzad et al. 2023). As a result, feature density in *Prune* became detrimental to the model's performance; and in contrast, *Zero*'s sparsity allowed the model to learn representations that are robust to irrelevant information being propagated through the graph.

RQ3: Do semantic features work with existing adaptation methods? We first evaluate the effectiveness of our method when combined with TTA methods, which do not require any additional training data. Tab. 4 shows the results of our experiments on MalNet-Tiny-Common, which is a covariate-shifted dataset with the same malware families but different malware types. We observe:

Finding 5. *Our method consistently improves the performance of the adapted models across different generic TTA methods and architectures. Specifically, Prune works best with all models except for Exphormer.*

Interestingly, *Prune* works best with the majority of models for test-time adaptation approaches. On the other hand, *Zero* remains the best option for Exphormer, consistent with our previous findings. However, it is a different situation for graph-based TTA, where *Zero* works best for only traditional message-passing architectures. We hypothesize that directly applying GTrans to graph transformers is not effective, which was not tested in the original work.

Finetuning-based adaptation. For the methods that require additional training data, we report the results in Tab. 5:

Finding 6. *Our method consistently further improves the adapted models across different finetuning approaches.*

Method	GCN		GIN		GPS		Exphormer	
	Cmn.	Dst.	Cmn.	Dst.	Cmn.	Dst.	Cmn.	Dst.
k-NN Probe	74.1%	**92.1%**	77.9%	**89.1%**	69.2%	87.5%	75.9%	87.9%
. . . + Prune	**74.3%**	89.5%	**78.4%**	86.2%	76.7%	87.9%	78.2%	88.2%
. . . + Zero	71.9%	87.8%	78.3%	85.8%	**77.1%**	**88.2%**	**81.4%**	**90.6%**
Finetune	83.7%	94.0%	79.8%	94.1%	92.3%	95.9%	92.0%	96.6%
. . . + Prune	92.5%	**96.7%**	91.4%	**96.2%**	**95.6%**	**97.8%**	95.2%	**97.7%**
. . . + Zero	**93.7%**	**96.7%**	**92.2%**	95.7%	95.0%	97.4%	**96.1%**	97.4%
AdapterGNN	81.7%	94.7%	85.9%	**95.3%**	87.0%	95.3%	88.6%	96.1%
. . . + Prune	**89.4%**	**95.0%**	**87.9%**	95.1%	**90.6%**	95.4%	**93.6%**	95.3%
. . . + Zero	86.9%	94.2%	87.7%	93.0%	88.3%	**96.4%**	89.4%	**96.5%**

Tab. 5: Full dataset accuracy on MalNet-Tiny variants (Common, Distinct) of models trained on MalNet-Tiny after finetuning. Largest accuracy for each experiment set is in bold.

The only exception is the k-NN Probe, which does not benefit from our method when adapting to MalNet-Tiny-Distinct. Meanwhile, adding semantic features improves performance across almost all finetuning-based methods by up to 7.7% in classification accuracy. Overall, these results show that our semantic feature construction can be effectively combined with existing adaptation methods to enhance their robustness to distribution shift.

Related Work

Android Malware Graph. Most works on malware graphs also rely on FCGs for structure (Malhotra, Potika, and Stamp 2024). Older works are based on random walks (Ge et al. 2019), while some other adapted NLP methods to feature extraction (Feng et al. 2021, 2023). Other methods such as (Alasmary et al. 2019) extracted CFGs as the representations instead, and focused on the Internet-of-Thing domain. Meanwhile, (Zhang et al. 2022) took a different approach and extracted the abstract syntax tree for the malware graph.

Graph Distribution Shift. The effect of distribution shift in graphs can be attacked from many angles, with some are finetuning-based (Gui, Ye, and Xiao 2023; Li, Han, and Bai 2023), while others adapted domain adaptation techniques (Chen et al. 2022; Wang et al. 2022; Jin et al. 2023; Ju et al. 2023; Hsu et al. 2025). A more novel approach to graph adaptation is graph prompt tuning (Liu et al. 2023; Fang et al. 2024; Fu, He, and Li 2025); however, these methods typically require pretraining on a large corpus of graphs. We refer the interested readers to (Zhang et al. 2024) for further reading on these topics.

Conclusion

In this work, we proposed a method to generate semantic features for graph representations of Android malwares, which can be used to enhance the robustness of graph-based malware classification models against distribution shift, while also improving their upstream classification performance. We also introduced two new datasets, MalNet-Tiny-Common and MalNet-Tiny-Distinct, to evaluate the robustness of our method against covariate and domain shifts, respectively. We hope that our work will inspire future research on enriching input graphs for better model generalization, and that our contributed datasets will be useful for evaluating the effectiveness of new methods in this area.

References

Alasmary, H.; Khormali, A.; Anwar, A.; Park, J.; Choi, J.; Abusnaina, A.; Awad, A.; Nyang, D.; and Mohaisen, A. 2019. Analyzing and Detecting Emerging Internet of Things Malware: A Graph-Based Approach. *IEEE Internet of Things Journal*, 6(5): 8977–8988.

Allix, K.; Bissyandé, T. F.; Klein, J.; and Le Traon, Y. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, 468–471. New York, NY, USA: ACM. ISBN 978-1-4503-4186-8.

Anderson, H. S.; and Roth, P. 2018. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. arXiv:1804.04637.

Cai, C.; and Wang, Y. 2022. A simple yet effective baseline for non-attributed graph classification. arXiv:1811.03508.

Chen, G.; Zhang, J.; Xiao, X.; and Li, Y. 2022. GraphTTA: Test Time Adaptation on Graph Neural Networks. arXiv:2208.09126.

Chen, T.; Kornblith, S.; Norouzi, M.; and Hinton, G. 2020. A Simple Framework for Contrastive Learning of Visual Representations. arXiv:2002.05709.

Church, K. W.; Chen, Z.; and Ma, Y. 2021. Emerging trends: A gentle introduction to fine-tuning. *Natural Language Engineering*, 27(6): 763–778.

Desnos, A.; and Gueguen, G. 2018. Androguard documentation. *Obtenido de Androguard*.

Fang, T.; Zhang, Y.; Yang, Y.; Wang, C.; and Chen, L. 2024. Universal Prompt Tuning for Graph Neural Networks. arXiv:2209.15240.

Feng, P.; Ma, J.; Li, T.; Ma, X.; Xi, N.; and Lu, D. 2021. Android Malware Detection via Graph Representation Learning. *Mobile Information Systems*, 2021: 1–14.

Feng, P.; Yang, L.; Lu, D.; Xi, N.; and Ma, J. 2023. BejaGNN: behavior-based Java malware detection via graph neural network. *The Journal of Supercomputing*, 79(14): 15390–15414.

Freitas, S.; Dong, Y.; Neil, J.; and Chau, D. H. 2021. A Large-Scale Database for Graph Representation Learning. arXiv:2011.07682.

Freitas, S.; Duggal, R.; and Chau, D. H. 2022. MalNet: A Large-Scale Image Database of Malicious Software. arXiv:2102.01072.

Fu, X.; He, Y.; and Li, J. 2025. Edge Prompt Tuning for Graph Neural Networks. arXiv:2503.00750.

Ge, X.; Pan, Y.; Fan, Y.; and Fang, C. 2019. AMDroid: Android Malware Detection Using Function Call Graphs. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 71–77.

Gui, A.; Ye, J.; and Xiao, H. 2023. G-Adapter: Towards Structure-Aware Parameter-Efficient Transfer Learning for Graph Transformer Networks. arXiv:2305.10329.

Hamilton, W. L.; Ying, R.; and Leskovec, J. 2018. Inductive Representation Learning on Large Graphs. arXiv:1706.02216.

Han, Z.; Gao, C.; Liu, J.; Zhang, J.; and Zhang, S. Q. 2024. Parameter-Efficient Fine-Tuning for Large Models: A Comprehensive Survey. arXiv:2403.14608.

Harang, R.; and Rudd, E. M. 2020. SOREL-20M: A Large Scale Benchmark Dataset for Malicious PE Detection. arXiv:2012.07634.

Hou, S.; Fan, Y.; Zhang, Y.; Ye, Y.; Lei, J.; Wan, W.; Wang, J.; Xiong, Q.; and Shao, F. 2019. aCyber: Enhancing Robustness of Android Malware Detection System against Adversarial Attacks on Heterogeneous Graph based Model. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, CIKM '19, 609–618. New York, NY, USA: Association for Computing Machinery. ISBN 9781450369763.

Hsu, H. H.-H.; Liu, S.; Zhao, H.; and Li, P. 2025. Structural Alignment Improves Graph Test-Time Adaptation. arXiv:2502.18334.

Hu, W.; Fey, M.; Zitnik, M.; Dong, Y.; Ren, H.; Liu, B.; Catasta, M.; and Leskovec, J. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv preprint arXiv:2005.00687*.

Hurier, M. 2025. Euphony. https://github.com/fmind/euphony.

Iwasawa, Y.; and Matsuo, Y. 2021. Test-Time Classifier Adjustment Module for Model-Agnostic Domain Generalization. In Ranzato, M.; Beygelzimer, A.; Dauphin, Y.; Liang, P.; and Vaughan, J. W., eds., *Advances in Neural Information Processing Systems*, volume 34, 2427–2440. Curran Associates, Inc.

Jin, W.; Zhao, T.; Ding, J.; Liu, Y.; Tang, J.; and Shah, N. 2023. Empowering Graph Representation Learning with Test-Time Graph Transformation. In *The Eleventh International Conference on Learning Representations*.

Joyce, R. J.; Amlani, D.; Nicholas, C.; and Raff, E. 2021. MOTIF: A Large Malware Reference Dataset with Ground Truth Family Labels. arXiv:2111.15031.

Ju, M.; Zhao, T.; Yu, W.; Shah, N.; and Ye, Y. 2023. GraphPatcher: Mitigating Degree Bias for Graph Neural Networks via Test-time Augmentation. arXiv:2310.00800.

Kipf, T. N.; and Welling, M. 2017. Semi-Supervised Classification with Graph Convolutional Networks. arXiv:1609.02907.

Li, C.; Cheng, Z.; Zhu, H.; Wang, L.; Lv, Q.; Wang, Y.; Li, N.; and Sun, D. 2022. DMalNet: Dynamic malware analysis based on API feature engineering and graph learning. *Computers & Security*, 122: 102872.

Li, S.; Han, X.; and Bai, J. 2023. AdapterGNN: Parameter-Efficient Fine-Tuning Improves Generalization in GNNs. arXiv:2304.09595.

Liu, Y.; Meng, R.; Joty, S.; Savarese, S.; Xiong, C.; Zhou, Y.; and Yavuz, S. 2024. CodeXEmbed: A Generalist Embedding Model Family for Multiligual and Multi-task Code Retrieval. arXiv:2411.12644.

Liu, Z.; Yu, X.; Fang, Y.; and Zhang, X. 2023. GraphPrompt: Unifying Pre-Training and Downstream Tasks for Graph Neural Networks. arXiv:2302.08043.

Malhotra, V.; Potika, K.; and Stamp, M. 2024. A comparison of graph neural networks for malware classification. *Journal of Computer Virology and Hacking Techniques*, 20(1): 53–69.

Mikolov, T.; Chen, K.; Corrado, G.; and Dean, J. 2013. Efficient Estimation of Word Representations in Vector Space. arXiv:1301.3781.

Prakash, N.; Shaham, T. R.; Haklay, T.; Belinkov, Y.; and Bau, D. 2024. Fine-Tuning Enhances Existing Mechanisms: A Case Study on Entity Tracking. arXiv:2402.14811.

Quiñonero-Candela, J.; Sugiyama, M.; Schwaighofer, A.; and Lawrence, N. D. 2022. *Dataset shift in machine learning*. Mit Press.

Rampášek, L.; Galkin, M.; Dwivedi, V. P.; Luu, A. T.; Wolf, G.; and Beaini, D. 2023. Recipe for a General, Powerful, Scalable Graph Transformer. arXiv:2205.12454.

Shirzad, H.; Velingker, A.; Venkatachalam, B.; Sutherland, D. J.; and Sinop, A. K. 2023. Exphormer: Sparse Transformers for Graphs. arXiv:2303.06147.

Sood, G. 2021. *virustotal: R Client for the virustotal API*.

Srivastava, N.; Hinton, G.; Krizhevsky, A.; Sutskever, I.; and Salakhutdinov, R. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1): 1929–1958.

Sun, X.; Cheng, H.; Li, J.; Liu, B.; and Guan, J. 2023. All in One: Multi-task Prompting for Graph Neural Networks. arXiv:2307.01504.

Wang, D.; Shelhamer, E.; Liu, S.; Olshausen, B.; and Darrell, T. 2021. Tent: Fully Test-time Adaptation by Entropy Minimization. arXiv:2006.10726.

Wang, Y.; Li, C.; Jin, W.; Li, R.; Zhao, J.; Tang, J.; and Xie, X. 2022. Test-Time Training for Graph Neural Networks. arXiv:2210.08813.

Weinberger, K.; Dasgupta, A.; Langford, J.; Smola, A.; and Attenberg, J. 2009. Feature hashing for large scale multitask learning. In *Proceedings of the 26th annual international conference on machine learning*, 1113–1120.

Weisfeiler, B.; and Leman, A. 1968. The reduction of a graph to canonical form and the algebra which appears therein. *nti, Series*, 2(9): 12–16.

Wolf, T.; Debut, L.; Sanh, V.; Chaumond, J.; Delangue, C.; Moi, A.; Cistac, P.; Rault, T.; Louf, R.; Funtowicz, M.; Davison, J.; Shleifer, S.; von Platen, P.; Ma, C.; Jernite, Y.; Plu, J.; Xu, C.; Scao, T. L.; Gugger, S.; Drame, M.; Lhoest, Q.; and Rush, A. M. 2020. HuggingFace's Transformers: State-of-the-art Natural Language Processing. arXiv:1910.03771.

Wu, M.; Zheng, X.; Zhang, Q.; Shen, X.; Luo, X.; Zhu, X.; and Pan, S. 2024. Graph learning under distribution shifts: A comprehensive survey on domain adaptation, out-of-distribution, and continual learning. *arXiv preprint arXiv:2402.16374*.

Xu, K.; Hu, W.; Leskovec, J.; and Jegelka, S. 2019. How Powerful are Graph Neural Networks? arXiv:1810.00826.

Yang, L.; Ciptadi, A.; Laziuk, I.; Ahmadzadeh, A.; and Wang, G. 2021. BODMAS: An Open Dataset for Learning based Temporal Analysis of PE Malware. In *4th Deep Learning and Security Workshop*.

Zhang, C.; Zhou, Q.; Huang, Y.; Tang, K.; Gui, H.; and Liu, F. 2022. Automatic Detection of Android Malware via Hybrid Graph Neural Network. *Wireless Communications and Mobile Computing*, 2022(1): 7245403.

Zhang, K.; Liu, S.; Wang, S.; Shi, W.; Chen, C.; Li, P.; Li, S.; Li, J.; and Ding, K. 2024. A survey of deep graph learning under distribution shifts: from graph out-of-distribution generalization to adaptation. *arXiv preprint arXiv:2410.19265*.

Appendix

Dataset Construction Specifications

Original MalNet Construction.

Collected from AndroZoo (Allix et al. 2016), each malware sample in MalNet (Freitas et al. 2021) is represented by its function call graph (FCG). These FCGs are generated using AndroGuard (Desnos and Gueguen 2018), with any other extracted information discarded during the process. The corresponding malware labels were collected from VirusTotal (Sood 2021), a service detecting malicious files by providing the their analysis reports from antivirus (AV) engines. These reports categorize malwares into their *families* and *types* based on their behavior and characteristics, and are unified into one final label for the dataset using Euphony (Hurier 2025). An structure example of this hierarchical labeling is shown in Fig. 2. As detection results of different AVs do not always agree, the unified labels can be very noisy, which can lead to ambiguity in classification tasks. To address this, the authors of MalNet introduced the *MalNet-Tiny* subset, which is a smaller version of the original dataset with balanced sample distribution across different classes, where each class contains exactly one malware family and one malware type classification. This subset is designed to be more manageable for training and evaluation purposes, while still retaining the essential characteristics of the original dataset.

MalNet Function Feature Specifications

We construct function node features by adapting the EMBER feature set (Anderson and Roth 2018) to Android malwares, on a per-function basis. To check for storage access, we search for strings such as "/storage/" or "/sdcard/". For the equivalent of registry access, the strings of interest are "/shared_prefs/", "Settings.Secure", "Settings.System", and "Settings.Global". MZ dropper is substituted with any sign of in-memory code execution, which exhibits in keywords such as "ClassLoader", "DexFile", "loadDex", "loadClass", "defineClass", or "loadLibrary".

We also utilize method information available to us from Androguard (Desnos and Gueguen 2018) analyzer getter functions. All numerical features are kept as is without normalization, and any string/string list features are converted into a 50-dimensional number vector using the hashing trick (Weinberger et al. 2009). We list all extracted features in Tab. 6. Note that only the first 5 features in the table are available across all methods: for example, we cannot extract any bytecode statistics from external functions as they are not defined/available in the extracted APK. These 5 features form the **Trim** variant as described in the main text.

Edge Features for Malware FCGs

For edge features, we note that the relationship between two functions is an invocation, and thus any information about it (e.g. passed parameters, return values) requires an inspection of the call stack, which is only available at run time (Li et al. 2022). As we do not conduct dynamic analysis in this work, we do not extract any edge features to enrich the FCG representation.

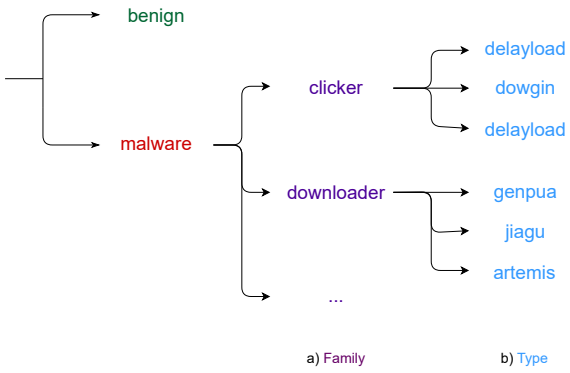

Fig. 2: Example of the hierarchical malware labeling in MalNet.

MalNet-Tiny Variants Specifications

We construct MalNet-Tiny-Common to have the same malware families but different malware types, to illustrate the final trained model's ability to deal with covariate shift. We construct MalNet-Tiny-Distinct to have different malware families/types, to illustrate the final trained model's ability to deal with domain shift. The malware families/types are listed in Tab. 7. For sample selection, we only select malwares with under 5000 nodes/functions to match with MalNet-Tiny. On malware type selection, there are two important details:

- With the benign class not having a different type (i.e. just *benign/benign*), where we select a disjoint set of malware samples from the original dataset.
- For some malware families, there are no subtype that has 1000 samples. To deal with this issue, we opt for including other subtypes with as little overlap as possible, denoted as *mixed* in Tab. 7.

While we have done our best to avoid label duplicates, it is unavoidable given the limitation of the original MalNet. We also believe that this labeling noise is not detrimental, as our classification results still reach over 94% for both datasets.

Detailed Experiment Setup

GNN Architectures

Depending on the model architecture, different feature collation schemes may be better or worse. For example, traditional message-passing graph neural networks propagate information from only the neighboring nodes, and thus may benefit from any additional features. In contrast, Transformer-based models can attend to all nodes in the graph (e.g. a global pseudonode), which may cause noisy features of an unrelated node to negatively affect the aggregation of another node's features. Therefore, we experiment

	Feature	Type	Embedding
Names	Class name	String list[a]	Hashing trick
	Method name	String	Hashing trick
Method signature	Number of parameters	Integer	As-is
	Parameter types	String list	Hashing trick
	Return type	String	Hashing trick
Method misc.	Access flags	Binary[b]	Multi-hot
	Number of local registers	Integer	As-is
Code	Length	Integer	As-is
	Byte histogram	Integer list	Distribution[c]
	Byte-entropy histogram	Integer list	Distribution
Instructions	Length	Integer	As-is
	Opcode names	String list	Hashing trick
Strings	Contains invalid characters	Boolean	As-is
	String literal	String	Hashing trick
	Number of strings	Integer	As-is
	Average string length	Float	As-is
	Character histogram	Integer list	Distribution
	Character entropy	Float	As-is
	Number of external paths	Integer	As-is
	Number of URLs	Integer	As-is
	Number of IP addresses	Integer	As-is
	Registry modifications	Integer	As-is
	In-memory executions	Integer	As-is
Misc.	Are instructions cached?	Boolean	As-is

Tab. 6: All features used for MalNet-Tiny-Feature.

[a] Class names are tokenized, e.g. `["java", "lang", "Object"]`

[b] The concerned flags are: public, private, protected, static, final, synchronized, bridge, varargs, native, interface, abstract, strictfp, synthetic, constructor.

[c] The number list is normalized into a proper distribution.

with different graph neural network architectures to see how they perform under different settings:

- Graph Convolutional Network (GCN) (Kipf and Welling 2017): a traditional message-passing graph neural network that uses a convolutional operation to aggregate features from neighboring nodes.
- Graph Isomorphism Network (GIN) (Xu et al. 2019): a more powerful message-passing architecture that can distinguish between different graph structures, based on the 1-WL test (Weisfeiler and Leman 1968).
- GPS (Rampášek et al. 2023): a hybrid Transformer architecture that combines message-passing and attention mechanisms to capture both local and global information in the graph.
- Exphormer (Shirzad et al. 2023): a sparse hybrid Transformer architecture that improves upon GPS via an improved attention mechanism, allowing better information propagation.

Adaptation Methods

We select these methods such that they do not alter the upstream training process, and cover a wide range of approaches. For test-time adaptation methods:

- Tent (Wang et al. 2021): a generic approach that finetunes only the normalization layers per inference.
- T3A (Iwasawa and Matsuo 2021): a generic approach collecting inferenced samples' embeddings as class prototypes to replace the original classifier weights.
- GTrans (Jin et al. 2023): graph TTA for node classification that augments input graphs to optimize a surrogate loss, which we adapted to work with the graph level.

For domain adaptation approaches, we select the following baselines:

- k-NN Probe: utilizes the k-NN classifier fitted to the finetuning dataset to probe the model's predictions on the downstream task, and replaces the classifier head with a k-NN classifier (Chen et al. 2020).
- Finetuning: fully finetunes the model on the downstream dataset for a few epochs, a common approach in

MalNet-Tiny	MalNet-Tiny-Common	MalNet-Tiny-Distinct
addisplay/kuguo	addisplay/dowgin	spr/lootor
adware/airpush	adware/startapp	clicker++trojan/dowgin
benign/benign	benign/benign	riskware/nandrobox
downloader/jiagu	downloader: tencentprotect, openconnection, dowgin, genpua, hiddenapps, artemis	malware/sdi, tencentprotect, deepscan, fakeind, genpua, fwad, hiddenapps, oddjs, azshouyou, cve, jiagu
trojan/artemis	trojan/deng	spyware/smspay, ginmaster, genbl, wapsx, zwalls, opfake, lmmob, admogo, deng, adwo, axent, multiad, plankton, dowgin

Tab. 7: Malware labels in MalNet-Tiny, MalNet-Tiny-Common, and MalNet-Tiny-Distinct.

transfer learning (Church, Chen, and Ma 2021; Prakash et al. 2024).

- AdapterGNN (Li, Han, and Bai 2023): a graph-specific parameter-efficient finetuning method which adds an adapter to each message passing layer, and finetunes only the adapters' parameters.

Implementation Modifications

Readout function. After message-passing, we use the global max pooling operation to aggregate the node features into a single graph-level representation, as is also used in the Transformer-based architectures (Rampášek et al. 2023; Shirzad et al. 2023). This readout choice has a nice interpretation: a program is the product of all its functions, and thus if one function behaves like a malware, the whole program is likely malicious. This is in contrast to the more traditional global mean pooling operation, which may dilute the effect of a single malicious function by averaging it with other benign functions.

Classifier initialization. Across all experiments on MalNet-Tiny-Common, we start finetuning on the checkpoint as is. For MalNet-Tiny-Distinct, we reinitialize the classifier head for finetuning-based methods, as the model cannot adapt to new malware families without retraining the classifier.

Prune. For *Prune*, we omit removing isolated nodes during data preparation to prevent empty graphs. This is because for some malwares, all of its code-containing functions do not call each other (and e.g. only call APIs), and thus become completely isolated after all other nodes are pruned.

GTrans. We adapt the method as-is to graph classification by keeping all perturbation schemes and hyperparameters unchanged from the original code onto our graph classifiers, which only differ from their node classifying models in that the former has a readout step before classification. The only exception is that we disabled adjacency perturbation, as some graph architectures did not support edge features. While the original work did not experiment on graph classification, it mentioned that the method can theoretically be applied to the setting (Jin et al. 2023), motivating us to adapt it to our problem.

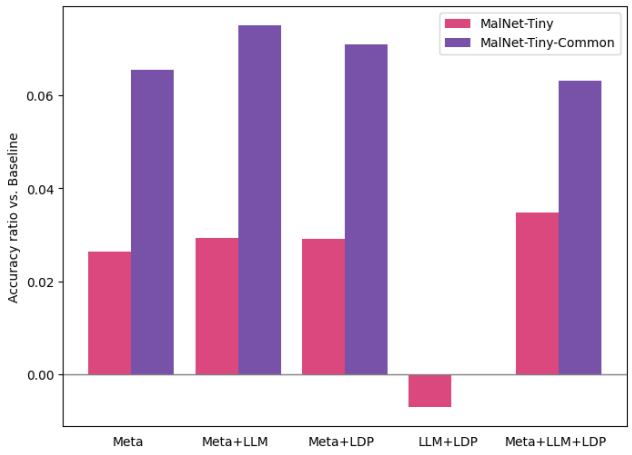

Fig. 3: Average models' accuracy ratio when compared to the baseline across different feature configurations for MalNet-Tiny and MalNet-Tiny-Common. LLM+LDP for MalNet-Tiny-Common is too close to the baseline to be visible.

Hardware and Implementation

All experiments are conducted on a single NVIDIA RTX 6000 Ada with 48Gb of memory. All runs are seeded with the same seed for reproducibility, using the hyperparameter configurations listed in the original paper (Rampášek et al. 2023; Shirzad et al. 2023) for a fair comparison. Each of our experiment takes 1-2 hours to run, depending on the model architecture in use.

Additional Experiment Results

LLM-only Features

Figure 3 presents the aggregated model accuracy across our experiments. We can see that LLM+LDP does not improve performance in general, warranting a deeper analysis for a future work. Note that LLM+LDP is effectively LLM-only, since some malware samples does not contain any code to extract features, and thus without a default LDP fallback they would yield an empty vector.

Additional Related Works

Graph Representations for Android Malwares

Before the the release of MalNet, αCyber (Hou et al. 2019) devised a graph-based feature extraction on malware metadata, representing all available data as one giant heterogeneous graph. This approach was not generalizable due to the exploding space complexity, and that the authors worked on a private propriety dataset. AMDroid (Ge et al. 2019) proposed to use function call graphs (FCGs) as the graph representation of Android malwares, but uses a random-walk based feature extraction method instead of a graph neural network (GNN). Alasmary et al. (Alasmary et al. 2019) extracted a Control Flow Graph as the representation instead, and focused on the Internet-of-Thing domain. CGDroid (Feng et al. 2021) proposed training a word2vec (Mikolov et al. 2013) embedding model for bytecode, with manual annotation of API security levels and permission lists as hints for the model to work with. HyGNN-Mal (Zhang et al. 2022) took a different approach and extracted the abstract syntax tree as the graph representation. BejaGNN (Feng et al. 2023) also used word2vec to generate features, but instead utilized inter-procedural control flow graph to for the graph structure.

Distribution Shift for Graph Machine Learning

To alleviate the effect of distribution shift in graph machine learning, a number of methods have been proposed. Some approaches are finetuning-based, which update the model parameters to adapt to the new distribution – latest development have been focusing on being parameter-efficient through the use of an adapter (Gui, Ye, and Xiao 2023; Li, Han, and Bai 2023). Others worked on domain adaptation techniques, such as applying self-supervised learning on a pretrained model at test-time (Chen et al. 2022; Wang et al. 2022), or adapting the input graphs themselves to match the learned training distribution (Jin et al. 2023; Ju et al. 2023; Hsu et al. 2025). A more novel approach to graph adaptation is graph prompt tuning, which uses a prompt to adapt the model to the new distribution without modifying the model parameters (Liu et al. 2023; Sun et al. 2023; Fang et al. 2024; Fu, He, and Li 2025). However, similar to prompt tuning in natural language processing, these methods typically require pretraining on a large corpus of graphs; thus, they are not applicable to many graph classification problems with limited training data.